Aprende a programar con Minecraft

### **Variables globales y locales**

Cuando defines funciones, tienes que enfrentarte a un nuevo reto: el ámbito de una variable. El *ámbito* de una variable describe cómo puede acceder tu programa a sus datos. La mejor forma de aprender sobre el ámbito es verlo en acción, así que veamos algo de código. Supongamos que utilizas el siguiente código, que aumenta el número de huevos que tienes para una fiesta:

➊ eggs = 12  
  
def increaseEggs():  
➋ eggs += 1  
print(eggs)  
  
increaseEggs()

Dos variables se llaman huevos, una fuera de la función ➊ y otra dentro de la función ➋. Nada parece terriblemente incorrecto, pero Python lanzará un error. Aquí tienes parte del mensaje de error:

UnboundLocalError: local variable 'eggs' referenced before assignment

El problema es que la variable huevos está definida fuera de la función, pero cuando intentas añadirla dentro de la función, Python no puede ver la variable. Para Python, la variable dentro de la función es totalmente distinta de la que está fuera de la función, aunque tengan el mismo nombre. Python hace esto a propósito para evitar que las variables dentro de diferentes funciones compartan accidentalmente los mismos nombres y causen errores inesperados.

En el código Python, tienes dos formas de abordar las variables de un archivo: puedes hacer que una variable *sea global*, lo que significa que afecta a todo un programa o archivo, o hacer que una variable sea *local*, lo que significa que sólo puede ser vista por el código de una función o bucle concretos. En otras palabras, puedes utilizar la misma variable dentro y fuera de una función, o puedes hacer dos variables diferentes que afecten a distintas partes del código.

Una variable *global* será tratada como la misma variable dentro y fuera de una función. Cualquier cambio en la variable dentro de la función afectará a la variable que se definió fuera de la función y viceversa. Para hacer una variable global, utiliza la palabra clave global ➊:

eggs = 12  
  
def increaseEggs():  
➊ global eggs  
eggs += 1  
print(eggs)  
  
increaseEggs()

En este ejemplo, el valor de huevos será 13 cuando se imprima.

En cambio, puedes tratar la variable como una variable *local* para producir un efecto diferente. En este caso, las variables dentro y fuera de la función se tratarán como variables diferentes. Los cambios en la variable dentro de la función no afectarán a la variable fuera de la función y viceversa. Así que podrías cambiar el código para que la variable sea una variable local ➊, de esta forma

eggs = 12  
  
def increaseEggs():  
➊ eggs = 0  
eggs += 1  
➋ print(eggs)  
  
increaseEggs()  
➌ print(eggs)

Cuando se imprima el valor de huevos en la función ➋, será 1 porque el valor de la variable huevos fuera de la función no afecta a la variable local dentro de la función. El valor de los huevos dentro de la función aumentarHuevos( ) es 1, y la variable global huevos sigue teniendo el valor 12 ➌.

#### **Misión 46: Un bloque en movimiento**

Hace un tiempo, se me ocurrió que sería genial hacer que un bloque se moviera solo por el mundo de Minecraft. Cada segundo avanzaría. Si chocaba contra una pared, un árbol o algo alto, giraba y se dirigía en otra dirección. Sin embargo, si cayera en un agujero, se quedaría atascado y no podría escapar.

[El listado 8-5](ch08.xhtml#ch8ex5) es el inicio de un programa para crear un bloque mágico que se mueva por sí solo.

*movingBlock.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
import time  
  
  
def calculateMove():  
  
""" Changes the x and z variables for a block. If the block  
in front of the block is less than 2 blocks higher, it will move  
forward; otherwise it will try to move left, then backward,  
then finally right. """  
➊ # Create global variables here  
  
currentHeight = mc.getHeight(x, z) - 1  
  
forwardHeight = mc.getHeight(x + 1, z)  
rightHeight = mc.getHeight(x, z + 1)  
backwardHeight = mc.getHeight(x - 1, z)  
leftHeight = mc.getHeight(x, z - 1)  
  
if forwardHeight - currentHeight < 3:  
x += 1  
elif rightHeight - currentHeight < 3:  
z += 1  
elif leftHeight - currentHeight < 3:  
z -= 1  
elif backwardHeight - currentHeight < 3:  
x -= 1  
  
y = mc.getHeight(x, z)  
  
  
pos = mc.player.getTilePos()  
x = pos.x  
z = pos.z  
y = mc.getHeight(x, z)  
  
while True:  
# Calculate block movement  
calculateMove()  
  
# Place block  
mc.setBlock(x, y, z, 103)  
  
# Wait  
time.sleep(1)  
  
# Remove the block  
mc.setBlock(x, y, z, 0)

*Listado 8-5: Por desgracia, este código no funcionará hasta que se añadan variables globales.*

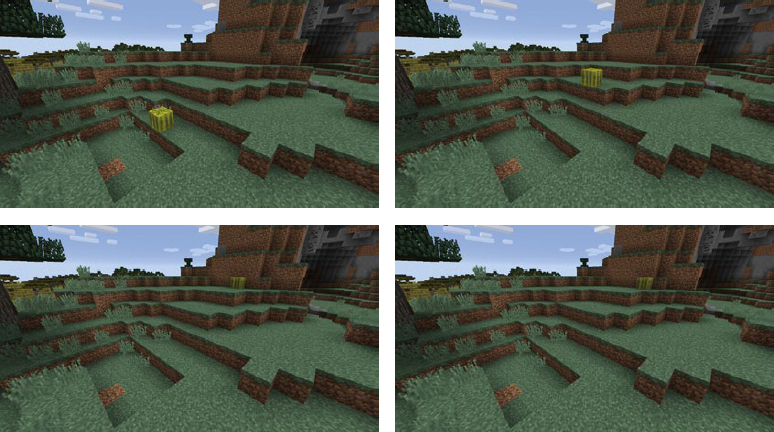
Pero este código aún no se ejecutará porque las variables de la función calculateMove() no son globales.

Tu misión es terminar el código del [Listado 8-5](ch08.xhtml#ch8ex5). Cópialo en IDLE y guárdalo como *moverBloque.py* en la carpeta de *funciones*. Añade código al inicio de la función para hacer globales las variables x, y y z. Las definiciones globales deben colocarse en ➊.

Cuando hayas declarado algunas variables globales, ejecuta el programa. Tu bloque debería moverse. [La Figura 8-6](ch08.xhtml#ch8fig6) muestra cómo el bloque se acerca a una pared y luego empieza a moverse a su alrededor.

**OBJETIVO EXTRA: UN BLOQUE MELÓN MÁS INTELIGENTE**

Cuando ejecutes el programa *movingBlock.py*, puede que notes que el bloque avanza más por el eje x, lo que a veces hace que se quede atascado en un bucle entre dos bloques. La razón es que el código no tiene en cuenta las direcciones en las que ya se ha movido el bloque y siempre intentará moverse primero a lo largo del eje x. ¿Puedes averiguar cómo almacenar la última dirección en la que se movió el bloque y cambiar la sentencia if para que se mueva primero en esa dirección?



*Figura 8-6: Ha sido divertido ver cómo el melón avanzaba y luego intentaba rodear la pared.*

[anterior](ch08_3.html)[Subtema 4 de 5: (Ver todo)](ch08.html)[siguiente](ch08_5.html)