Chapter 4

Funciones

4.1 Llamadas a funciones

En el contexto de la programación, una *función* es una secuencia de sentencias que realizan una operación y que reciben un nombre. Cuando se define una función, se especifica el nombre y la secuencia de sentencias. Más adelante, se puede "llamar" a la función por ese nombre. Ya hemos visto un ejemplo de una *llamada a una función*:

```
>>> type(32)
<class 'int'>
```

El nombre de la función es `type`. La expresión entre paréntesis recibe el nombre de *argumento* de la función. El argumento es un valor o variable que se pasa a la función como parámetro de entrada. El resultado de la función `type` es el tipo del argumento.

Es habitual decir que una función "toma" (o recibe) un argumento y "retorna" (o devuelve) un resultado. El resultado se llama *valor de retorno*.

4.2 Funciones internas

Python proporciona un número importante de funciones internas, que pueden ser usadas sin necesidad de tener que definirlas previamente. Los creadores de Python han escrito un conjunto de funciones para resolver problemas comunes y las han incluido en Python para que las podamos utilizar.

Las funciones `max` y `min` nos darán respectivamente el valor mayor y menor de una lista:

```
>>> max('¡Hola, mundo!')
'u'
>>> min('¡Hola, mundo!')
' '
>>>
```

La función `max` nos dice cuál es el "carácter más grande" de la cadena (que resulta ser la letra "u"), mientras que la función `min` nos muestra el carácter más pequeño (que en ese caso es un espacio).

Otra función interna muy común es `len`, que nos dice cuántos elementos hay en su argumento. Si el argumento de `len` es una cadena, nos devuelve el número de caracteres que hay en la cadena.

```
>>> len('Hola, mundo')
11
>>>
```

Estas funciones no se limitan a buscar en cadenas. Pueden operar con cualquier conjunto de valores, como veremos en los siguientes capítulos.

Se deben tratar los nombres de las funciones internas como si fueran palabras reservadas (es decir, evita usar "max" como nombre para una variable).

4.3 Funciones de conversión de tipos

Python también proporciona funciones internas que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte en un entero, si puede, o se queja si no puede:

```
>>> int('32')
32
>>> int('Hola')
ValueError: invalid literal for int() with base 10: 'Hola'
```

`int` puede convertir valores en punto flotante a enteros, pero no los redondea; simplemente corta y descarta la parte decimal:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` convierte enteros y cadenas en números de punto flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` convierte su argumento en una cadena:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Funciones matemáticas

Python tiene un módulo matemático (`math`), que proporciona la mayoría de las funciones matemáticas habituales. Antes de que podamos utilizar el módulo, deberemos importarlo:

```
>>> import math
```

Esta sentencia crea un *objeto módulo* llamado math. Si se imprime el objeto módulo, se obtiene cierta información sobre él:

```
>>> print(math)
<module 'math' (built-in)>
```

El objeto módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de esas funciones, es necesario especificar el nombre del módulo y el nombre de la función, separados por un punto (también conocido en inglés como *peŕiod*). Este formato recibe el nombre de *notación punto*.

```
>>> relacion = int_senal / int_ruido
>>> decibelios = 10 * math.log10(relacion)

>>> radianes = 0.7
>>> altura = math.sin(radianes)
```

El primer ejemplo calcula el logaritmo en base 10 de la relación señal-ruido. El módulo math también proporciona una función llamada `log` que calcula logaritmos en base `e`.

El segundo ejemplo calcula el seno de la variable `radianes`. El nombre de la variable es una pista de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes. Para convertir de grados a radianes, hay que dividir por 360 y multiplicar por 2π:

```
>>> grados = 45
>>> radianes = grados / 360.0 * 2 * math.pi
>>> math.sin(radianes)
0.7071067811865476
```

La expresión `math.pi` toma la variable `pi` del módulo math. El valor de esa variable es una aproximación de π, con una precisión de unos 15 dígitos.

Si sabes de trigonometría, puedes comprobar el resultado anterior, comparándolo con la raíz cuadrada de dos dividida por dos:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5 Números aleatorios

A partir de las mismas entradas, la mayoría de los programas generarán las mismas salidas cada vez, que es lo que llamamos comportamiento *determinista*. El determinismo normalmente es algo bueno, ya que esperamos que la misma operación nos proporcione siempre el mismo resultado. Para ciertas aplicaciones, sin embargo, querremos que el resultado sea impredecible. Los juegos son el ejemplo obvio, pero hay más.

Conseguir que un programa sea realmente no-determinista no resulta tan fácil, pero hay modos de hacer que al menos lo parezca. Una de ellos es usar *algoritmos* que generen números *pseudoaleatorios*. Los números pseudoaleatorios no son verdaderamente aleatorios, ya que son generados por una operación determinista, pero si sólo nos fijamos en los números resulta casi imposible distinguirlos de los aleatorios de verdad.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (a los que simplemente llamaremos "aleatorios" de ahora en adelante).

La función `random` devuelve un número flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0, pero no 1.0). Cada vez que se llama a `random`, se obtiene el número siguiente de una larga serie. Para ver un ejemplo, ejecuta este bucle:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Este programa produce la siguiente lista de 10 números aleatorios entre 0.0 y hasta (pero no incluyendo) 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Ejercicio 1: Ejecuta el programa en tu sistema y observa qué números obtienes.

La función `random` es solamente una de las muchas que trabajan con números aleatorios. La función `randint` toma los parámetros `inferior` y `superior`, y devuelve un entero entre `inferior` y `superior` (incluyendo ambos extremos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para elegir un elemento de una secuencia aleatoriamente, se puede usar `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

El módulo `random` también proporciona funciones para generar valores aleatorios de varias distribuciones continuas, incluyendo gaussiana, exponencial, gamma, y unas cuantas más.

4.6 Añadiendo funciones nuevas

Hasta ahora, sólo hemos estado usando las funciones que vienen incorporadas en Python, pero es posible añadir también funciones nuevas. Una *definición de función* especifica el nombre de una función nueva y la secuencia de sentencias que se ejecutan cuando esa función es llamada. Una vez definida una función, se puede reutilizar una y otra vez a lo largo de todo el programa.

He aquí un ejemplo:

```
def muestra_estribillo():
    print('Soy un leñador, qué alegría.')
    print('Duermo toda la noche y trabajo todo el día.')
```

`def` es una palabra clave que indica que se trata de una definición de función. El nombre de la función es `muestra_estribillo`. Las reglas para los nombres de las funciones son los mismos que para las variables: se pueden usar letras, números y algunos signos de puntuación, pero el primer carácter no puede ser un número. No se puede usar una palabra clave como nombre de una función, y se debería evitar también tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no toma ningún argumento. Más tarde construiremos funciones que reciban argumentos de entrada.

La primera línea de la definición de la función es llamada la *cabecera*; el resto se llama el *cuerpo*. La cabecera debe terminar con dos-puntos (:), y el cuerpo debe ir indentado. Por convención, el indentado es siempre de cuatro espacios. El cuerpo puede contener cualquier número de sentencias.

Las cadenas en la sentencia print están encerradas entre comillas. Da igual utilizar comillas simples que dobles; la mayoría de la gente prefiere comillas simples, excepto en aquellos casos en los que una comilla simple (que también se usa como apostrofe) aparece en medio de la cadena.

Si escribes una definición de función en modo interactivo, el intérprete mostrará puntos suspensivos (. . .) para informarte de que la definición no está completa:

```
>>> def muestra_estribillo():
...     print 'Soy un leñador, qué alegría.'
...     print 'Duermo toda la noche y trabajo todo el día.'
...
```

Para finalizar la función, debes introducir una línea vacía (esto no es necesario en un script).

Al definir una función se crea una variable con el mismo nombre.

```
>>> print(muestra_estribillo)
<function muestra_estribillo at 0xb7e99e9c>
>>> print(type(muestra_estribillo))
<type 'function'>
```

El valor de `muestra_estribillo` es *function object* (objeto función), que tiene como tipo "function".

La sintaxis para llamar a nuestra nueva función es la misma que usamos para las funciones internas:

```
>>> muestra_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Una vez que se ha definido una función, puede usarse dentro de otra. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada `repite_estribillo`:

```
def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()
```

Y después llamar a `repite_estribillo`:

```
>>> repite_estribillo()
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
Soy un leñador, qué alegría.
Duermo toda la noche y trabajo todo el día.
```

Pero en realidad la canción no es así.

4.7 Definición y usos

Reuniendo los fragmentos de código de las secciones anteriores, el programa completo sería algo como esto:

```
def muestra_estribillo():
    print('Soy un leñador, que alegría.')
    print('Duermo toda la noche y trabajo todo el día.')

def repite_estribillo():
    muestra_estribillo()
    muestra_estribillo()

repite_estribillo()

# Código: https://es.py4e.com/code3/lyrics.py
```

Este programa contiene dos definiciones de funciones: `muestra_estribillo` y `repite_estribillo`. Las definiciones de funciones son ejecutadas exactamente igual que cualquier otra sentencia, pero su resultado consiste en crear objetos del tipo función. Las sentencias dentro de cada función son ejecutadas solamente cuando se llama a esa función, y la definición de una función no genera ninguna salida.

Como ya te imaginarás, es necesario crear una función antes de que se pueda ejecutar. En otras palabras, la definición de la función debe ser ejecutada antes de que la función se llame por primera vez.

Ejercicio 2: Desplaza la última línea del programa anterior hacia arriba, de modo que la llamada a la función aparezca antes que las definiciones. Ejecuta el programa y observa qué mensaje de error obtienes.

Ejercicio 3: Desplaza la llamada de la función de nuevo hacia el final, y coloca la definición de `muestra_estribillo` después de la definición de `repite_estribillo`. ¿Qué ocurre cuando haces funcionar ese programa?

4.8 Flujo de ejecución

Para asegurarnos de que una función está definida antes de usarla por primera vez, es necesario saber el orden en que las sentencias son ejecutadas, que es lo que llamamos el *flujo de ejecución*.

La ejecución siempre comienza en la primera sentencia del programa. Las sentencias son ejecutadas una por una, en orden de arriba hacia abajo.

Las *definiciones* de funciones no alteran el flujo de la ejecución del programa, pero recuerda que las sentencias dentro de una función no son ejecutadas hasta que se llama a esa función.

Una llamada a una función es como un desvío en el flujo de la ejecución. En vez de pasar a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta todas las sentencias que hay allí, y después vuelve al punto donde lo dejó.

Todo esto parece bastante sencillo, hasta que uno recuerda que una función puede llamar a otra. Cuando está en mitad de una función, el programa puede tener que ejecutar las sentencias de otra función. Pero cuando está ejecutando esa nueva función, ¡tal vez haya que ejecutar todavía más funciones!

Afortunadamente, Python es capaz de llevar el seguimiento de dónde se encuentra en cada momento, de modo que cada vez que completa la ejecución de una función, el programa vuelve al punto donde lo dejó en la función que había llamado a esa. Cuando esto le lleva hasta el final del programa, simplemente termina.

¿Cuál es la moraleja de esta extraña historia? Cuando leas un programa, no siempre te convendrá hacerlo de arriba a abajo. A veces tiene más sentido seguir el flujo de la ejecución.

4.9 Parámetros y argumentos

Algunas de las funciones internas que hemos visto necesitan argumentos. Por ejemplo, cuando se llama a `math.sin`, se le pasa un número como argumento. Algunas funciones necesitan más de un argumento: `math.pow` toma dos, la base y el exponente.

Dentro de las funciones, los argumentos son asignados a variables llamadas *parámetros*. A continuación mostramos un ejemplo de una función definida por el usuario que recibe un argumento:

```
def muestra_dos_veces(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro llamado `bruce`. Cuando la función es llamada, imprime el valor del parámetro (sea éste lo que sea) dos veces.

Esta función funciona con cualquier valor que pueda ser mostrado en pantalla.

```
>>> muestra_dos_veces('Spam')
Spam
Spam
>>> muestra_dos_veces(17)
17
17
>>> muestra_dos_veces(math.pi)
3.14159265359
3.14159265359
```

Las mismas reglas de composición que se aplican a las funciones internas, también se aplican a las funciones definidas por el usuario, de modo que podemos usar cualquier tipo de expresión como argumento para `muestra_dos_veces`:

```
>>> muestra_dos_veces('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> muestra_dos_veces(math.cos(math.pi))
-1.0
-1.0
```

El argumento es evaluado antes de que la función sea llamada, así que en los ejemplos, la expresión `Spam *4` y `math.cos(math.pi)` son evaluadas sólo una vez.

También se puede usar una variable como argumento:

```
>>> michael = 'Eric, la medio-abeja.'
>>> muestra_dos_veces(michael)
Eric, la medio-abeja.
Eric, la medio-abeja.
```

El nombre de la variable que pasamos como argumento, (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se haya llamado al valor en origen (en la llamada); dentro de `muestra_dos_veces`, siempre se llamará `bruce`.

4.10 Funciones productivas y funciones estériles

Algunas de las funciones que estamos usando, como las matemáticas, producen resultados; a falta de un nombre mejor, las llamaremos *funciones productivas* (fruitful functions). Otras funciones, como `muestra_dos_veces`, realizan una acción, pero no devuelven un valor. A esas las llamaremos *funciones estériles* (void functions).

Cuando llamas a una función productiva, casi siempre querrás hacer luego algo con el resultado; por ejemplo, puede que quieras asignarlo a una variable o usarlo como parte de una expresión:

```
x = math.cos(radians)
aurea = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.23606797749979
```

Pero en un script, si llamas a una función productiva y no almacenas el resultado de la misma en una variable, ¡el valor de retorno se desvanece en la niebla!

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero dado que no almacena el resultado en una variable ni lo muestra, no resulta en realidad muy útil.

Las funciones estériles pueden mostrar algo en la pantalla o tener cualquier otro efecto, pero no devuelven un valor. Si intentas asignar el resultado a una variable, obtendrás un valor especial llamado `None` (nada).

```
>>> resultado = muestra_dos_veces('Bing')
Bing
Bing
>>> print(resultado)
None
```

El valor `None` no es el mismo que la cadena "None". Es un valor especial que tiene su propio tipo:

```
>>> print(type(None))
<class 'NoneType'>
```

Para devolver un resultado desde una función, usamos la sentencia `return` dentro de ella. Por ejemplo, podemos crear una función muy simple llamada `sumados`, que suma dos números y devuelve el resultado.

```
def sumados(a, b):
    suma = a + b
    return suma

x = sumados(3, 5)
print(x)

# Código: https://es.py4e.com/code3/addtwo.py
```

Cuando se ejecuta este script, la sentencia `print` mostrará "8", ya que la función `sumados` ha sido llamada con 3 y 5 como argumentos. Dentro de la función, los parámetros `a` y `b` equivaldrán a 3 y a 5 respectivamente. La función calculó la suma de ambos número y la guardó en una variable local a la función llamada `suma`. Después usó la sentencia `return` para enviar el valor calculado de vuelta al código de llamada como resultado de la función, que fue asignado a la variable `x` y mostrado en pantalla.

4.11 ¿Por qué funciones?

Puede no estar muy claro por qué merece la pena molestarse en dividir un programa en funciones. Existen varias razones:

- El crear una función nueva te da la oportunidad de dar nombre a un grupo de sentencias, lo cual hace tu programa más fácil de leer, entender y depurar.
- Las funciones pueden hacer un programa más pequeño, al eliminar código repetido. Además, si quieres realizar cualquier cambio en el futuro, sólo tendrás que hacerlo en un único lugar.
- Dividir un programa largo en funciones te permite depurar las partes de una en una y luego ensamblarlas juntas en una sola pieza.
- Las funciones bien diseñadas a menudo resultan útiles para otros muchos programas. Una vez que has escrito y depurado una, puedes reutilizarla.

A lo largo del resto del libro, a menudo usaremos una definición de función para explicar un concepto. Parte de la habilidad de crear y usar funciones consiste en llegar a tener una función que represente correctamente una idea, como "encontrar el valor más pequeño en una lista de valores". Más adelante te mostraremos el código para encontrar el valor más pequeño de una lista de valores y te lo presentaremos como una función llamada `min`, que toma una lista de valores como argumento y devuelve el menor valor de esa lista.

4.12 Depuración

Si estás usando un editor de texto para escribir tus propios scripts, puede que tengas problemas con los espacios y tabulaciones. El mejor modo de evitar esos problemas es usar espacios exclusivamente (no tabulaciones). La mayoría de los editores de texto que reconocen Python lo hacen así por defecto, aunque hay algunos que no.

Las tabulaciones y los espacios normalmente son invisibles, lo cual hace que sea difícil depurar los errores que se pueden producir, así que mejor busca un editor que gestione el indentado por ti.

Tampoco te olvides de guardar tu programa antes de hacerlo funcionar. Algunos entornos de desarrollo lo hacen automáticamente, pero otros no. En ese caso, el programa que estás viendo en el editor de texto puede no ser el mismo que estás ejecutando en realidad.

¡La depuración puede llevar mucho tiempo si estás haciendo funcionar el mismo programa con errores una y otra vez!

Asegúrate de que el código que estás examinando es el mismo que estás ejecutando. Si no estás seguro, pon algo como `print("hola")` al principio del programa y hazlo funcionar de nuevo. Si no ves `hola` en la pantalla, ¡es que no estás ejecutando el programa correcto!

4.13 Glosario

algoritmo Un proceso general para resolver una categoría de problemas.

argumento Un valor proporcionado a una función cuando ésta es llamada. Ese valor se asigna al parámetro correspondiente en la función.

cabecera La primera línea de una definición de función.

cuerpo La secuencia de sentencias dentro de la definición de una función.

composición Uso de una expresión o sentencia como parte de otra más larga,

definición de función Una sentencia que crea una función nueva, especificando su nombre, parámetros, y las sentencias que ejecuta.

determinístico Perteneciente a un programa que hace lo mismo cada vez que se ejecuta, a partir de las mismas entradas.

función Una secuencia de sentencias con un nombre que realizan alguna operación útil. Las funciones pueden tomar argumentos o no, y pueden producir un resultado o no.

función productiva (fruitful function) Una función que devuelve un valor.

función estéril (void function) Una función que no devuelve ningún valor.

flujo de ejecución El orden en el cual se ejecutan las sentencias durante el funcionamiento de un programa.

llamada a función Una sentencia que ejecuta una función. Consiste en el nombre de la función seguido por una lista de argumentos.

notación punto La sintaxis para llamar a una función de otro módulo, especificando el nombre del módulo seguido por un punto y el nombre de la función.

objeto función Un valor creado por una definición de función. El nombre de la función es una variable que se refiere al objeto función.

objeto módulo Un valor creado por una sentencia `import`, que proporciona acceso a los datos y código definidos en un módulo.

parámetro Un nombre usado dentro de una función para referirse al valor pasado como argumento.

pseudoaleatorio Perteneciente a una secuencia de números que parecen ser aleatorios, pero son generados por un programa determinista.

sentencia import Una sentencia que lee un archivo módulo y crea un objeto módulo.

valor de retorno El resultado de una función. Si una llamada a una función es usada como una expresión, el valor de retorno es el valor de la expresión.

4.14 Ejercicios

Ejercicio 4: ¿Cuál es la utilidad de la palabra clave "def" en Python?

a) Es una jerga que significa "este código es realmente estupendo"
b) Indica el comienzo de una función
c) Indica que la siguiente sección de código indentado debe ser almacenada para usarla más tarde
d) b y c son correctas ambas
e) Ninguna de las anteriores

Ejercicio 5: ¿Qué mostrará en pantalla el siguiente programa Python?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Ejercicio 6: Reescribe el programa de cálculo del salario, con tarifa-y-media para las horas extras, y crea una función llamada `calculo_salario` que reciba dos parámetros (`horas` y `tarifa`).

```
Introduzca Horas: 45
Introduzca Tarifa: 10
Salario: 475.0
```

Ejercicio 7: Reescribe el programa de calificaciones del capítulo anterior usando una función llamada `calcula_calificacion`, que reciba una puntuación como parámetro y devuelva una calificación como cadena.

```
Puntuación Calificación
> 0.9      Sobresaliente
> 0.8      Notable
> 0.7      Bien
> 0.6      Suficiente
<= 0.6     Insuficiente


Introduzca puntuación: 0.95
Sobresaliente
```

```
Introduzca puntuación: perfecto
Puntuación incorrecta

Introduzca puntuación: 10.0
Puntuación incorrecta

Introduzca puntuación: 0.75
Bien

Introduzca puntuación: 0.5
Insuficiente
```

Ejecuta el programa repetidamente para probar con varios valores de entrada diferentes.