# 8 Funciones



En este capítulo aprenderás a escribir *functions*, que son bloques de código con nombre diseñados para realizar una tarea concreta. Cuando quieras realizar una tarea concreta que hayas definido en una función, escribe en *call* la función responsable de ella. Si necesitas realizar esa tarea varias veces a lo largo de tu programa, no necesitas escribir todo el código para la misma tarea una y otra vez; sólo tienes que llamar a la función dedicada a gestionar esa tarea, y la llamada indica a Python que ejecute el código dentro de la función. Verás que el uso de funciones hace que tus programas sean más fáciles de escribir, leer, probar y corregir.

En este capítulo también aprenderás diversas formas de pasar información a las funciones. Aprenderás a escribir ciertas funciones cuyo trabajo principal es mostrar información y otras funciones diseñadas para procesar datos y devolver un valor o conjunto de valores. Por último, aprenderás a almacenar las funciones en archivos independientes llamados *modules* para ayudarte a organizar los archivos principales de tu programa.

## Definir una función

Aquí tienes una función sencilla llamada greet\_user() que imprime un saludo:

**greeter.py**

def greet\_user():  
 """Display a simple greeting."""  
 print("Hello!")  
  
greet\_user()

Este ejemplo muestra la estructura más simple de una función. La primera línea utiliza la palabra clave def para informar a Python de que estás definiendo una función. Se trata de *function definition*, que indica a Python el nombre de la función y, si procede, qué tipo de información necesita la función para hacer su trabajo. Los paréntesis contienen esa información. En este caso, el nombre de la función es greet\_user(), y no necesita información para realizar su trabajo, por lo que sus paréntesis están vacíos. (Aún así, los paréntesis son necesarios.) Por último, la definición termina con dos puntos.

Las líneas sangradas que siguen a def greet\_user(): constituyen la *body* de la función. El texto de la segunda línea es un comentario llamado *docstring*, que describe lo que hace la función. Cuando Python genera documentación para las funciones de tus programas, busca una cadena inmediatamente después de la definición de la función. Estas cadenas suelen ir entre comillas triples, lo que te permite escribir varias líneas.

La línea print("Hello!") es la única línea de código real en el cuerpo de esta función, por lo que greet\_user() sólo tiene un trabajo: print("Hello!").

Cuando quieras utilizar esta función, tienes que llamarla. Un *function call* indica a Python que ejecute el código de la función. Para *call* una función, escribe el nombre de la función, seguido de cualquier información necesaria entre paréntesis. Como aquí no se necesita ninguna información, llamar a nuestra función es tan sencillo como escribir greet\_user(). Como era de esperar, imprime Hello!:

Hello!

### Pasar información a una función

Si modificas ligeramente la función greet\_user(), puede saludar al usuario por su nombre. Para que la función lo haga, introduce username entre los paréntesis de la definición de la función en def greet\_user(). Añadiendo aquí username, permites que la función acepte cualquier valor de username que especifiques. Ahora la función espera que proporciones un valor para username cada vez que la llames. Cuando llames a greet\_user(), puedes pasarle un nombre, como 'jesse', dentro de los paréntesis:

def greet\_user(username):  
 """Display a simple greeting."""  
 print(f"Hello, {username.title()}!")  
  
greet\_user('jesse')

Al introducir greet\_user('jesse') se llama a greet\_user() y se proporciona a la función la información que necesita para ejecutar la llamada a print(). La función acepta el nombre que le has pasado y muestra el saludo correspondiente a ese nombre:

Hello, Jesse!

Del mismo modo, al introducir greet\_user('sarah') se llama a greet\_user(), se le pasa 'sarah', y se imprime Hello, Sarah! Puedes llamar a greet\_user() tantas veces como quieras y pasarle cualquier nombre que desees para producir una salida predecible cada vez.

### Argumentos y parámetros

En la función greet\_user() anterior, definimos greet\_user() para requerir un valor para la variable username. Una vez que llamamos a la función y le dimos la información (el nombre de una persona), imprimió el saludo correcto.

La variable username en la definición de greet\_user() es un ejemplo de *parameter*, un dato que la función necesita para hacer su trabajo. El valor 'jesse' en greet\_user('jesse') es un ejemplo de argumento. Un *argument* es un fragmento de información que se pasa de una llamada a una función. Cuando llamamos a la función, colocamos entre paréntesis el valor con el que queremos que trabaje la función. En este caso, el argumento 'jesse' se pasó a la función greet\_user(), y el valor se asignó al parámetro username.

## Nota

A veces se habla indistintamente de argumentos y parámetros. No te sorprendas si ves que las variables de una definición de función se denominan argumentos o que las variables de una llamada a una función se denominan parámetros.

## Pruébalo tú mismo

8-1. Mensaje: Escribe una función llamada display\_message() que imprima una frase contando a todo el mundo lo que estás aprendiendo en este capítulo. Llama a la función y asegúrate de que el mensaje se muestra correctamente.

8-2. Libro favorito: Escribe una función llamada favorite\_book() que acepte un parámetro, title. La función debe imprimir un mensaje, como One of my favorite books is Alice in Wonderland. Llama a la función, asegurándote de incluir el título de un libro como argumento en la llamada a la función.

## Pasar argumentos

Dado que la definición de una función puede tener varios parámetros, la llamada a una función puede necesitar varios argumentos. Puedes pasar argumentos a tus funciones de varias formas. Puedes utilizar *positional arguments*, que deben estar en el mismo orden en que se escribieron los parámetros; *keyword arguments*, donde cada argumento consta de un nombre de variable y un valor; y listas y diccionarios de valores. Veamos cada uno de ellos por separado.

### Argumentos posicionales

Cuando llamas a una función, Python debe hacer coincidir cada argumento de la llamada a la función con un parámetro de la definición de la función. La forma más sencilla de hacerlo es basándose en el orden de los argumentos proporcionados. Los valores emparejados de esta forma se denominan *positional arguments*.

Para ver cómo funciona esto, considera una función que muestra información sobre mascotas. La función nos dice qué tipo de animal es cada mascota y el nombre de la mascota, como se muestra aquí:

**pets.py**

❶ def describe\_pet(animal\_type, pet\_name):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
❷ describe\_pet('hamster', 'harry')

La definición muestra que esta función necesita un tipo de animal y el nombre del animal ❶. Cuando llamamos a describe\_pet(), necesitamos proporcionar un tipo de animal y un nombre, en ese orden. Por ejemplo, en la llamada a la función, el argumento 'hamster' se asigna al parámetro animal\_type y el argumento 'harry' se asigna al parámetro pet\_name ❷. En el cuerpo de la función, estos dos parámetros se utilizan para mostrar información sobre la mascota que se está describiendo.

La salida describe un hámster llamado Harry:

I have a hamster.  
My hamster's name is Harry.

#### Llamadas múltiples a una función

Puedes llamar a una función tantas veces como sea necesario. La descripción de una segunda mascota distinta sólo requiere una llamada más a describe\_pet():

def describe\_pet(animal\_type, pet\_name):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
describe\_pet('hamster', 'harry')  
describe\_pet('dog', 'willie')

En esta segunda llamada a la función, pasamos a describe\_pet() los argumentos 'dog' y 'willie'. Al igual que con el anterior conjunto de argumentos que utilizamos, Python hace coincidir 'dog' con el parámetro animal\_type y 'willie' con el parámetro pet\_name. Como antes, la función hace su trabajo, pero esta vez imprime los valores de un perro llamado Willie. Ahora tenemos un hámster llamado Harry y un perro llamado Willie:

I have a hamster.  
My hamster's name is Harry.  
  
I have a dog.  
My dog's name is Willie.

Llamar a una función varias veces es una forma muy eficaz de trabajar. El código que describe una mascota se escribe una vez en la función. Luego, cada vez que quieras describir una nueva mascota, llamas a la función con la información de la nueva mascota. Aunque el código para describir una mascota se ampliara a 10 líneas, podrías describir una nueva mascota en una sola línea llamando de nuevo a la función.

#### El orden importa en los argumentos posicionales

Puedes obtener resultados inesperados si confundes el orden de los argumentos en una llamada a función cuando utilizas argumentos posicionales:

def describe\_pet(animal\_type, pet\_name):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
describe\_pet('harry', 'hamster')

En esta llamada a la función, enumeramos primero el nombre y después el tipo de animal. Como el argumento 'harry' aparece primero esta vez, ese valor se asigna al parámetro animal\_type. Del mismo modo, 'hamster' se asigna a pet\_name. Ahora tenemos un "harry" llamado "Hamster":

I have a harry.  
My harry's name is Hamster.

Si obtienes resultados raros como éste, comprueba que el orden de los argumentos en tu llamada a la función coincide con el orden de los parámetros en la definición de la función.

### Argumentos de palabra clave

Un *keyword argument* es un par nombre-valor que pasas a una función. Asocias directamente el nombre y el valor dentro del argumento, de modo que cuando pases el argumento a la función, no haya confusión (no acabarás con un harry llamado Hamster). Los argumentos con palabras clave te liberan de tener que preocuparte de ordenar correctamente tus argumentos en la llamada a la función, y aclaran el papel de cada valor en la llamada a la función.

Reescribamos *pets.py* utilizando argumentos de palabra clave para llamar a describe\_pet():

def describe\_pet(animal\_type, pet\_name):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
describe\_pet(animal\_type='hamster', pet\_name='harry')

La función describe\_pet() no ha cambiado. Pero cuando llamamos a la función, le decimos explícitamente a Python con qué parámetro debe coincidir cada argumento. Cuando Python lee la llamada a la función, sabe que debe asignar el argumento 'hamster' al parámetro animal\_type y el argumento 'harry' a pet\_name. La salida muestra correctamente que tenemos un hámster llamado Harry.

El orden de los argumentos de las palabras clave no importa porque Python sabe dónde debe ir cada valor. Las dos llamadas a funciones siguientes son equivalentes:

describe\_pet(animal\_type='hamster', pet\_name='harry')  
describe\_pet(pet\_name='harry', animal\_type='hamster')

## Nota

Cuando utilices argumentos de palabra clave, asegúrate de utilizar los nombres exactos de los parámetros en la definición de la función.

### Valores por defecto

Al escribir una función, puedes definir un *default value* para cada parámetro. Si se proporciona un argumento para un parámetro en la llamada a la función, Python utiliza el valor del argumento. Si no, utiliza el valor por defecto del parámetro. Así que cuando defines un valor por defecto para un parámetro, puedes excluir el argumento correspondiente que normalmente escribirías en la llamada a la función. Utilizar valores por defecto puede simplificar tus llamadas a funciones y aclarar la forma en que se utilizan normalmente tus funciones.

Por ejemplo, si observas que la mayoría de las llamadas a describe\_pet() se utilizan para describir perros, puedes establecer el valor por defecto de animal\_type en 'dog'. Ahora cualquiera que llame a describe\_pet() para describir un perro puede omitir esa información:

def describe\_pet(pet\_name, animal\_type='dog'):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
describe\_pet(pet\_name='willie')

Cambiamos la definición de describe\_pet() para incluir un valor por defecto, 'dog', para animal\_type. Ahora, cuando se llama a la función sin especificar animal\_type, Python sabe que debe utilizar el valor 'dog' para este parámetro:

I have a dog.  
My dog's name is Willie.

Ten en cuenta que ha habido que cambiar el orden de los parámetros en la definición de la función. Como el valor por defecto hace innecesario especificar un tipo de animal como argumento, el único argumento que queda en la llamada a la función es el nombre de la mascota. Python sigue interpretándolo como un argumento posicional, por lo que si se llama a la función sólo con el nombre de la mascota, ese argumento coincidirá con el primer parámetro de la definición de la función. Esta es la razón por la que el primer parámetro debe ser pet\_name.

La forma más sencilla de utilizar esta función ahora es proporcionar sólo el nombre de un perro en la llamada a la función:

describe\_pet('willie')

Esta llamada a la función tendría el mismo resultado que el ejemplo anterior. El único argumento proporcionado es 'willie', por lo que se empareja con el primer parámetro de la definición, pet\_name. Como no se proporciona ningún argumento para animal\_type, Python utiliza el valor por defecto 'dog'.

Para describir un animal que no sea un perro, podrías utilizar una llamada a una función como ésta:

describe\_pet(pet\_name='harry', animal\_type='hamster')

Como se proporciona un argumento explícito para animal\_type, Python ignorará el valor por defecto del parámetro.

## Nota

Cuando utilices valores por defecto, cualquier parámetro con un valor por defecto debe aparecer después de todos los parámetros que no tengan valores por defecto. Esto permite que Python siga interpretando correctamente los argumentos posicionales.

### Llamadas a funciones equivalentes

Como los argumentos posicionales, los argumentos de palabra clave y los valores por defecto pueden utilizarse juntos, a menudo tendrás varias formas equivalentes de llamar a una función. Considera la siguiente definición de describe\_pet() con un valor por defecto:

def describe\_pet(pet\_name, animal\_type='dog'):

Con esta definición, siempre es necesario proporcionar un argumento para pet\_name, y este valor puede proporcionarse utilizando el formato posicional o de palabra clave. Si el animal que se describe no es un perro, debe incluirse en la llamada un argumento para animal\_type, y este argumento también puede especificarse utilizando el formato posicional o de palabra clave.

Todas las llamadas siguientes servirían para esta función:

# A dog named Willie.  
describe\_pet('willie')  
describe\_pet(pet\_name='willie')  
  
# A hamster named Harry.  
describe\_pet('harry', 'hamster')  
describe\_pet(pet\_name='harry', animal\_type='hamster')  
describe\_pet(animal\_type='hamster', pet\_name='harry')

Cada una de estas llamadas a la función tendría la misma salida que los ejemplos anteriores.

En realidad, no importa qué estilo de llamada utilices. Mientras tus llamadas a funciones produzcan la salida que deseas, utiliza el estilo que te resulte más fácil de entender.

### Evitar errores en los argumentos

Cuando empieces a utilizar funciones, no te sorprendas si encuentras errores por argumentos no coincidentes. Los argumentos no coincidentes se producen cuando proporcionas menos o más argumentos de los que una función necesita para hacer su trabajo. Por ejemplo, esto es lo que ocurre si intentamos llamar a describe\_pet() sin argumentos:

def describe\_pet(animal\_type, pet\_name):  
 """Display information about a pet."""  
 print(f"\nI have a {animal\_type}.")  
 print(f"My {animal\_type}'s name is {pet\_name.title()}.")  
  
describe\_pet()

Python reconoce que falta alguna información en la llamada a la función, y el rastreo nos lo indica:

Traceback (most recent call last):  
❶ File "pets.py", line 6, in <module>  
❷ describe\_pet()  
 ^^^^^^^^^^^^^^  
❸ TypeError: describe\_pet() missing 2 required positional arguments:   
 'animal\_type' and 'pet\_name'

En primer lugar, el rastreo nos indica la ubicación del problema ❶, lo que nos permite mirar atrás y ver que algo salió mal en nuestra llamada a la función. A continuación, se escribe la llamada a la función infractora para que la veamos ❷. Por último, el rastreo nos dice que a la llamada le faltan dos argumentos e informa de los nombres de los argumentos que faltan ❸. Si esta función estuviera en un archivo aparte, probablemente podríamos reescribir la llamada correctamente sin tener que abrir ese archivo y leer el código de la función.

Python es útil porque lee el código de la función por nosotros y nos dice los nombres de los argumentos que tenemos que proporcionar. Esta es otra motivación para dar nombres descriptivos a tus variables y funciones. Si lo haces, los mensajes de error de Python serán más útiles para ti y para cualquiera que pueda utilizar tu código.

Si proporcionas demasiados argumentos, deberías obtener un rastreo similar que puede ayudarte a hacer coincidir correctamente tu llamada a la función con la definición de la función.

## Pruébalo tú mismo

8-3. Camiseta: Escribe una función llamada make\_shirt() que acepte una talla y el texto de un mensaje que debe imprimirse en la camiseta. La función debe imprimir una frase que resuma la talla de la camiseta y el mensaje impreso en ella.

Llama a la función una vez utilizando argumentos posicionales para hacer una camisa. Llama a la función una segunda vez utilizando argumentos de palabra clave.

8-4. Camisas grandes: Modifica la función make\_shirt() para que las camisas sean grandes por defecto con un mensaje que diga *I love Python*. Haz una camisa grande y una mediana con el mensaje por defecto, y una camisa de cualquier talla con un mensaje diferente.

8-5. Ciudades: Escribe una función llamada describe\_city() que acepte el nombre de una ciudad y su país. La función debe imprimir una frase sencilla, como Reykjavik is in Iceland. Dale al parámetro del país un valor por defecto. Llama a tu función para tres ciudades diferentes, de las que al menos una no esté en el país por defecto.

## Valores de retorno

Una función no siempre tiene que mostrar su resultado directamente. En su lugar, puede procesar algunos datos y luego devolver un valor o conjunto de valores. El valor que devuelve la función se llama *return value*. La sentencia return toma un valor del interior de una función y lo devuelve a la línea que llamó a la función. Los valores de retorno te permiten trasladar gran parte del trabajo pesado de tu programa a las funciones, lo que puede simplificar el cuerpo de tu programa.

### Devolver un valor simple

Veamos una función que toma un nombre y un apellido y devuelve un nombre completo con un formato sencillo:

**formatted\_name.py**

def get\_formatted\_name(first\_name, last\_name):  
 """Return a full name, neatly formatted."""  
❶ full\_name = f"{first\_name} {last\_name}"  
❷ return full\_name.title()  
  
❸ musician = get\_formatted\_name('jimi', 'hendrix')  
print(musician)

La definición de get\_formatted\_name() toma como parámetros un nombre y un apellido. La función combina estos dos nombres, añade un espacio entre ellos y asigna el resultado a full\_name ❶. El valor de full\_name se convierte a mayúsculas y minúsculas, y luego se devuelve a la línea de llamada ❷.

Cuando llamas a una función que devuelve un valor, tienes que proporcionar una variable a la que se pueda asignar el valor devuelto. En este caso, el valor devuelto se asigna a la variable musician ❸. La salida muestra un nombre bien formateado compuesto por las partes del nombre de una persona:

Jimi Hendrix

Esto puede parecer mucho trabajo para obtener un nombre bien formateado cuando podríamos haber escrito simplemente:

print("Jimi Hendrix")

Sin embargo, cuando te planteas trabajar con un programa grande que necesita almacenar muchos nombres y apellidos por separado, funciones como get\_formatted\_name() resultan muy útiles. Almacenas los nombres y apellidos por separado y luego llamas a esta función siempre que quieras mostrar un nombre completo.

### Convertir un argumento en opcional

A veces tiene sentido hacer que un argumento sea opcional, para que las personas que utilicen la función puedan elegir proporcionar información extra sólo si lo desean. Puedes utilizar valores por defecto para hacer que un argumento sea opcional.

Por ejemplo, supongamos que queremos ampliar get\_formatted\_name() para que también incluya los segundos nombres. Un primer intento de incluir los segundos nombres podría tener este aspecto:

def get\_formatted\_name(first\_name, middle\_name, last\_name):  
 """Return a full name, neatly formatted."""  
 full\_name = f"{first\_name} {middle\_name} {last\_name}"  
 return full\_name.title()  
  
musician = get\_formatted\_name('john', 'lee', 'hooker')  
print(musician)

Esta función funciona cuando se le da un nombre, un segundo nombre y un apellido. La función toma las tres partes del nombre y construye una cadena a partir de ellas. La función añade espacios donde corresponda y convierte el nombre completo a mayúsculas y minúsculas:

John Lee Hooker

Pero los segundos nombres no siempre son necesarios, y esta función, tal como está escrita, no funcionaría si intentaras llamarla sólo con un nombre y un apellido. Para que el segundo nombre sea opcional, podemos dar al argumento middle\_name un valor por defecto vacío e ignorar el argumento a menos que el usuario proporcione un valor. Para que get\_formatted\_name() funcione sin segundo nombre, establecemos el valor por defecto de middle\_name en una cadena vacía y lo movemos al final de la lista de parámetros:

def get\_formatted\_name(first\_name, last\_name, middle\_name=''):  
 """Return a full name, neatly formatted."""  
❶ if middle\_name:  
 full\_name = f"{first\_name} {middle\_name} {last\_name}"  
❷ else:  
 full\_name = f"{first\_name} {last\_name}"  
 return full\_name.title()  
  
musician = get\_formatted\_name('jimi', 'hendrix')  
print(musician)  
  
❸ musician = get\_formatted\_name('john', 'hooker', 'lee')  
print(musician)

En este ejemplo, el nombre se construye a partir de tres partes posibles. Como siempre hay un nombre y un apellido, estos parámetros aparecen primero en la definición de la función. El segundo nombre es opcional, por lo que aparece el último en la definición, y su valor por defecto es una cadena vacía.

En el cuerpo de la función, comprobamos si se ha proporcionado un segundo nombre. Python interpreta las cadenas no vacías como True, por lo que la prueba condicional if middle\_name se evalúa como True si hay un argumento de segundo nombre en la llamada a la función ❶. Si se proporciona un segundo nombre, el nombre, el segundo nombre y los apellidos se combinan para formar un nombre completo. Este nombre se cambia a mayúsculas y se devuelve a la línea de llamada a la función, donde se asigna a la variable musician y se imprime. Si no se proporciona el segundo nombre, la cadena vacía falla la prueba if y el bloque else se ejecuta ❷. El nombre completo se hace sólo con el nombre y los apellidos, y el nombre formateado se devuelve a la línea de llamada, donde se asigna a musician y se imprime.

Llamar a esta función con un nombre y apellidos es sencillo. Sin embargo, si utilizamos un segundo nombre, tenemos que asegurarnos de que el segundo nombre es el último argumento pasado para que Python haga coincidir correctamente los argumentos posicionales ❸.

Esta versión modificada de nuestra función funciona para las personas que sólo tienen nombre y apellidos, y también para las que tienen segundo nombre:

Jimi Hendrix  
John Lee Hooker

Los valores opcionales permiten a las funciones manejar una amplia gama de casos de uso, al tiempo que permiten que las llamadas a la función sigan siendo lo más sencillas posible.

### Devolver un diccionario

Una función puede devolver cualquier tipo de valor que necesites, incluidas estructuras de datos más complicadas como listas y diccionarios. Por ejemplo, la siguiente función toma partes de un nombre y devuelve un diccionario que representa a una persona:

**person.py**

def build\_person(first\_name, last\_name):  
 """Return a dictionary of information about a person."""  
❶ person = {'first': first\_name, 'last': last\_name}  
❷ return person  
  
musician = build\_person('jimi', 'hendrix')  
❸ print(musician)

La función build\_person() toma un nombre y un apellido, y pone estos valores en un diccionario ❶. El valor de first\_name se almacena con la clave 'first', y el valor de last\_name se almacena con la clave 'last'. A continuación, se devuelve el diccionario completo que representa a la persona ❷. El valor devuelto se imprime ❸ con las dos piezas originales de información textual almacenadas ahora en un diccionario:

{'first': 'jimi', 'last': 'hendrix'}

Esta función toma información textual simple y la coloca en una estructura de datos más significativa que te permite trabajar con la información más allá de simplemente imprimirla. Las cadenas 'jimi' y 'hendrix' se etiquetan ahora como nombre y apellidos. Puedes ampliar fácilmente esta función para que acepte valores opcionales como un segundo nombre, una edad, una ocupación o cualquier otra información que quieras almacenar sobre una persona. Por ejemplo, el siguiente cambio te permite almacenar también la edad de una persona:

def build\_person(first\_name, last\_name, age=None):  
 """Return a dictionary of information about a person."""  
 person = {'first': first\_name, 'last': last\_name}  
 if age:  
 person['age'] = age  
 return person  
  
musician = build\_person('jimi', 'hendrix', age=27)  
print(musician)

Añadimos un nuevo parámetro opcional age a la definición de la función y le asignamos el valor especial None, que se utiliza cuando una variable no tiene asignado ningún valor específico. Puedes pensar en None como un valor marcador de posición. En las pruebas condicionales, None se evalúa como False. Si la llamada a la función incluye un valor para age, ese valor se almacena en el diccionario. Esta función siempre almacena el nombre de una persona, pero también puede modificarse para almacenar cualquier otra información que desees sobre una persona.

### Utilizar una función con un bucle while

Puedes utilizar funciones con todas las estructuras de Python que has aprendido hasta ahora. Por ejemplo, vamos a utilizar la función get\_formatted\_name() con un bucle while para saludar más formalmente a los usuarios. Aquí tienes un primer intento de saludar a la gente utilizando su nombre y apellidos:

**greeter.py**

def get\_formatted\_name(first\_name, last\_name):  
 """Return a full name, neatly formatted."""  
 full\_name = f"{first\_name} {last\_name}"  
 return full\_name.title()  
  
# This is an infinite loop!  
while True:  
❶ print("\nPlease tell me your name:")  
 f\_name = input("First name: ")  
 l\_name = input("Last name: ")  
  
 formatted\_name = get\_formatted\_name(f\_name, l\_name)  
 print(f"\nHello, {formatted\_name}!")

Para este ejemplo, utilizamos una versión sencilla de get\_formatted\_name() que no incluye los segundos nombres. El bucle while pide al usuario que introduzca su nombre, y le preguntamos su nombre y apellidos por separado ❶.

Pero hay un problema con este bucle while: No hemos definido una condición de abandono. ¿Dónde se pone una condición de abandono cuando se pide una serie de entradas ? Queremos que el usuario pueda salir lo más fácilmente posible, así que cada pregunta debe ofrecer una forma de salir. La sentencia break ofrece una forma directa de salir del bucle en cualquiera de los dos indicadores:

def get\_formatted\_name(first\_name, last\_name):  
 """Return a full name, neatly formatted."""  
 full\_name = f"{first\_name} {last\_name}"  
 return full\_name.title()  
  
while True:  
 print("\nPlease tell me your name:")  
 print("(enter 'q' at any time to quit)")  
  
 f\_name = input("First name: ")  
 if f\_name == 'q':  
 break  
  
 l\_name = input("Last name: ")  
 if l\_name == 'q':  
 break  
  
 formatted\_name = get\_formatted\_name(f\_name, l\_name)  
 print(f"\nHello, {formatted\_name}!")

Añadimos un mensaje que informa al usuario de cómo salir y, a continuación, salimos del bucle si el usuario introduce el valor quit en cualquiera de los dos indicadores. Ahora el programa seguirá saludando a la gente hasta que alguien introduzca q para cualquiera de los dos nombres:

Please tell me your name:  
(enter 'q' at any time to quit)  
First name: eric  
Last name: matthes  
  
Hello, Eric Matthes!  
  
Please tell me your name:  
(enter 'q' at any time to quit)  
First name: q

## Pruébalo tú mismo

8-6. Nombres de ciudades: Escribe una función llamada city\_country() que reciba el nombre de una ciudad y su país. La función debe devolver una cadena con el siguiente formato:

"Santiago, Chile"

Llama a tu función con al menos tres pares ciudad-país e imprime los valores devueltos.

8-7. Álbum: Escribe una función llamada make\_album() que construya un diccionario que describa un álbum de música. La función debe recibir un nombre de artista y un título de álbum, y debe devolver un diccionario que contenga estas dos informaciones. Utiliza la función para crear tres diccionarios que representen distintos álbumes. Imprime cada valor devuelto para mostrar que los diccionarios almacenan correctamente la información del álbum.

Utiliza None para añadir un parámetro opcional a make\_album() que te permita almacenar el número de canciones de un álbum. Si la línea de llamada incluye un valor para el número de canciones, añade ese valor al diccionario del álbum. Haz al menos una nueva llamada a una función que incluya el número de canciones de un álbum.

8-8. Álbumes de Usuario: Empieza con tu programa del Ejercicio 8-7. Escribe un bucle while que permita a los usuarios introducir el artista y el título de un álbum. Una vez que tengas esa información, llama a make\_album() con la entrada del usuario e imprime el diccionario que se crea. Asegúrate de incluir un valor de salida en el bucle while.

## Pasar una lista

A menudo te resultará útil pasar una lista a una función, ya sea una lista de nombres, números u objetos más complejos, como diccionarios. Cuando pasas una lista a una función, ésta obtiene acceso directo al contenido de la lista. Utilicemos las funciones para que trabajar con listas sea más eficaz.

Supongamos que tenemos una lista de usuarios y queremos imprimir un saludo a cada uno. El siguiente ejemplo envía una lista de nombres a una función llamada greet\_users(), que saluda individualmente a cada persona de la lista:

**greet\_users.py**

def greet\_users(names):  
 """Print a simple greeting to each user in the list."""  
 for name in names:  
 msg = f"Hello, {name.title()}!"  
 print(msg)  
  
usernames = ['hannah', 'ty', 'margot']  
greet\_users(usernames)

Definimos greet\_users() para que espere una lista de nombres, que asigna al parámetro names. La función recorre la lista que recibe e imprime un saludo a cada usuario. Fuera de la función, definimos una lista de usuarios y pasamos la lista usernames a greet\_users() en la llamada a la función:

Hello, Hannah!  
Hello, Ty!  
Hello, Margot!

Éste es el resultado que queríamos. Cada usuario ve un saludo personalizado, y puedes llamar a la función siempre que quieras saludar a un conjunto específico de usuarios.

### Modificar una lista en una función

Cuando pasas una lista a una función, la función puede modificar la lista. Cualquier cambio realizado en la lista dentro del cuerpo de la función es permanente, lo que te permite trabajar con eficacia incluso cuando manejas grandes cantidades de datos.

Piensa en una empresa que crea modelos impresos en 3D de diseños que envían los usuarios. Los diseños que necesitan ser impresos se almacenan en una lista, y después de ser impresos se mueven a otra lista. El siguiente código hace esto sin utilizar funciones:

**printing\_models.py**

# Start with some designs that need to be printed.  
unprinted\_designs = ['phone case', 'robot pendant', 'dodecahedron']  
completed\_models = []  
  
# Simulate printing each design, until none are left.  
# Move each design to completed\_models after printing.  
while unprinted\_designs:  
 current\_design = unprinted\_designs.pop()  
 print(f"Printing model: {current\_design}")  
 completed\_models.append(current\_design)  
  
# Display all completed models.  
print("\nThe following models have been printed:")  
for completed\_model in completed\_models:  
 print(completed\_model)

Este programa comienza con una lista de diseños que necesitan ser impresos y una lista vacía llamada completed\_models a la que se moverá cada diseño después de ser impreso. Mientras los diseños permanezcan en unprinted\_designs, el bucle while simula la impresión de cada diseño eliminando un diseño del final de la lista, almacenándolo en current\_design, y mostrando un mensaje de que se está imprimiendo el diseño actual. A continuación, añade el diseño a la lista de modelos completados. Cuando el bucle termina de ejecutarse, se muestra una lista de los diseños que se han impreso:

Printing model: dodecahedron  
Printing model: robot pendant  
Printing model: phone case  
  
The following models have been printed:  
dodecahedron  
robot pendant  
phone case

Podemos reorganizar este código escribiendo dos funciones, cada una de las cuales realiza un trabajo específico. La mayor parte del código no cambiará; sólo lo estamos estructurando con más cuidado. La primera función se encargará de imprimir los diseños, y la segunda resumirá las impresiones realizadas:

❶ def print\_models(unprinted\_designs, completed\_models):  
 """  
 Simulate printing each design, until none are left.  
 Move each design to completed\_models after printing.  
 """  
 while unprinted\_designs:  
 current\_design = unprinted\_designs.pop()  
 print(f"Printing model: {current\_design}")  
 completed\_models.append(current\_design)  
  
❷ def show\_completed\_models(completed\_models):  
 """Show all the models that were printed."""  
 print("\nThe following models have been printed:")  
 for completed\_model in completed\_models:  
 print(completed\_model)  
  
unprinted\_designs = ['phone case', 'robot pendant', 'dodecahedron']  
completed\_models = []  
  
print\_models(unprinted\_designs, completed\_models)  
show\_completed\_models(completed\_models)

Definimos la función print\_models() con dos parámetros: una lista de diseños que hay que imprimir y una lista de modelos completados ❶. Dadas estas dos listas, la función simula la impresión de cada diseño vaciando la lista de diseños sin imprimir y rellenando la lista de modelos completados. A continuación, definimos la función show\_completed\_models() con un parámetro: la lista de modelos completados ❷. Dada esta lista, show\_completed\_models() muestra el nombre de cada modelo que se imprimió.

Este programa tiene la misma salida que la versión sin funciones, pero el código está mucho más organizado. El código que hace la mayor parte del trabajo se ha trasladado a dos funciones separadas, lo que hace que la parte principal del programa sea más fácil de entender. Mira el cuerpo del programa y fíjate en lo fácil que es seguir lo que ocurre:

unprinted\_designs = ['phone case', 'robot pendant', 'dodecahedron']  
completed\_models = []  
  
print\_models(unprinted\_designs, completed\_models)  
show\_completed\_models(completed\_models)

Creamos una lista de diseños sin imprimir y una lista vacía que contendrá los modelos terminados. Luego, como ya hemos definido nuestras dos funciones, todo lo que tenemos que hacer es llamarlas y pasarles los argumentos adecuados. Llamamos a print\_models() y le pasamos las dos listas que necesita; como era de esperar, print\_models() simula la impresión de los diseños. Luego llamamos a show\_completed\_models() y le pasamos la lista de modelos completados para que nos informe de los modelos que se han impreso. Los nombres descriptivos de las funciones permiten que otros lean este código y lo entiendan, incluso sin comentarios.

Este programa es más fácil de ampliar y mantener que la versión sin funciones. Si más adelante necesitamos imprimir más diseños, basta con volver a llamar a print\_models(). Si nos damos cuenta de que hay que modificar el código de impresión, podemos cambiar el código una vez, y nuestros cambios tendrán lugar en todos los lugares donde se llame a la función. Esta técnica es más eficaz que tener que actualizar el código por separado en varios lugares del programa.

Este ejemplo también demuestra la idea de que cada función debe tener un trabajo específico. La primera función imprime cada diseño, y la segunda muestra los modelos completados. Esto es más beneficioso que utilizar una función para hacer ambos trabajos. Si estás escribiendo una función y notas que realiza demasiadas tareas diferentes, intenta dividir el código en dos funciones. Recuerda que siempre puedes llamar a una función desde otra función, lo que puede ser útil al dividir una tarea compleja en una serie de pasos.

### Evitar que una función modifique una lista

A veces querrás evitar que una función modifique una lista. Por ejemplo, supongamos que empiezas con una lista de diseños sin imprimir y escribes una función para moverlos a una lista de modelos terminados, como en el ejemplo anterior. Puede que decidas que, aunque hayas impreso todos los diseños, quieres conservar la lista original de diseños sin imprimir para tus archivos. Pero como has movido todos los nombres de los diseños fuera de unprinted\_designs, la lista está ahora vacía, y la lista vacía es la única versión que tienes; la original ha desaparecido. En este caso, puedes solucionar este problema pasando a la función una copia de la lista, no el original. Cualquier cambio que la función haga en la lista afectará sólo a la copia, dejando intacta la lista original.

Puedes enviar una copia de una lista a una función de esta forma:

function\_name(list\_name[:])

La notación de corte [:] hace una copia de la lista para enviarla a la función. Si no quisiéramos vaciar la lista de diseños no impresos en *printing\_models.py*, podríamos llamar a print\_models() de este modo:

print\_models(unprinted\_designs[:], completed\_models)

La función print\_models() puede hacer su trabajo porque sigue recibiendo los nombres de todos los diseños no impresos. Pero esta vez utiliza una copia de la lista original de diseños sin imprimir, no la lista real unprinted\_designs. La lista completed\_models se llenará con los nombres de los modelos impresos como antes, pero la lista original de diseños no impresos no se verá afectada por la función.

Aunque puedes conservar el contenido de una lista pasando una copia de ella a tus funciones, debes pasar la lista original a las funciones a menos que tengas una razón específica para pasar una copia. Es más eficaz que una función trabaje con una lista existente, porque así se evita emplear el tiempo y la memoria necesarios para hacer una copia por separado. Esto es especialmente cierto cuando se trabaja con listas grandes.

## Pruébalo tú mismo

8-9. Mensajes: Haz una lista que contenga una serie de mensajes cortos de texto. Pasa la lista a una función llamada show\_messages(), que imprime cada mensaje de texto.

8-10. Enviar mensajes: Empieza con una copia de tu programa del Ejercicio 8-9. Escribe una función llamada send\_messages() que imprima cada mensaje de texto y mueva cada mensaje a una nueva lista llamada sent\_messages a medida que se imprime. Después de llamar a la función, imprime ambas listas para asegurarte de que los mensajes se han movido correctamente.

8-11. Mensajes Archivados: Empieza con tu trabajo del Ejercicio 8-10. Llama a la función send\_messages() con una copia de la lista de mensajes. Después de llamar a la función, imprime tus dos listas para mostrar que la lista original ha conservado sus mensajes.

## Pasar un número arbitrario de argumentos

A veces no sabrás de antemano cuántos argumentos debe aceptar una función. Afortunadamente, Python permite que una función recoja un número arbitrario de argumentos de la sentencia que la llama.

Por ejemplo, considera una función que construye una pizza. Necesita aceptar un número de ingredientes, pero no puedes saber de antemano cuántos ingredientes querrá una persona. La función del siguiente ejemplo tiene un parámetro, \*toppings, pero este parámetro recoge tantos argumentos como proporcione la línea de llamada:

**pizza.py**

def make\_pizza(\*toppings):  
 """Print the list of toppings that have been requested."""  
 print(toppings)  
  
make\_pizza('pepperoni')  
make\_pizza('mushrooms', 'green peppers', 'extra cheese')

El asterisco en el nombre del parámetro \*toppings indica a Python que cree una tupla llamada toppings, que contenga todos los valores que recibe esta función. La llamada print() en el cuerpo de la función produce una salida que muestra que Python puede manejar una llamada a la función con un valor y una llamada con tres valores. Trata las distintas llamadas de forma similar. Observa que Python empaqueta los argumentos en una tupla, aunque la función sólo reciba un valor:

('pepperoni',)  
('mushrooms', 'green peppers', 'extra cheese')

Ahora podemos sustituir la llamada a print() por un bucle que recorra la lista de ingredientes y describa la pizza que se pide:

def make\_pizza(\*toppings):  
 """Summarize the pizza we are about to make."""  
 print("\nMaking a pizza with the following toppings:")  
 for topping in toppings:  
 print(f"- {topping}")  
  
make\_pizza('pepperoni')  
make\_pizza('mushrooms', 'green peppers', 'extra cheese')

La función responde adecuadamente, tanto si recibe un valor como tres:

Making a pizza with the following toppings:  
- pepperoni  
  
Making a pizza with the following toppings:  
- mushrooms  
- green peppers  
- extra cheese

Esta sintaxis funciona independientemente del número de argumentos que reciba la función.

### Mezclar argumentos posicionales y arbitrarios

Si quieres que una función acepte varios tipos diferentes de argumentos, el parámetro que acepta un número arbitrario de argumentos debe colocarse en último lugar en la definición de la función. Python combina primero los argumentos posicionales y los de palabra clave, y luego reúne los argumentos restantes en el último parámetro.

Por ejemplo, si la función necesita recoger un tamaño para la pizza, ese parámetro debe ir antes que el parámetro \*toppings:

def make\_pizza(size, \*toppings):  
 """Summarize the pizza we are about to make."""  
 print(f"\nMaking a {size}-inch pizza with the following toppings:")  
 for topping in toppings:  
 print(f"- {topping}")  
  
make\_pizza(16, 'pepperoni')  
make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

En la definición de la función, Python asigna el primer valor que recibe al parámetro size. Todos los demás valores que vengan después se almacenan en la tupla toppings. Las llamadas a la función incluyen primero un argumento para el tamaño, seguido de tantos ingredientes como sean necesarios.

Ahora cada pizza tiene un tamaño y un número de ingredientes, y cada información se imprime en el lugar adecuado, mostrando primero el tamaño y después los ingredientes:

Making a 16-inch pizza with the following toppings:  
- pepperoni  
  
Making a 12-inch pizza with the following toppings:  
- mushrooms  
- green peppers  
- extra cheese

## Nota

A menudo verás el nombre genérico de parámetro \*args, que recoge argumentos posicionales arbitrarios como éste.

### Uso de argumentos arbitrarios de palabras clave

A veces querrás aceptar un número arbitrario de argumentos, pero no sabrás de antemano qué tipo de información se pasará a la función. En este caso, puedes escribir funciones que acepten tantos pares clave-valor como proporcione la sentencia de llamada. Un ejemplo tiene que ver con la creación de perfiles de usuario: sabes que obtendrás información sobre un usuario, pero no estás seguro de qué tipo de información recibirás. La función build\_profile() del siguiente ejemplo siempre recibe un nombre y un apellido, pero también acepta un número arbitrario de argumentos de palabra clave:

**user\_profile.py**

def build\_profile(first, last, \*\*user\_info):  
 """Build a dictionary containing everything we know about a user."""  
❶ user\_info['first\_name'] = first  
 user\_info['last\_name'] = last  
 return user\_info  
  
user\_profile = build\_profile('albert', 'einstein',  
 location='princeton',  
 field='physics')  
print(user\_profile)

La definición de build\_profile() espera un nombre y un apellido, y luego permite al usuario pasar tantos pares nombre-valor como desee. Los asteriscos dobles delante del parámetro \*\*user\_info hacen que Python cree un diccionario llamado user\_info que contiene todos los pares nombre-valor adicionales que recibe la función. Dentro de la función, puedes acceder a los pares clave-valor de user\_info como lo harías con cualquier diccionario.

En el cuerpo de build\_profile(), añadimos el nombre y los apellidos al diccionario user\_info porque siempre recibiremos estos dos datos del usuario ❶, y aún no se han colocado en el diccionario. Después devolvemos el diccionario user\_info a la línea de llamada a la función.

Llamamos a build\_profile(), pasándole el nombre 'albert', el apellido 'einstein', y los dos pares clave-valor location='princeton' y field='physics'. Asignamos el profile devuelto a user\_profile e imprimimos user\_profile:

{'location': 'princeton', 'field': 'physics',  
'first\_name': 'albert', 'last\_name': 'einstein'}

El diccionario devuelto contiene el nombre y apellidos del usuario y, en este caso, también la localidad y la especialidad. La función funcionará independientemente de cuántos pares clave-valor adicionales se proporcionen en la llamada a la función.

Puedes mezclar valores posicionales, de palabra clave y arbitrarios de muchas formas distintas al escribir tus propias funciones. Es útil saber que existen todos estos tipos de argumentos porque los verás a menudo cuando empieces a leer el código de otras personas. Utilizar correctamente los distintos tipos y saber cuándo utilizar cada uno de ellos requiere práctica. Por ahora, recuerda utilizar el método más sencillo que haga el trabajo. A medida que avances, aprenderás a utilizar cada vez el enfoque más eficaz.

## Nota

A menudo verás que el nombre de parámetro \*\*kwargs se utiliza para recoger argumentos no específicos de palabras clave.

## Pruébalo tú mismo

8-12. Bocadillos: Escribe una función que acepte una lista de elementos que una persona quiere en un bocadillo. La función debe tener un parámetro que recoja tantos elementos como proporcione la llamada a la función, y debe imprimir un resumen del bocadillo que se está pidiendo. Llama a la función tres veces, utilizando un número distinto de argumentos cada vez.

8-13. Perfil de usuario: Empieza con una copia de *user\_profile.py* de la página 148. Construye un perfil tuyo llamando a build\_profile(), utilizando tu nombre y apellidos y otros tres pares clave-valor que te describan.

8-14. Coches: Escribe una función que almacene información sobre un coche en un diccionario. La función debe recibir siempre un fabricante y un nombre de modelo. A continuación, debe aceptar un número arbitrario de argumentos de palabra clave. Llama a la función con la información requerida y otros dos pares nombre-valor, como un color o una característica opcional. Tu función debería funcionar para una llamada como ésta:

car = make\_car('subaru', 'outback', color='blue', tow\_package=True)

Imprime el diccionario devuelto para asegurarte de que toda la información se ha almacenado correctamente.

## Almacenar tus funciones en módulos

Una ventaja de las funciones es la forma en que separan bloques de código de tu programa principal. Cuando utilizas nombres descriptivos para tus funciones, tus programas son mucho más fáciles de seguir. Puedes ir un paso más allá almacenando tus funciones en un archivo separado llamado *module* y luego *importing* ese módulo en tu programa principal. Una sentencia import indica a Python que haga que el código de un módulo esté disponible en el archivo del programa que se esté ejecutando en ese momento.

Almacenar tus funciones en un archivo separado te permite ocultar los detalles del código de tu programa y centrarte en su lógica de alto nivel. También te permite reutilizar funciones en muchos programas diferentes. Cuando almacenas tus funciones en archivos separados, puedes compartir esos archivos con otros programadores sin que tenga que compartir todo tu programa. Saber importar funciones también te permite utilizar bibliotecas de funciones que han escrito otros programadores.

Hay varias formas de importar un módulo, y te mostraré brevemente cada una de ellas.

### Importar un módulo entero

Para empezar a importar funciones, primero tenemos que crear un módulo. Un *module* es un archivo que termina en *.py* y que contiene el código que quieres importar a tu programa. Vamos a crear un módulo que contenga la función make\_pizza(). Para crear este módulo, eliminaremos todo del archivo *pizza.py* excepto la función make\_pizza():

**pizza.py**

def make\_pizza(size, \*toppings):  
 """Summarize the pizza we are about to make."""  
 print(f"\nMaking a {size}-inch pizza with the following toppings:")  
 for topping in toppings:  
 print(f"- {topping}")

Ahora crearemos un archivo independiente llamado *making\_pizzas.py* en el mismo directorio que *pizza.py*. Este archivo importa el módulo que acabamos de crear y luego hace dos llamadas a make\_pizza():

**making\_pizzas.py**

import pizza  
  
❶ pizza.make\_pizza(16, 'pepperoni')  
pizza.make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

Cuando Python lee este archivo, la línea import pizza le dice a Python que abra el archivo *pizza.py* y copie todas las funciones del mismo en este programa. En realidad, no ves que se copie código entre archivos, porque Python copia el código entre bastidores, justo antes de que se ejecute el programa. Todo lo que necesitas saber es que cualquier función definida en *pizza.py* estará ahora disponible en *making\_pizzas.py*.

Para llamar a una función desde un módulo importado, introduce el nombre del módulo que has importado, pizza, seguido del nombre de la función, make\_pizza(), separados por un punto ❶. Este código produce la misma salida que el programa original que no importó ningún módulo:

Making a 16-inch pizza with the following toppings:  
- pepperoni  
  
Making a 12-inch pizza with the following toppings:  
- mushrooms  
- green peppers  
- extra cheese

Este primer enfoque de la importación, en el que simplemente escribes import seguido del nombre del módulo, hace que todas las funciones del módulo estén disponibles en tu programa. Si utilizas este tipo de sentencia import para importar un módulo entero llamado *module\_name.py*, cada función del módulo estará disponible mediante la siguiente sintaxis:

module\_name.function\_name()

### Importar funciones específicas

También puedes importar una función específica de un módulo. Esta es la sintaxis general para este enfoque:

from module\_name import function\_name

Puedes importar tantas funciones como quieras de un módulo separando el nombre de cada función con una coma:

from module\_name import function\_0, function\_1, function\_2

El ejemplo de *making\_pizzas.py* tendría este aspecto si quisiéramos importar sólo la función que vamos a utilizar:

from pizza import make\_pizza  
  
make\_pizza(16, 'pepperoni')  
make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

Con esta sintaxis, no necesitas utilizar la notación de punto cuando llames a una función. Como hemos importado explícitamente la función make\_pizza() en la sentencia import, podemos llamarla por su nombre cuando utilicemos la función.

### Utilizar as para dar un alias a una función

Si el nombre de una función que estás importando puede entrar en conflicto con un nombre existente en tu programa, o si el nombre de la función es largo, puedes utilizar un *alias*corto y único -un nombre alternativo similar a un apodo para la función. Le darás a la función este apodo especial cuando importes la función.

Aquí damos a la función make\_pizza() un alias, mp(), importando make\_pizza as mp. La palabra clave as cambia el nombre de una función utilizando el alias que le proporciones:

from pizza import make\_pizza as mp  
  
mp(16, 'pepperoni')  
mp(12, 'mushrooms', 'green peppers', 'extra cheese')

La sentencia import mostrada aquí cambia el nombre de la función make\_pizza() a mp() en este programa. Cada vez que queramos llamar a make\_pizza() podemos simplemente escribir mp() en su lugar, y Python ejecutará el código en make\_pizza() evitando cualquier confusión con otra función make\_pizza() que puedas haber escrito en este archivo de programa.

La sintaxis general para proporcionar un alias es:

from module\_name import function\_name as fn

### Usar as para dar un alias a un módulo

También puedes proporcionar un alias para el nombre de un módulo. Dar a un módulo un alias corto, como p para pizza, te permite llamar a las funciones del módulo más rápidamente. Llamar a p.make\_pizza() es más conciso que llamar a pizza.make\_pizza():

import pizza as p  
  
p.make\_pizza(16, 'pepperoni')  
p.make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

El módulo pizza recibe el alias p en la sentencia import, pero todas las funciones del módulo conservan sus nombres originales. Llamar a las funciones escribiendo p.make\_pizza() no sólo es más conciso que pizza.make\_pizza(), sino que también desvía tu atención del nombre del módulo y te permite centrarte en los nombres descriptivos de sus funciones. Estos nombres de función, que te indican claramente lo que hace cada función, son más importantes para la legibilidad de tu código que utilizar el nombre completo del módulo.

La sintaxis general de este enfoque es

import module\_name as mn

### Importar todas las funciones de un módulo

Puedes indicar a Python que importe todas las funciones de un módulo utilizando el operador asterisco (\*):

from pizza import \*  
  
make\_pizza(16, 'pepperoni')  
make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

El asterisco en la sentencia import indica a Python que copie todas las funciones del módulo pizza en este archivo de programa. Como se importan todas las funciones, puedes llamar a cada una de ellas por su nombre sin utilizar la notación de punto. Sin embargo, es mejor no utilizar este método cuando trabajes con módulos más grandes que no hayas escrito tú: si el módulo tiene un nombre de función que coincide con un nombre existente en tu proyecto, puedes obtener resultados inesperados. Python puede ver varias funciones o variables con el mismo nombre, y en lugar de importar todas las funciones por separado, las sobrescribirá.

Lo mejor es importar la función o funciones que quieras, o importar todo el módulo y utilizar la notación por puntos. Esto conduce a un código claro que es fácil de leer y comprender. Incluyo esta sección para que reconozcas declaraciones import como las siguientes cuando las veas en el código de otras personas:

from module\_name import \*

## Funciones de estilo

Debes tener en cuenta algunos detalles a la hora de dar estilo a las funciones. Las funciones deben tener nombres descriptivos, y estos nombres deben utilizar letras minúsculas y guiones bajos. Los nombres descriptivos te ayudan a ti y a los demás a entender lo que intenta hacer tu código. Los nombres de los módulos también deben utilizar estas convenciones.

Cada función debe tener un comentario que explique de forma concisa lo que hace la función. Este comentario debe aparecer inmediatamente después de la definición de la función y utilizar el formato docstring. En una función bien documentada, otros programadores pueden utilizar la función leyendo únicamente la descripción del docstring. Deberían poder confiar en que el código funciona como se describe, y siempre que conozcan el nombre de la función, los argumentos que necesita y el tipo de valor que devuelve, deberían poder utilizarla en sus programas.

Si especificas un valor por defecto para un parámetro, no deben utilizarse espacios a ambos lados del signo igual:

def function\_name(parameter\_0, parameter\_1='default value')

La misma convención debe utilizarse para los argumentos de palabras clave en las llamadas a funciones:

function\_name(value\_0, parameter\_1='value')

PEP 8[(https://www.python.org/dev/peps/pep-0008)](https://www.python.org/dev/peps/pep-0008) recomienda limitar las líneas de código a 79 caracteres para que cada línea sea visible en una ventana del editor de tamaño razonable. Si un conjunto de parámetros hace que la definición de una función tenga más de 79 caracteres, pulsa INTRO después del paréntesis de apertura en la línea de definición. En la línea siguiente, pulsa dos veces la tecla TAB para separar la lista de argumentos del cuerpo de la función, que sólo se sangrará un nivel.

La mayoría de los editores alinean automáticamente las líneas adicionales de argumentos para que coincidan con la sangría que has establecido en la primera línea:

def function\_name(  
 parameter\_0, parameter\_1, parameter\_2,  
 parameter\_3, parameter\_4, parameter\_5):  
 function body...

Si tu programa o módulo tiene más de una función, puedes separar cada una de ellas con dos líneas en blanco para que sea más fácil ver dónde acaba una función y empieza la siguiente.

Todas las declaraciones import deben escribirse al principio de un archivo. La única excepción es si utilizas comentarios al principio de tu archivo para describir el programa en general.

## Pruébalo tú mismo

8-15. Imprimiendo modelos: Coloca las funciones del ejemplo *printing\_models.py* en un archivo aparte llamado *printing\_functions.py*. Escribe una sentencia import al principio de *printing\_models.py*, y modifica el archivo para utilizar las funciones importadas.

8-16. Importaciones: Utilizando un programa que hayas escrito y que contenga una función, guarda esa función en un archivo aparte. Importa la función a tu archivo de programa principal, y llama a la función utilizando cada uno de estos enfoques:

import module\_name  
from module\_name import function\_name  
from module\_name import function\_name as fn  
import module\_name as mn  
from module\_name import \*

8-17. Estilizar funciones: Elige tres programas cualesquiera que hayas escrito para este capítulo, y asegúrate de que siguen las pautas de estilo descritas en esta sección.

## Resumen

En este capítulo has aprendido a escribir funciones y a pasar argumentos para que tus funciones tengan acceso a la información que necesitan para hacer su trabajo. Aprendiste a utilizar argumentos posicionales y de palabra clave, y también a aceptar un número arbitrario de argumentos. Viste funciones que muestran la salida y funciones que devuelven valores. Aprendiste a utilizar funciones con listas, diccionarios, sentencias if y bucles while. También aprendiste a almacenar tus funciones en archivos separados llamados *modules*, para que los archivos de tu programa sean más sencillos y fáciles de entender. Por último, aprendiste a dar estilo a tus funciones para que tus programas sigan estando bien estructurados y sean lo más fáciles de leer para ti y para los demás.

Uno de tus objetivos como programador debe ser escribir código sencillo que haga lo que tú quieres, y las funciones te ayudan a conseguirlo. Te permiten escribir bloques de código y dejarlos en paz una vez que sabes que funcionan. Cuando sabes que una función hace su trabajo correctamente, puedes confiar en que seguirá funcionando y pasar a tu siguiente tarea de codificación.

Las funciones te permiten escribir código una vez y reutilizarlo tantas veces como quieras. Cuando necesites ejecutar el código de una función, todo lo que tienes que hacer es escribir una llamada de una sola línea y la función hará su trabajo. Cuando necesites modificar el comportamiento de una función, sólo tendrás que modificar un bloque de código, y tu cambio surtirá efecto en todos los lugares en los que hayas hecho una llamada a esa función.

Utilizar funciones facilita la lectura de tus programas, y los buenos nombres de función resumen lo que hace cada parte de un programa. Leer una serie de llamadas a funciones te da una idea mucho más rápida de lo que hace un programa que leer una larga serie de bloques de código.

Las funciones también hacen que tu código sea más fácil de probar y depurar. Cuando la mayor parte del trabajo de tu programa lo realiza un conjunto de funciones, cada una de las cuales tiene un trabajo específico, es mucho más fácil probar y mantener el código que has escrito. Puedes escribir un programa independiente que llame a cada función y compruebe si cada función funciona en todas las situaciones que pueda encontrar. Al hacer esto, puedes estar seguro de que tus funciones funcionarán correctamente cada vez que las llames.

En el Capítulo 9, aprenderás a escribir clases. *Classes* combinan funciones y datos en un paquete ordenado que puede utilizarse de forma flexible y eficiente.