Aprende a programar con Minecraft

### **Definir tus propias funciones**

Veamos cómo puedes utilizar funciones en tu código. En el siguiente ejemplo, hago una función llamada saludo() que simplemente imprime dos líneas:

def greeting():  
print("Hello")  
print("Nice to meet you")

La palabra clave def, que es una abreviatura de *definir*, indica a Python que estás escribiendo una función. Siempre que quieras escribir una función, debes escribir primero def seguido del nombre de la función. En este ejemplo, saludo es el nombre de la función. No olvides añadir los paréntesis y los dos puntos al final de la primera línea. Las líneas que siguen a los dos puntos son el *cuerpo* de la función, que es el código que se ejecutará cuando se llame a la función.

**NOTA**

*Mantén una sangría coherente en tu código. Utiliza siempre cuatro espacios de sangría en el cuerpo de la función.*

Una función puede contener tantas sentencias como quieras. También puede incluir sentencias if, bucles, variables, condiciones, operadores matemáticos, etc. Cuando llegues al final del código de la función, deja de sangrar las líneas para que Python sepa qué sentencias pertenecen a la función y qué sentencias pertenecen a otras partes de tu código.

Puedes crear tantas funciones como quieras en un programa, siempre que tengan nombres diferentes.

#### **Llamar a una función**

Para utilizar, o *llamar*, a una función, escribe el nombre de la función con los argumentos que pueda necesitar entre paréntesis. Si tu función no necesita argumentos, escribe simplemente el nombre de la función y un conjunto de paréntesis vacíos.

Para llamar a la función saludo( ) definida anteriormente, utilizarías el siguiente código:

greeting()

Puedes llamar a la función tantas veces como quieras. Vamos a llamar a la función saludo () tres veces:

greeting()  
greeting()  
greeting()

Cuando ejecutes el programa, debería producir la salida de la función tres veces, así:

Hello  
Nice to meet you  
Hello  
Nice to meet you  
Hello  
Nice to meet you

Debes llamar a la función en el cuerpo de tu código, o la función no hará nada. Esto es un error muy común. Si ejecutas un programa que define algunas funciones y tu código no hace nada, puede deberse a que olvidaste llamar a las funciones que creaste.

También puedes llamar a funciones desde dentro de otra función que hayas creado. Esto incluye tanto las funciones incorporadas en Python como las que tú hayas creado. Verás esto en acción dentro de un momento.

#### **Las funciones reciben argumentos**

Los paréntesis de una función contienen sus argumentos, que son valores que utiliza la función. Los valores se utilizan para variables específicas dentro de la función cuando ésta se ejecuta. No todas las funciones necesitan argumentos. Por ejemplo, la función saludo( ) no necesita argumentos.

Pero supongamos que quiero mostrar un saludo a alguien utilizando su nombre. Escribiré esto como una función para poder reutilizar el código para saludar a distintas personas:

def fancyGreeting(personName):  
print("Hello, " + personName)  
  
fancyGreeting("Mario")  
fancyGreeting("Steve")

En este ejemplo, la función se llama dos veces utilizando argumentos diferentes, "Mario" y "Steve". Cuando ejecutas el programa, la salida tiene este aspecto:

Hello, Mario  
Hello, Steve

Si olvidas incluir un argumento cuando llamas a una función que necesita uno, obtendrás un error. Además, si una función necesita varios argumentos y te olvidas de incluir aunque sólo sea uno de ellos, obtendrás un error. Por ejemplo, intentemos llamar a la función fancyGreeting( ) sin argumentos, así:

fancyGreeting()

Aparecerá el siguiente mensaje de error:

Traceback (most recent call last):  
File "<pyshell#2>", line 1, in <module>  
fancyGreeting()  
➊ TypeError: fancyGreeting() takes exactly 1 argument (0 given)

Se trata de un mensaje de error útil porque la última línea explica lo que está mal en el código ➊. La función fancyGreeting( ) toma un argumento, pero como no se le dio ningún argumento, eso provocó el error.

Puedes crear una función que tome varios argumentos. Por ejemplo, el siguiente programa contiene una función que saluda a alguien, espera unos segundos y luego se despide. La función utiliza un argumento para el nombre de la persona y el número de segundos que esperará el programa:

import time  
  
➊ def helloAndGoodbye(personName, secsToWait):  
print("Hello, " + personName)  
time.sleep(secsToWait)  
print("Goodbye, " + personName)  
  
➋ helloAndGoodbye("Mario", 10)  
helloAndGoodbye("Steve", 23)

Cada argumento se separa con una coma cuando se define la función ➊. Luego, cuando se llama a las funciones, los argumentos se pasan en el mismo orden en que se definieron ➋.

**NOTA**

Es*posible que encuentres los términos* argumento *y* parámetro *utilizados casi indistintamente. Los parámetros de una función definen los tipos de argumentos que acepta o requiere, y los argumentos son los valores que pasas a la función cuando la llamas. Para simplificar,* *en este* libro *sólo utilizaremos el término* argumento *.*

#### **Misión 41: Construir un bosque**

Tu misión es crear un bosque de árboles en Minecraft. Como un bosque no es más que un montón de árboles, crearemos el bosque creando una función que construya un árbol y luego llamaremos a esa función muchas veces para crear un bosque.

El listado[8-1](ch08.xhtml#ch8ex1) es el código básico que utilizarás.

*forest.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
➊ def growTree(x, y, z):  
# Creates a tree at the coordinates given  
# Write your code to make a tree here  
  
pos = mc.player.getTilePos()  
x = pos.x  
y = pos.y  
z = pos.z  
  
➋ growTree(x + 1, y, z)

*Listado 8-1: Estructura de un programa que utiliza funciones para crear un bosque de árboles*

La función growTree() ➊ creada en este código toma argumentos para las coordenadas donde se construirá el árbol. Tu tarea consiste en escribir código en el cuerpo de la función que cree un árbol en las coordenadas dadas. Para ello utilizarás las funciones setBlock() y setBlocks().

Copia [el Listado 8-1](ch08.xhtml#ch8ex1) en un nuevo archivo en IDLE y guárdalo como *bosque.py* en una nueva carpeta llamada *funciones*.

Cuando hayas creado algo que se parezca a un árbol y aparezca en pantalla, prueba a escribir más llamadas a la función utilizando argumentos distintos para que los árboles aparezcan en lugares diferentes. La primera ya la has hecho ➋. Intenta crear al menos nueve árboles delante del jugador cada vez que ejecutes tu programa. [La Figura 8-1](ch08.xhtml#ch8fig1) muestra los árboles que creó mi programa.



*Figura 8-1: Acabo de crear una hermosa hilera de árboles.*

**OBJETIVO EXTRA: BOSQUE ALEATORIO**

Utiliza la función randint() del módulo random para aleatorizar la distancia entre los árboles del bosque.

#### **Refactorizar un programa**

A menudo escribirás un programa que utiliza varias veces el mismo bloque de código. Hacer cambios en el programa se volverá tedioso cuando quieras cambiar el mismo código en distintos lugares. Puede que hayas hecho esto en programas que has escrito en el pasado, pero hay una forma mucho mejor.

Puedes reestructurar tus programas para que utilicen funciones. Para ello, traslada el código que se repite varias veces a una única función que luego podrás utilizar tantas veces como quieras en el resto del código. Como sólo tendrás que hacer cambios en un lugar en lugar de en varios, ahorrarás espacio y el programa será más fácil de mantener. El proceso de reestructurar tu código de este modo se denomina *refactorización*.

Por ejemplo, el siguiente código pregunta su nombre a tres personas y luego imprime un saludo a cada una de ellas:

name1 = input("Hello, what is your name?")  
print("Pleased to meet you, " + name1)  
name2 = input("Hello, what is your name?")  
print("Pleased to meet you, " + name2)  
name3 = input("Hello, what is your name?")  
print("Pleased to meet you, " + name3)

El código repite tres veces las mismas dos afirmaciones. ¿Y si quisieras cambiar la pregunta o el saludo? No hay mucho problema en cambiar el código para 3 personas, pero ¿y si escribieras código para 100 personas?

La alternativa es escribir el código como una función y llamarla tres veces. Aquí tienes el código después de refactorizarlo:

def helloFriend():  
name = input("Hello, what is your name?")  
print("Pleased to meet you, " + name)  
  
helloFriend()  
helloFriend()  
helloFriend()

Ahora, cuando el programa se ejecute, pedirá una entrada y luego emitirá una cadena, y hará ambas tareas tres veces. Aquí tienes la entrada y la salida:

Hello, what is your name? Craig  
Pleased to meet you, Craig  
Hello, what is your name? Still Craig  
Pleased to meet you, Still Craig  
Hello, what is your name? Craig again  
Pleased to meet you, Craig again

La segunda versión del código tiene el mismo resultado que la primera, pero como puedes ver, es mucho más fácil de leer y mucho más fácil de cambiar.

#### **Misión nº 42: Refactoriza**

A veces escribirás un programa para darte cuenta después de que deberías haber utilizado funciones (a mí me pasa a menudo). Refactorizar el código para incluir funciones es una habilidad muy importante.

En esta misión, practicarás la refactorización de un programa para utilizar una función en lugar de repetir varias veces las mismas sentencias.

[El listado 8-2](ch08.xhtml#ch8ex2) coloca un bloque melón debajo del jugador cada 10 segundos. Reescribiremos el código para utilizar una función. Actualmente, el programa coloca tres bloques utilizando la misma línea de código tres veces. [La Figura 8-2](ch08.xhtml#ch8fig2) muestra el resultado del programa.

*melonFunction.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
import time  
  
pos = mc.player.getPos()  
x = pos.x  
y = pos.y  
z = pos.z  
mc.setBlock(x, y - 1, z, 103)  
time.sleep(10)  
  
pos = mc.player.getPos()  
x = pos.x  
y = pos.y - 1  
z = pos.z  
mc.setBlock(x, y, z, 103)  
time.sleep(10)  
  
pos = mc.player.getPos()  
x = pos.x  
y = pos.y - 1  
z = pos.z  
mc.setBlock(x, y, z, 103)  
time.sleep(10)

*Listado 8-2: Código que necesita refactorización*

Este código no es muy bonito, ¿verdad? Se repiten varias líneas, lo que siempre es señal de que el código necesita ser refactorizado con la ayuda de una definición de función.

**CONSEJO**

*Identifica qué partes del código se repiten para hacerte una idea de lo que debe hacer tu función.*



*Figura 8-2: Tres deliciosos melones bajo tierra*

Cambia el código para que coloque seis bloques en total llamando a tu función seis veces. Crea un nuevo archivo y guárdalo como *funciónmelón.py* en la carpeta *funciones*. Copia [el Listado 8-2](ch08.xhtml#ch8ex2) en tu archivo y refactoriza el código para utilizar una función. Llama a la nueva función hacerMelón().

**OBJETIVO EXTRA: BLOQUES BAJO PIE**

Añade argumentos a tu función makeMelon () para controlar el tipo de bloque, el tiempo de reposo o la distancia por debajo del jugador.

#### **Comentar con Docstrings**

Utilizar comentarios en el código Python es una forma de explicar lo que hace el código. Cuando Python ejecuta un programa, ignora todo lo que hay en un comentario, por lo que los comentarios no afectan a cómo se ejecuta el código. El objetivo principal de los comentarios es explicar lo que se supone que debe hacer tu código a otras personas que puedan verlo o utilizarlo. Los comentarios también son recordatorios útiles para ti mismo en el futuro.

Como se supone que las funciones son reutilizables, tiene sentido explicar su finalidad. Para escribir las explicaciones de nuestras funciones, utilizaremos explicaciones largas llamadas *docstrings*. Una docstring es un comentario de varias líneas que colocas al principio de una función para explicar su uso.

La función duplicarPalabra( ) del siguiente ejemplo tiene una docstring que explica su tarea:

def duplicateString(stringToDbl):  
➊ """ Prints a string twice on the same line.  
stringToDbl argument should be a string """  
print(stringToDbl \* 2)

La docstring debe estar en la primera línea de una función ➊. La docstring comienza y termina con un conjunto de tres comillas (""") y puede escribirse en tantas líneas como sea necesario.

#### **Saltos de línea en los argumentos**

Para facilitar la lectura de largas listas de argumentos a los programadores, Python te permite colocar los argumentos a lo largo de varias líneas. Por ejemplo, la llamada a la función de este programa tiene sus argumentos divididos en varias líneas para aumentar la legibilidad:

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
pos = mc.player.getPos()  
width = 10  
height = 12  
length = 13  
block = 103  
mc.setBlocks(pos.x, pos.y, pos.z,  
pos.x + width, pos.y + height, pos.z + length, block)

Los saltos de línea en los argumentos son especialmente útiles cuando quieres utilizar operadores matemáticos en los argumentos, cuando utilizas nombres largos de variables como argumentos o cuando tienes que proporcionar varios argumentos a una función.

#### **Valores de retorno de las funciones**

Hay dos tipos de funciones: las que devuelven un valor y las que no. Hasta ahora, has creado funciones que no devuelven ningún valor. Veamos las que sí devuelven un valor.

Devolver un valor desde una función es muy útil, porque permite que una función trabaje con datos y luego devuelva un valor al cuerpo principal del programa. Por ejemplo, imagina que vendes galletas. Para calcular el precio al que tienes que vender cada galleta para obtener suficientes beneficios, añades dos monedas de oro a la cantidad que pagaste por hacer la galleta y luego multiplicas la suma por 10. Utilizando una función que devuelva un valor, puedes escribir este cálculo y reutilizarlo en Python.

Cuando crees tus propias funciones, puedes utilizar la palabra clave return para devolver un valor de la función. Por ejemplo, aquí tienes el código para calcular el precio de venta de una galleta:

def calculateCookiePrice(cost):  
price = cost + 2  
price = price \* 10  
return price

Para devolver un valor, sólo tienes que escribir return seguido del valor que quieras, que en este caso es el precio. Para utilizar una función que devuelve un valor, la llamas en un lugar en el que se esperaría un valor. Por ejemplo, para establecer la variable precioDeCookie, llama a la función calcularPrecioDeCookie() e introduce un coste, como 6:

priceOfCookie = calculateCookiePrice(6) # Value will be 80

Puedes utilizar funciones que devuelvan valores para establecer los valores de las variables, y puedes utilizarlas en cualquier lugar en el que se espere que pongas un valor, incluso como argumento de otra función.

Las funciones que no devuelven un valor no pueden utilizarse para establecer los valores de las variables. Veamos rápidamente la diferencia.

Como la siguiente función devuelve un valor, se puede utilizar en cualquier lugar donde se pueda utilizar un valor, como para fijar una variable o incluso como argumento en otra llamada a una función:

def numberOfChickens():  
return 5  
  
coop = numberOfChickens()  
print(numberOfChickens())

Ejecuta este código para ver su resultado. Puedes tratar el resultado de la función como un valor e incluso hacer operaciones matemáticas con él. Aquí añado 4 al valor devuelto y lo almaceno en una variable llamada extraGallinas:

extraChickens = 4 + numberOfChickens() # Value of 9

Sin embargo, la siguiente función no tiene declaración de retorno, lo que significa que no puedes utilizarla en lugar de un valor. Lo único que puedes hacer es llamar a la función:

def chickenNoise():  
print("Cluck")  
  
chickenNoise()

Escribiendo este código en el editor de texto y ejecutándolo se imprime "Cluck", aunque no se puede utilizar en otras sentencias porque no devuelve un valor al programa. Por ejemplo, podría intentar concatenar la función con una cadena, así:

multipleNoises = chickenNoise() + ", Bork"

Si ejecutara este programa, obtendría el siguiente mensaje de error:

Traceback (most recent call last):  
File "<pyshell#3>", line 1, in <module>  
multipleNoises = chickenNoise + ", Bork"  
TypeError: unsupported operand type(s) for +: 'function' and 'str'

Este error significa que no puedes combinar esta función con una cadena, porque la función no devuelve un valor.

Sin embargo, si cambio el código para que devuelva un valor en lugar de limitarse a imprimirlo

def chickenNoise():  
return "Cluck"  
  
multipleNoises = chickenNoise() + ", Bork"  
print(multipleNoises)

el archivo se ejecutaría y mostraría la siguiente salida:

Cluck, Bork

Ten en cuenta esta diferencia. Recuerda incluir una sentencia return cuando la necesites y excluirla cuando tu función no necesite devolver un valor. Cuanto más experiencia adquieras con las funciones, más fácil te resultará decidir si quieres que tu función devuelva un valor.

#### **Misión nº 43: Recordatorio de ID de bloque**

Como Minecraft tiene tantos bloques, es difícil recordar todos los ID de bloque. Yo siempre recuerdo los valores de melón (103) y aire (0), pero olvido otros, ¡así que sigo teniendo que construir casas de melones!

Para que recordar me resulte más fácil, quiero que crees un programa para mí que devuelva los valores de los distintos bloques. Tu programa debe tener muchas funciones que me ayuden a recordar los ID de los bloques. El nombre de cada función debe ser el mismo que el del bloque cuyo valor devuelve. Por ejemplo, [el Listado 8-3](ch08.xhtml#ch8ex3) tiene una función llamada melón() que devuelve el valor del bloque melón (103).

*blockIds.py*

def melon():  
""" Returns the value of the melon block """  
return 103

*Listado 8-3: Inicio del programa que me ayudará a recordar los ID de los bloques*

Crea un nuevo archivo en IDLE y guárdalo como *blockIds.py* en la carpeta *functions*. Copia [el Listado 8-3](ch08.xhtml#ch8ex3) en el archivo y añádele funciones que devuelvan los valores de los siguientes bloques (consulta "[Hoja de trucos](app03.xhtml#app03)de ID de bloque" en la [página 299](app03.xhtml#page_299)):

- Agua

- Lana

- Lava

- TNT

- Flor

- Bloque de diamante

Cuando hayas añadido tus funciones, pruébalas llamándolas para crear bloques. Como tus nuevas funciones devuelven el valor de un bloque, puedes utilizarlas para establecer el valor de una variable que pasarás a la función setBlock(). El siguiente código te ayudará a empezar:

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
# Functions go here  
  
block = melon()  
pos = mc.player.getTilePos()  
mc.setBlock(pos.x, pos.y, pos.z, block)

[La Figura 8-3](ch08.xhtml#ch8fig3) muestra el resultado del programa completado con una prueba de la función melón(). Observa que la colocación de cualquier bloque está codificada en este programa; siempre colocará un bloque en su ubicación actual.



*Figura 8-3: Ahora no tengo que recordar los tipos de bloque, todo gracias a esta práctica función.*

**CONSEJO**

*Para colocar un bloque diamante, TNT o cualquier otro tipo de bloque, primero tendrás que definir la función que devuelve el valor del bloque que quieres. Luego tendrás que llamar a esa función en tu código, igual que yo he llamado a la* *función* melón() *en este ejemplo.*

**OBJETIVO EXTRA: MÁS BLOQUES**

Añade funciones adicionales para cualquier otro tipo de bloque que desees.

[anterior](ch08_1.html)[Subtema 2 de 5: (Ver todo)](ch08.html)[siguiente](ch08_3.html)