Aprende a programar con Minecraft

## **12** **GANAR CLASE CON LA PROGRAMACIÓN ORIENTADA A** OBJETOS



La reutilización es un aspecto muy importante de la programación. Ahorra tiempo y esfuerzo. Ya lo has visto con los bucles y las funciones, y ahora aprenderás sobre la programación orientada a *objetos*.

La programación orientada a objetos es un enfoque de la programación que agrupa funciones y variables para crear *clases*. Cada clase puede utilizarse para crear *objetos* que comparten las mismas variables y funciones que la clase. Puedes crear muchos objetos a partir de la misma clase, haciendo que las variables y funciones de la clase sean reutilizables.

Cuando una función forma parte de una clase, se denomina *método*, y una variable que forma parte de una clase se denomina *atributo*.

En este capítulo, aprenderás programación orientada a objetos y a utilizar clases para reutilizar código. Dominar la programación orientada a objetos y las clases hace que construir programas sea pan comido, e incluso puedes utilizar la programación orientada a objetos para crear juegos. En las misiones de este capítulo, utilizarás clases para crear algunos programas básicos. Empezarás creando un edificio sencillo, pero pronto construirás una ciudad entera.

### **Conceptos básicos orientados a objetos**

La programación orientada a objetos es muy popular, y puedes utilizarla para crear todo tipo de programas geniales, pero puede ser un concepto difícil de entender. Vamos a relacionarlo con algo que te resulte más familiar: tú.

Eres una persona. Tienes una serie de métodos: puedes comer, respirar, dormir, contar hasta 10 y hacer muchas otras cosas. También tienes atributos: nombre, edad, altura, número de calzado, etc.

Tu amiga María tiene los mismos métodos que tú; ella también puede comer, respirar, dormir, contar hasta 10 y hacer muchas otras cosas. También tiene los mismos atributos (nombre, edad, etc.), aunque contengan valores distintos.

De hecho, todo el mundo tiene estos métodos y atributos. Puedes describir a las personas como una clase. Tú y María sois personas, por lo que podría decirse que ambos sois objetos de la clase Persona.

En programación orientada a objetos, los objetos se denominan *instancias* de una clase. Todos los objetos comparten los métodos y atributos de la clase, pero los valores de los atributos pueden ser diferentes para cada objeto.

Entremos en Python y creemos una clase.

### **Crear una clase**

Empezarás creando una clase y luego crearás todos tus objetos a partir de esa clase. Para crear una clase, utiliza la palabra clave class, el nombre con el que quieras llamar a la clase y la clase objeto entre paréntesis (explicaré la clase objeto en "[Heredar una clase](ch12.xhtml#ch12lev2sec07)" en [la página 278](ch12.xhtml#page_278)):

class ClassName(object):  
def \_\_init\_\_(self):  
# Body of init

Es una buena práctica poner en mayúsculas los nombres de tus clases. Así es más fácil distinguir las clases de las funciones, que deben empezar por minúscula.

Cuando crees una nueva clase, debes incluir el método \_\_init\_\_() y pasarle self como argumento. El argumento self es necesario para todos los métodos de una clase. Hace referencia a la clase a la que pertenece el método. El método \_\_init\_\_ () indica a Python lo que quieres que haga la clase cuando la utilices por primera vez en un programa. Esto se llama *inicializar* la clase, que es la abreviatura de \_\_init\_\_().

Por ejemplo, vamos a crear una clase llamada Gato y a crear algunos objetos gato. La clase Gato almacenará dos atributos para cada gato, su nombre y su peso en kilogramos. Cada objeto gato tendrá sus propios valores de nombre y peso. Abre un nuevo archivo en el editor de texto de IDLE y guárdalo como claseGato *.py* en una nueva carpeta llamada *clases*. Introduce el siguiente código para crear una clase llamada Gato:

*catClass.py*

class Cat(object):  
➊ def \_\_init\_\_(self, name, weight):  
➋ self.name = name  
➌ self.weight = weight

En este ejemplo, el método \_\_init\_\_( ) toma tres argumentos ➊. El primero es self, que es un argumento obligatorio en todo método de clase. El segundo argumento, nombre, y el último argumento, peso, son argumentos adicionales para crear atributos para todos los gatos.

Las dos últimas líneas crean los atributos nombre ➋ y peso ➌ y los establecen en los valores de los argumentos nombre y peso. Cuando creas atributos dentro de una clase, utilizas la notación de puntos con self. Los atributos siempre se identifican con un self, que indica a Python que un atributo pertenece a la clase.

A continuación, aprenderás a utilizar esta clase para crear instancias de objetos.

#### **Crear un objeto**

Utilizando la clase recién creada, vamos a crear algunos objetos gato, o instancias de la clase Gato.

Inicializar un objeto es similar a crear una variable. Para inicializar un objeto, introduces el nombre del objeto, un signo igual(=) y el nombre de la clase. Pasas argumentos a la clase entre paréntesis, igual que haces con una llamada a una función.

Por ejemplo, adoptemos un gato y llamémosle Pelusa. Utilizando la clase Gato, podemos crear un objeto gato llamado pelusa añadiendo el código siguiente en la última línea de *catClass.* py (fíjate en que no tiene sangría):

*catClass.py*

class Cat(object):  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
fluff = Cat("Fluff", 4.5)

Cuando creas un objeto, el número de argumentos que proporcionas depende de los argumentos de su función \_\_init\_\_(). Aquí incluimos dos argumentos, uno para el nombre ("Pelusa") y otro para el peso(4,5). No necesitas incluir el argumento self al crear un objeto porque Python lo añade automáticamente.

Crear un objeto también se conoce como *llamar a un constructor*. El método \_\_init\_\_ () suele denominarse constructor porque construye una clase cuando se le llama. El método \_\_init\_\_ () es un tipo especial de método porque no haces referencia a él por su nombre. En cambio, se ejecuta cuando creas un objeto utilizando el nombre de la clase. Por ejemplo, aquí el código pelusa = Gato("Pelusa", 4.5) llama al método \_\_init\_\_ (), que construye un objeto Gato llamado pelusa.

A continuación, aprenderás a acceder a los atributos del objeto pelusa.

#### **Acceder a los atributos**

Puedes acceder a los atributos de un objeto para obtener más información sobre él. Por ejemplo, añade el siguiente código a *catClass.* py después del objeto pelusa para imprimir el atributo peso del objeto pelusa:

*catClass.py*

print(fluff.weight)

El valor que se imprime al ejecutar el programa debe ser 4,5, porque ése es el valor que le pusiste al atributo peso cuando creaste el objeto.

Observa que estamos utilizando la notación de puntos entre el nombre del objeto, pelusa, y el atributo peso. El punto significa que quieres utilizar el atributo que pertenece a un objeto concreto. En este caso, el valor del atributo peso pertenece al objeto pelusa. Siempre que obtengas o establezcas el valor del atributo de un objeto, utilizarás la notación de punto.

Puedes cambiar el valor de un atributo como lo harías con cualquier otra variable, utilizando un signo igual(=). Por ejemplo, cambiemos el peso de Pelusa a 5 porque ha engordado durante las vacaciones de invierno. Para ello, cambia el atributo peso del objeto pelusa a 5:

*catClass.py*

fluff.weight = 5

Ahora, siempre que accedas al atributo peso del objeto pelusa, será 5.

Utilizando los conocimientos que ahora tienes sobre cómo crear una clase y crear una instancia de la misma, vamos a hacer cosas chulas en Minecraft.

#### **Misión nº 68: Objetos de Ubicación**

A lo largo del libro, has almacenado ubicaciones, como tu casa, un castillo o un palacio, en tu mundo Minecraft. Has utilizado variables, listas, tuplas y diccionarios para hacerlo de diversas formas.

También puedes crear y almacenar información relacionada, como ubicaciones, utilizando la programación orientada a objetos. Por ejemplo, puedes utilizar objetos para almacenar las coordenadas de un montón de ubicaciones diferentes.

Cada ubicación tiene una coordenada x, y y z, pero los valores de cada ubicación son diferentes. Creando una clase de ubicación, puedes almacenar y acceder a las coordenadas de distintas ubicaciones. Eso te ayudará a llevar un registro de todas las cosas increíbles que construyas en Minecraft. ¡Podrás acceder fácilmente a las coordenadas de todas tus creaciones Minecraft para poder teletransportar al jugador a ellas en un instante!

[El listado 12-1](ch12.xhtml#ch12ex1) contiene el inicio de la clase Ubicación. Cuando el código esté terminado, se podrá utilizar para almacenar las coordenadas de una ubicación en un único objeto. Copia el código en un nuevo archivo llamado *locationClass.py* en la carpeta de *clases*.

*locationClass.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
➊ class Location(object):  
def \_\_init\_\_(self, x, y, z):  
➋ self.x = x  
➌ # Add the y and z attributes here  
  
➍ bedroom = Location(64, 52, -8)  
➎ mc.player.setTilePos(bedroom.x, bedroom.y, bedroom.z)

*Listado 12-1: El inicio de la* *clase* Ubicación

Para iniciar la clase, he incluido la palabra clave class y he llamado a la clase Ubicación ➊. En ➍ está el código para inicializar un objeto llamado dormitorio, que almacenará la ubicación del dormitorio en mi casa de Minecraft. El método setTilePos( ) establece la posición del jugador en la ubicación del dormitorio -los atributos x, y y z del objeto dormitorio ➎. Sin embargo, el programa está incompleto. Tienes que terminar el método \_\_init\_\_( ) de la clase y establecer los atributos y y z a los valores de los argumentos pasados al método \_\_init\_\_(). Yo he fijado el valor del atributo x ➋, pero es tarea tuya hacer lo mismo con los atributos y y z ➌. ¡No olvides utilizar la ubicación de tu propio dormitorio en ➍!

[La Figura 12-1](ch12.xhtml#ch12fig1) muestra el programa completado en acción, ya que teletransporta al jugador a mi dormitorio.



*Figura 12-1: El programa ha teletransportado al jugador a mi dormitorio.*

**OBJETIVO EXTRA: HOGAR DULCE HOGAR**

¿A qué otras habitaciones de tu casa quieres teletransportarte? Crea más objetos con la clase Ubicación para recorrer tu casa con estilo.

### **Entender los métodos**

Las clases pueden contener métodos, que son funciones asociadas a la clase. Escribir métodos de clase te permite crear funciones que pueden utilizar todas las instancias de esa clase. Es una forma estupenda de ahorrar tiempo y reutilizar código, porque sólo tendrás que escribir un método.

Para crear un método, escribe una función en el cuerpo de una clase utilizando la palabra clave def. Ya has utilizado la palabra clave def en capítulos anteriores para crear funciones. Los métodos también se crean con la palabra clave def, pero están sangrados bajo la clase a la que pertenecen. Por ejemplo, vamos a actualizar la clase Gato en *catClass.py*. Queremos que el gato pueda comer, así que añadamos un método llamado comer() a la clase Gato. Introduce el código y realiza los cambios en catClass *.* py a medida que vayas avanzando:

*catClass.py*

class Cat(object):  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
def eat(self, food):  
self.weight = self.weight + 0.05  
print(self.name + " is eating " + food)

Observa que la definición del método y el cuerpo del método están sangrados con cuatro espacios extra para que Python sepa que pertenecen a la clase.

Al igual que las funciones, los métodos pueden recibir argumentos. Aquí el método comer( ) toma un argumento llamado comida que indica lo que está comiendo el gato. El método comer () aumenta el atributo peso del gato en 0,05 y luego imprime un mensaje indicando que el gato se está comiendo la comida.

Tras crear un objeto, puedes llamar a cualquiera de los métodos de su clase. Por ejemplo, puedes llamar al método comer() utilizando el objeto pelusa. Añade este código al final de *catClass.py*:

*catClass.py*

fluff = Cat("Fluff", 4.5)  
fluff.eat("tuna")

Aquí vemos nuestro código anterior, en el que creamos un objeto llamado pelusa que forma parte de la clase Gato. Luego llamamos al método comer() y le damos el argumento "atún". Cuando ejecutes el programa, la salida tendrá este aspecto:

Fluff is eating tuna

Ahora Pelusa está feliz comiendo atún. Recuerda que el método comer () también aumenta el atributo peso. Después de llamar al método comer( ), añade el código para imprimir el peso de Pelusa.

También puedes llamar a métodos desde dentro de la clase llamando a un método dentro de otro método. Vamos a crear otro método llamado comerYDormir( ) dentro de la clase Gato. El método comerYDormir () llama al método comer() y luego imprime que el gato está durmiendo. Añade este código a *catClass.py*, justo después del método eat( ) (asegúrate de que indentas el nuevo método como se muestra para que Python sepa que forma parte de la clase):

*catClass.py*

def eatAndSleep(self, food):  
self.eat(food)  
print(self.name + " is now sleeping...")

Para llamar a un método desde dentro de la clase a la que pertenece, añade self. al principio del nombre del método. Aquí se llama al método comer( ) utilizando self. comer(). Ten en cuenta que esto es diferente de llamar a un método fuera de una clase. Cuando haces eso, sólo tienes que introducir el nombre del objeto y el método al que llamas. Por ejemplo, el siguiente código llama al nuevo método comerYDormir () del objeto pelusa. Añádelo a tu archivo *catClass.py*. Debe ser la última línea de código de tu programa:

*catClass.py*

fluff.eatAndSleep("tuna")

Esta es la salida que deberías obtener al ejecutar el programa:

Fluff is eating tuna  
Fluff is now sleeping...

Aquí tienes el programa completo para que veas dónde están todas las piezas:

class Cat(object):  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
def eat(self, food):  
self.weight = self.weight + 0.05  
print(self.name + " is eating " + food)  
  
def eatAndSleep(self, food):  
self.eat(food)  
print(self.name + " is now sleeping...")  
  
fluff = Cat("Fluff", 4.5)  
print(fluff.weight)  
fluff.eat("tuna")  
fluff.eatAndSleep("tuna")

¡Llevemos las nuevas habilidades que has aprendido al mundo de Minecraft!

#### **Misión nº 69: Casa Fantasma**

Lo mejor de programar con Python y Minecraft es que puedes empezar con una idea tonta y correr con ella. Tu idea puede empezar siendo pequeña, pero con sólo unas pocas líneas de código, puedes construir un programa divertido muy rápidamente.

¿No sería divertido construir una casa fantasma que apareciera en un juego, sólo para desaparecer 30 segundos después? La casa podría reaparecer en otro lugar y volver a desaparecer si tú quisieras.

Aquí tienes la primera versión del programa de la casa fantasma. Guarda [el Listado 12-2](ch12.xhtml#ch12ex2) en un archivo llamado *casafantasma.py* en la carpeta de *clases*.

*ghostHouse.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
import time  
➊ class Building(object):  
➋ def \_\_init\_\_(self, x, y, z, width, height, depth):  
self.x = x  
self.y = y  
self.z = z  
  
self.width = width  
self.height = height  
self.depth = depth  
  
➌ def build(self):  
mc.setBlocks(self.x, self.y, self.z,  
self.x + self.width, self.y + self.height,  
self.z + self.depth, 4)  
  
mc.setBlocks(self.x + 1, self.y + 1, self.z + 1,  
self.x + self.width - 1, self.y + self.height - 1,  
self.z + self.depth - 1, 0)  
➍ # Call the buildDoor() and buildWindows() methods here  
  
➎ def clear(self):  
mc.setBlocks(self.x, self.y, self.z,  
self.x + self.width, self.y + self.height,  
self.z + self.depth, 0)  
➏ # Remove the doors and windows here  
  
pos = mc.player.getTilePos()  
x = pos.x  
y = pos.y  
z = pos.z  
  
➐ ghostHouse = Building(x, y, z, 10, 6, 8) ghostHouse.build()  
time.sleep(30)  
  
ghostHouse.clear()  
➑ ghostHouse.x = 8

*ghostHouse.py*

Listado 12-2*: La* clase Edificio *crea un edificio.*

[El](ch12.xhtml#ch12ex2) listado 12-2 utiliza una clase llamada Edificio ➊ con un método \_\_init\_\_( ) para establecer la posición y el tamaño de la casa ➋. Crea un objeto Edificio con el nombre CasaFantasma ➐. El edificio aparece y luego desaparece misteriosamente al cabo de 30 segundos utilizando los métodos build() ➌ y clear() ➎. El único problema es que no parece una casa. Ahora mismo parece un gran cascarón vacío hecho de adoquines.

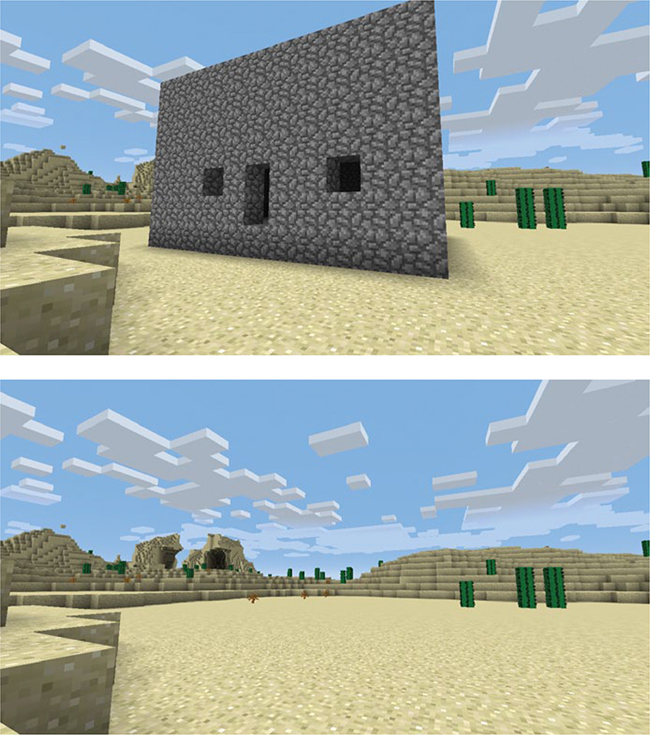
Tienes que hacer que la casa fantasma se parezca más a una casa y menos a un cascarón, porque los cascarones fantasma no dan tanto miedo como las casas fantasma. Para que el edificio parezca más una casa, tu misión es añadir un método que construya una puerta en la parte delantera de la casa y un segundo método que añada ventanas. Llama a estos dos métodos desde dentro del método build( ) para que se construyan al mismo tiempo ➍.

Después de añadir los métodos para construir la puerta y las ventanas, tendrás que actualizar el método clear () para eliminarlos ➏; de lo contrario, se quedarán atrás cuando desaparezca la casa.

Cuando hayas añadido los métodos adicionales, traslada el edificio a una nueva ubicación cambiando los atributos x, y y z del objeto ghostHouse y añadiendo más llamadas a los métodos build() y clear ( ). He empezado esto por ti cambiando la posición x de la casa ➑.

Cuando ejecutes el programa, la casa fantasma debería aparecer de repente y desaparecer 30 segundos después, para reaparecer en otro lugar. ¡Espeluznante!

[La Figura 12-2](ch12.xhtml#ch12fig2) muestra mi casa fantasma.



*Figura 12-2: La casa fantasma aparece y luego desaparece.*

**OBJETIVO EXTRA: MEJORA DE LA CASA**

De momento, la casa fantasma es muy básica. Utilizando las increíbles habilidades de Python que has aprendido en este libro, añade lo que quieras a la función build( ) para personalizar tu casa.

### **Devolver valores con métodos**

Al igual que las funciones, los métodos también pueden devolver valores, o los atributos de un objeto, utilizando la palabra clave return. Por ejemplo, supongamos que queremos convertir el peso del gato Fluff de kilogramos a gramos. Un kilogramo equivale a 1000 gramos, así que para hacer la conversión, multiplica el atributo peso por 1000 y devuélvelo. Añade el siguiente método getPesoEnGramos() a la clase Gato en *catClass.py*:

*catClass.py*

class Cat(object):  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
def getWeightInGrams(self):  
return self.weight \* 1000

Para obtener el valor devuelto por el método, creas un objeto y llamas al método. En el código siguiente, se utiliza el objeto pelusa, y se llama al método dentro de una función print( ) para obtener el peso del gato en gramos:

*catClass.py*

fluff = Cat("Fluff", 4.5)  
print(fluff.getWeightInGrams())

Ahora, cuando ejecutes el archivo, saldrá lo siguiente:

4500

En la próxima misión, ampliaremos el programa de la casa fantasma para incluir un método que devuelva información sobre el edificio.

#### **Misión nº 70: Castillo fantasma**

Tengo en mente todo tipo de nombres para los distintos lugares que he construido en mi mundo Minecraft: la casa de la playa, la granja de plantas, la granja de animales, el almacén, el palacio, el palacio submarino, el palacio subterráneo, y un montón más. ¡El problema es que los nombres sólo existen en mi cabeza!

Con las clases, puedes crear atributos como ubicación y tamaño para las cosas que construyas, como viste en la Misión 69[(página 267](ch12.xhtml#page_267)). ¡También puedes incluir nombres!

Pongamos un nombre a la casa fantasma y hagamos que Python lo recuerde por nosotros. Actualizaremos la clase Edificio de la Misión nº 69 para añadir un método extra que devuelva el nombre del edificio. Copia [el Listado 12-3](ch12.xhtml#ch12ex3) en un nuevo archivo llamado *castillofantasma.py* en la carpeta de *clases*.

*ghostCastle.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
import time  
  
➊ class NamedBuilding(object):  
➋ def \_\_init\_\_(self, x, y, z, width, height, depth, name):  
self.x = x  
self.y = y  
self.z = z  
  
self.width = width  
self.height = height  
self.depth = depth  
  
➌ self.name = name  
  
def build(self):  
mc.setBlocks(self.x, self.y, self.z,  
self.x + self.width, self.y + self.height,  
self.z + self.depth, 4)  
  
mc.setBlocks(self.x + 1, self.y + 1, self.z + 1,  
self.x + self.width - 1, self.y + self.height - 1,  
self.z + self.depth - 1, 0)  
  
def clear(self):  
mc.setBlocks(self.x, self.y, self.z,  
self.x + self.width, self.y + self.height,  
self.z + self.depth, 0)  
  
➍ def getInfo():  
# Add the body of the getInfo() method here  
  
pos = mc.player.getTilePos()  
x = pos.x  
y = pos.y  
z = pos.z  
ghostCastle = NamedBuilding(x, y, z, 10, 16, 16, "Ghost Castle")  
ghostCastle.build()  
➎ mc.postToChat(ghostCastle.getInfo())  
  
time.sleep(30)  
  
ghostCastle.clear()

*Listado 12-3:* EdificioNombrado *es muy similar a la* *clase* Edificio *, salvo que tiene un atributo extra llamado* nombre *y un método extra que devuelve una descripción del edificio.*

En primer lugar, he cambiado el nombre de la clase a EdificioNombrado para que no la confundamos con la clase Edificio de la misión anterior ➊. He añadido un argumento y un atributo extra al constructor llamado nombre ➋. El argumento te permite dar un nombre al edificio, y el constructor asigna el nombre al atributo nombre ➌.

Tu misión es añadir un método llamado getInfo() a la nueva clase EdificioNombrado que devuelva el nombre y la posición del edificio. Te he añadido el inicio del método getInfo () en ➍. Sólo tienes que añadir el cuerpo. El método getInfo () se invoca en el objeto castillofantasma en ➎ para que envíe la cadena devuelta por el método al chat de Minecraft. Por ejemplo, si el castillo fantasma está situado en x = -310, y = 64, z = 1081, el método getInfo ( ) debe devolver la cadena "La ubicación del castillo fantasma está en -310, 64, 1081".

[La Figura 12-3](ch12.xhtml#ch12fig3) muestra mi programa en funcionamiento. Aunque el castillo fantasma es más alto, se parece a la casa de la Misión nº 69. Esto se debe a que los métodos build( ) son los mismos para ambos, pero siéntete libre de cambiar tu versión del código para que tu edificio se parezca más a un castillo.



*Figura 12-3: Aparece la descripción del castillo fantasma.*

**OBJETIVO EXTRA: UNA CÁLIDA BIENVENIDA**

¿No sería genial que el nombre de cualquier edificio al que entraras apareciera automáticamente en el chat? Bueno, es posible, pero es un poco difícil. Si quieres intentarlo, puedes utilizar el programa *ducha.py* de la Misión 32[(página 124](ch06.xhtml#page_124)) como punto de partida. El archivo debe estar en tu carpeta *ifDeclaraciones*. Puedes utilizar el programa para detectar las coordenadas del jugador y, si está dentro del edificio, llamar al método getInfo() del objeto edificio.

### **Crear varios objetos**

Puedes crear varios objetos a partir de la misma clase creando objetos con nombres diferentes utilizando el mismo constructor de clase (recuerda que *constructor* es otro nombre para el método \_\_init\_\_() ). Por ejemplo, supongamos que encontramos una segunda gata llamada Stella que ahora es amiga de Fluff. Abre *catClass.py* e introduce el siguiente código para añadir a Stella:

*catClass.py*

class Cat(object):  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
fluff = Cat("Fluff", 4.5)  
stella = Cat("Stella", 3.9)

Ahora tenemos dos objetos gato, Pelusa y Stella. Cada uno tiene los mismos atributos, nombre y peso, pero con valores diferentes.

Añade el siguiente código a *catClass.* py para imprimir los nombres de los gatos:

*catClass.py*

print(fluff.name)  
print(stella.name)

Cuando ejecutes el archivo, obtendrás esta salida:

Fluff  
Stella

Los dos objetos gato también tienen acceso a los mismos métodos. Ambos pueden llamar a la función comer(). Añade este código a *catClass.*py:

*catClass.py*

fluff.eat("tuna")  
stella.eat("cake")

Y la salida tendrá este aspecto:

Fluff is eating tuna  
Stella is eating cake

Escribir una clase facilita mucho la creación de muchos objetos. ¡Intentemos crear varios objetos con Minecraft!

#### **Misión nº 71: Ciudad Fantasma**

¿Qué da más miedo que una casa fantasma? Exacto, dos casas fantasma. Pero tres casas fantasma darían aún más miedo. ¿Y más de tres casas fantasma? Tengo que dejar de pensar en esto, ¡o no podré dormir esta noche!

En la Misión 69[(página 267](ch12.xhtml#page_267)), creaste una clase que construye una casa que desaparece. Ahora puedes crear varios objetos utilizando la misma clase, y Python recordará cada uno de los atributos y métodos del objeto. Puedes hacer tantas casas como quieras, y puedes hacer que aparezcan y desaparezcan con facilidad.

Tu misión es crear cuatro o más objetos casa fantasma y disponerlos en un pueblo. Al cabo de cierto tiempo, haz que desaparezcan todas y reaparezcan en otro lugar del mapa, como en un verdadero pueblo fantasma.

Abre *ghostHouse.py* en IDLE: lo utilizaremos como base. Cuando hayas creado una casa en el programa *ghostHouse.* py, tu código debería tener este aspecto:

*ghostHouse.py*

ghostHouse = Building(17, 22, -54, 10, 6, 8)  
ghostHouse.build()  
  
time.sleep(30)  
  
ghostHouse.clear()

Guarda *ghostHouse*.py como un nuevo archivo llamado *ghostVillage.py*, y luego crea tres o más objetos en el archivo utilizando la clase Edificio para construir el pueblo. Para ayudarte a empezar, en [el Listado 12-4](ch12.xhtml#ch12ex4) he creado un segundo objeto llamado tienda. También he establecido las variables x, y y z para que contengan la posición actual del jugador, que encontraremos utilizando player.getTilePos(). Esto facilita la construcción de la aldea a tu alrededor.

*ghostVillage.py*

pos = mc.player.getTilePos()  
x = pos.x  
y = pos.y  
z = pos.z  
ghostHouse = Building(x, y, z, 10, 6, 8)  
shop = Building(x + 12, y, z, 8, 12, 10)  
# Create more ghost building objects here  
  
ghostHouse.build()  
shop.build()  
# Build more ghost building objects here  
  
time.sleep(30)  
  
ghostHouse.clear()  
shop.clear()

*Listado 12-4: Creación de múltiples objetos de construcción fantasma*

[La Figura 12-4](ch12.xhtml#ch12fig4) muestra mi aldea fantasma. Después de 30 segundos, los edificios fantasma desaparecen de repente.



*Figura 12-4: ¡Mira todos los edificios fantasma del pueblo fantasma!*

### **Atributos de clase**

A veces puedes querer establecer atributos que tengan el mismo valor para cada instancia de objeto de una clase. Sería redundante pasar el mismo argumento a la clase cada vez que se crea un objeto. En su lugar, puedes crear un atributo preestablecido en la clase, y todas las instancias de objetos de esa clase compartirán esos atributos.

Cuando varios objetos comparten el mismo atributo, se denomina *atributo de* clase. Por ejemplo, todos los objetos gato que hemos creado pertenecen a Craig (yo). Puedo volver a visitar la clase Gato en el archivo *catClass.py*, crear un atributo de clase llamado propietario y establecerlo como "Craig":

*catClass.py*

class Cat(object):  
owner = "Craig"  
  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight

Como puedes ver, los atributos de clase no utilizan self antes de su nombre. En este ejemplo, owner es un atributo de clase y self.name es un atributo. Observa que defines los atributos de clase fuera de la función \_\_init\_\_().

Los atributos de clase funcionan igual que cualquier otro atributo de un objeto. Por ejemplo, puedes acceder al valor de un atributo de clase como lo harías con un atributo normal. En este caso, para encontrar al propietario de Fluff, podemos imprimir el atributo de clase propietario del objeto Fluff:

*catClass.py*

fluff = Cat("Fluff", 4.5)  
print(fluff.owner)

El valor impreso debería ser "Craig". Si imprimiéramos el propietario de Stella, el valor sería el mismo porque los atributos de clase son iguales para todos los objetos de esa clase:

*catClass.py*

stella = Cat("Stella", 3.9)  
print(stella.owner)

El valor impreso aquí también es " Craig".

Puedes cambiar el valor de los atributos de clase para objetos individuales. Esto cambiará el valor del atributo para ese objeto, pero no para otros objetos de la clase. Por ejemplo, Stella ha sido adoptada por mi amigo Matthew, así que tenemos que cambiar el propietario de Stella a "Matthew":

*catClass.py*

stella.owner = "Matthew"  
print(stella.owner)  
print(fluff.owner)

Cuando se imprime el atributo propietario de stella, aparece "Matthew", pero el propietario de fluffsigue siendo "Craig".

Después de todos los cambios que hemos hecho en *catClass.py*, el programa final tiene el siguiente aspecto. También está disponible en los recursos del libro en [*https://www.nostarch.com/programwithminecraft/.*](https://www.nostarch.com/programwithminecraft/)

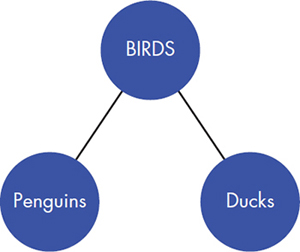
*catClass.py*

class Cat(object):  
owner = "Craig"  
  
def \_\_init\_\_(self, name, weight):  
self.name = name  
self.weight = weight  
  
def eat(self, food):  
self.weight = self.weight + 0.05  
print(self.name + " is eating " + food)  
  
def eatAndSleep(self, food):  
self.eat(food)  
print(self.name + " is now sleeping...")  
  
def getWeightInGrams(self):  
return self.weight \* 1000  
  
fluff = Cat("Fluff", 4.5)  
print(fluff.owner)  
stella = Cat("Stella", 3.9)  
print(stella.owner)  
  
print(fluff.weight)  
fluff.eat("tuna")  
fluff.eatAndSleep("tuna")  
  
print(fluff.getWeightInGrams())  
print(fluff.name)  
print(stella.name)  
  
fluff.eat("tuna")  
stella.eat("cake")  
  
stella.owner = "Matthew"  
print(stella.owner)  
print(fluff.owner)

Ahora que ya has visto cómo utilizar objetos, vamos a ver cómo hacerlos aún más potentes con la herencia.

### **Entender la herencia**

La*herencia* se produce cuando las clases comparten los mismos métodos y atributos que otras clases. Por ejemplo, los patos son un tipo de ave. Comparten los mismos métodos que otras aves (volar, comer, etc.) y tienen los mismos atributos que otras aves (peso, envergadura, etc.). Así que se podría decir que *los patos* heredan sus atributos y métodos de la clase *aves*. [La figura 12-5](ch12.xhtml#ch12fig5) muestra esta relación en un diagrama.



*Figura 12-5: Tanto los pingüinos como los patos son clases de aves.*

La clase de la que heredan otras clases se llama *superclase*; la clase que hereda de la superclase se llama *subclase*.

La herencia es útil porque permite crear diferencias sutiles entre objetos similares. Por ejemplo, los pingüinos también son un tipo de ave, pero pueden nadar bajo el agua, a diferencia de la mayoría de las aves. Para representar a los pingüinos, tienes que crear una subclase que herede de la clase pájaro pero que tenga adaptaciones para que los pingüinos puedan nadar bajo el agua. Estas adaptaciones son la razón por la que creas subclases: puedes conservar las características principales de la superclase para evitar tener que escribir el código de nuevo, y sólo tienes que añadir los métodos y atributos que necesites en la subclase.

#### **Heredar una clase**

Cuando una subclase hereda de una superclase, la subclase puede utilizar todos los métodos y atributos de la superclase. La subclase también puede añadir clases y atributos adicionales sin alterar la superclase original.

Utilicemos el ejemplo de los pájaros para ilustrarlo. Primero escribiremos el código de la superclase Pájaro. Abre un nuevo archivo en IDLE, llámalo *clasepájaro.py* y añade el siguiente código para crear la clase:

*birdClass.py*

➊ class Bird(object):  
➋ def \_\_init\_\_(self, name, wingspan):  
self.name = name  
self.wingspan = wingspan  
  
➌ def birdcall(self):  
print("chirp")  
➍ def fly(self):  
print("flap")

Creamos una clase llamada Ave ➊, pero fíjate en que la clase Ave hereda de objeto. La clase objeto es una clase base sobre la que se construirán todas las demás clases. Todas las clases heredan de la clase objeto, y se utiliza cuando no hay otras superclases de las que heredar. Aunque haya varios niveles de herencia en los que muchas clases hereden unas de otras, la clase objeto siempre será la superclase utilizada en el nivel más alto de herencia.

El método \_\_init\_\_() de la clase Pájaro toma dos argumentos que establecen dos atributos: el nombre del pájaro y su envergadura ➋. Tiene dos métodos: birdcall() ➌ y fly() ➍. De momento, el método birdcall () sólo imprime "chirrido" y el método fly () sólo imprime "aleteo".

En el mismo archivo, crea un objeto llamado pájarojardín utilizando la clase Pájaro:

*birdClass.py*

gardenBird = Bird("Geoffrey", 12)  
gardenBird.birdcall()  
gardenBird.fly()

Este código imprimirá

chirp  
flap

Ahora que has creado una superclase, puedes crear una subclase que herede de la superclase pero que tenga su propio método. Lo harás en la siguiente sección.

#### **Añadir nuevos métodos a las subclases**

Añadamos una clase para pingüinos a *birdClass.py* y llamémosla Pingüino. Como los pingüinos pueden nadar bajo el agua, puedes añadir un método extra a la clase Pingüino llamado nadar():

*birdClass.py*

class Penguin(Bird):  
def swim(self):  
print("swimming")

Cuando defines una subclase y quieres que herede de otra superclase en lugar de un objeto, pones entre paréntesis el nombre de la superclase de la que hereda. Observa que no he creado un método \_\_init\_\_() para la clase Pingüino. La razón es que hereda de la clase Pájaro, por lo que utiliza el método \_\_init\_\_( ) de la clase Pájaro. Utilicemos ese método \_\_init\_\_( ) y probemos la función swim( ) creando un pingüino:

*birdClass.py*

sarahThePenguin = Penguin("Sarah", 10)  
sarahThePenguin.swim()

Este código mostrará lo siguiente:

swimming

La clase Pingüino también puede utilizar los métodos fly() y birdcall() porque los hereda de Bird.

*birdClass.py*

sarahThePenguin.fly()  
sarahThePenguin.birdcall()

En este caso, el resultado será el siguiente:

flap  
chirp

Pero aletear y piar no tienen sentido para un pingüino, porque los pingüinos no pueden volar y su canto de pájaro es más bien un graznido. Aprenderemos a anular métodos heredados y a solucionar esto en "[Anular métodos y atributos](ch12.xhtml#ch12lev1sec08)", en la [página 282](ch12.xhtml#page_282).

Pero antes, volvamos a Minecraft y creemos unos nuevos edificios fantasma utilizando la herencia.

#### **Misión nº 72: Hotel Fantasma**

Tanto las casas como los hoteles son tipos de edificios: tienen puertas, ventanas, habitaciones, escaleras y paredes. Los hoteles no son más que casas elegantes con extras como balcones, muchas habitaciones y una bonita entrada.

¿Cómo puedes programar unos hoteles fantasma utilizando el código que ya has creado para las casas fantasma? La estructura básica de los edificios es la misma. Digamos que la única diferencia es que los hoteles fantasma tienen métodos adicionales para crear alfombras dentro de las habitaciones y añadir flores alrededor del borde del edificio. Eso significa que la clase hotel fantasma puede heredar todos los métodos de la clase casa fantasma. Entonces, lo único que necesita la clase hotel fantasma son dos métodos extra para las alfombras y las flores.

En IDLE, crea un nuevo archivo y guárdalo como hotelfantasma *.py* en la carpeta *clases*. Copia y pega en el archivo el código de la clase Edificio del programa *ghostHotel*.py.

Crea una nueva clase llamada FancyBuilding que herede de la clase Building. La clase FancyBuilding debe tener un nuevo método llamado upgrade() que añada alfombra dentro del edificio y flores alrededor de las paredes. [El listado 12-5](ch12.xhtml#ch12ex5) muestra mi código para el método upgrade( ), pero siéntete libre de personalizar tus hoteles.

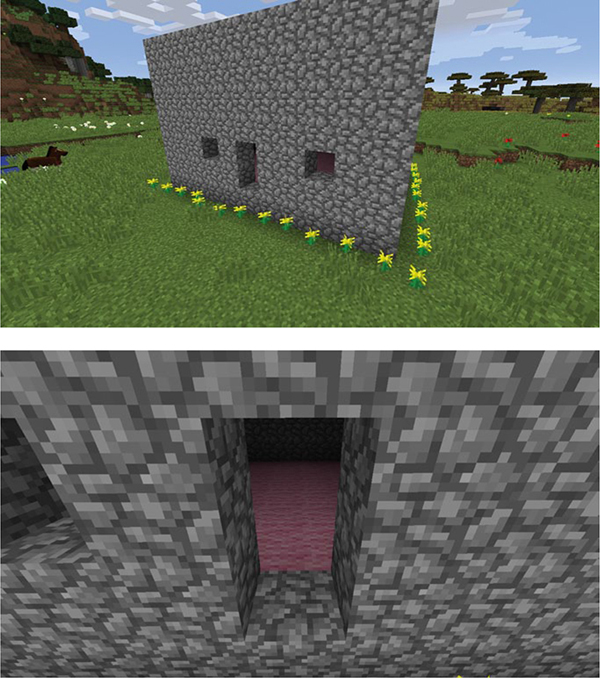
*ghostHotel.py*

# Create a FancyBuilding class here  
  
def upgrade(self):  
# Carpet  
mc.setBlocks(self.x + 1, self.y, self.z + 1,  
self.x + self.width - 1, self.y, self.z + self.depth - 1,  
35, 6)  
  
# Flowers  
mc.setBlocks(self.x - 1, self.y, self.z -1,  
self.x - 1, self.y, self.z + self.depth + 1,  
37)  
mc.setBlocks(self.x - 1, self.y, self.z - 1,  
self.x + self.width + 1, self.y, self.z – 1,  
37)  
mc.setBlocks(self.x + self.width + 1, self.y, self.z - 1,  
self.x + self.width + 1, self.y, self.z + self.depth + 1,  
37)  
mc.setBlocks(self.x - 1, self.y, self.z + self.depth + 1,  
self.x + self.width + 1, self.y, self.z + self.depth = 1,  
37)  
  
# Create an instance of the FancyBuilding class  
# Call the build() and upgrade() methods

*Listado 12-5: Un método de la* *clase* FancyBuilding que añade alfombra *y* flores al edificio

Cuando hayas creado la clase y añadido el nuevo método, crea una instancia de la clase FancyBuilding y llámala hotel fantasma. Construye el hotel fantasma con el método build( ) y, a continuación, añade los elementos adicionales con el método upgrade().

[La Figura 12-6](ch12.xhtml#ch12fig6) muestra mi elegante hotel fantasma.



*Figura 12-6: ¡Mira qué flores y qué alfombra!*

**OBJETIVO EXTRA: ALDEA FANTASMA**

En la Misión nº 71, creaste un pueblo fantasma en el que todos los edificios parecían más o menos iguales. Es raro ver edificios idénticos en los pueblos reales. Cambia el programa del pueblo fantasma creando varias clases que hereden de la clase Edificio. Podrías crear una clase Tienda, una clase Hospital y una clase Restaurante, por ejemplo. Luego, cuando crees los objetos, podrás elegir qué tipo de edificio crear utilizando las distintas clases.

### **Sobrescribir métodos y atributos**

Una subclase puede redefinir métodos y atributos de su superclase. Esto es útil cuando quieres utilizar el mismo nombre para un método pero quieres que se comporte de forma diferente en la subclase.

En "[Comprender la herencia](ch12.xhtml#ch12lev1sec07)", en [la página 277](ch12.xhtml#page_277), creamos una clase Pájaro y una clase Pingüino. La clase Pingüino heredó de Ave, por lo que compartió todos sus métodos. Pero los pingüinos no pueden volar, y su canto de pájaro es más un graznido que un gorjeo. Así que debemos cambiar los métodos fly() y birdcall() para reflejar esto. Abre *birdClass.py* y añade este código:

*birdClass.py*

class Penguin(Bird):  
def swim(self):  
print("swimming")  
  
➊ def birdcall(self):  
print("sort of a quack")  
  
➋ def fly(self):  
print("Penguins cannot fly :(")

He hecho dos cambios en la clase Pingüino. He añadido un método birdcall( ) ➊ y un método fly () ➋. Como ambos métodos se escriben igual que en la superclase Pájaro, anularán los métodos de la superclase.

Llama a los métodos añadiendo este código a *clasepájaro.py*:

*birdClass.py*

sarahThePenguin.fly()  
sarahThePenguin.birdcall()

Ahora, cuando ejecutes el programa, obtendrás este resultado:

Penguins cannot fly :(  
sort of a quack

Anular un método de una superclase cambiará lo que hace el método para la subclase, pero no para la superclase. Así, los pingüinos no podrán volar, pero otros pájaros que hereden de Pájaro podrán volar.

También puedes sobrescribir el método \_\_init\_\_() en una subclase. Esto significa que cuando se crea el objeto de la subclase, puede tener atributos o comportamientos diferentes a los de la superclase.

Por ejemplo, creemos una subclase Loro de Pájaro en el mismo archivo. Los loros pueden ser de distintos colores, así que incluyamos un argumento extra en el método \_\_init\_\_() para un atributo de color:

*birdClass.py*

class Parrot(Bird):  
➊ def \_\_init\_\_(self, name, wingspan, color):  
self.name = name  
self.wingspan = wingspan  
self.color = color

He incluido un nuevo método \_\_init\_\_ () para la clase Loro que tiene un argumento extra, color ➊, en comparación con la clase original Pájaro.

Ahora, cuando creamos un nuevo objeto Loro, podemos acceder al atributo color. También podemos acceder a los métodos birdcall() y fly( ) porque se heredaron de la superclase Bird:

*birdClass.py*

freddieTheParrot = Parrot("Freddie", 12, "blue")  
print(freddieTheParrot.color)  
freddieTheParrot.fly()  
freddieTheParrot.birdcall()

Este código producirá lo siguiente:

blue  
flap  
chirp

Recuerda que puedes sobrescribir cualquier método que una subclase herede de una superclase; incluso puedes sobrescribir el método \_\_init\_\_(). Esto te da mucho control sobre los objetos y sus numerosos atributos y métodos.

Después de todos los cambios que hemos hecho en *clasepájaro.py*, el programa final tiene el siguiente aspecto. También está disponible en los recursos del libro en [*https://www.nostarch.com/programwithminecraft/.*](https://www.nostarch.com/programwithminecraft/)

*birdClass.py*

class Bird(object):  
def \_\_init\_\_(self, name, wingspan):  
self.name = name  
self.wingspan = wingspan  
  
def birdcall(self):  
print("chirp")  
  
def fly(self):  
print("flap")  
  
  
class Penguin(Bird):  
def swim(self):  
print("swimming")  
  
def birdcall(self):  
print("sort of a quack")  
  
def fly(self):  
print("Penguins cannot fly :(")  
  
  
class Parrot(Bird):  
def \_\_init\_\_(self, name, wingspan, color):  
self.name = name  
self.wingspan = wingspan  
self.color = color  
  
gardenBird = Bird("Geoffrey", 12)  
gardenBird.birdcall()  
gardenBird.fly()  
  
sarahThePenguin = Penguin("Sarah", 10)  
sarahThePenguin.swim()  
sarahThePenguin.fly()  
sarahThePenguin.birdcall()  
  
freddieTheParrot = Parrot("Freddie", 12, "blue")  
print(freddieTheParrot.color)  
freddieTheParrot.fly()  
freddieTheParrot.birdcall()

Probarás a anular métodos y atributos en la próxima misión.

#### **Misión nº 73: Árbol fantasma**

Has creado varias formas de edificios fantasma. Llevémoslo al siguiente nivel y creemos un árbol fantasma. Es una idea increíble, pero ¿cómo podemos hacerlo? La clase Edificio es para edificios, que tienen paredes y techos; los árboles no tienen paredes ni techos. ¡No te preocupes! Puedes evitarlo modificando tu clase Edificio fantasma.

Al igual que los edificios fantasma, el árbol fantasma aparecerá y desaparecerá utilizando los métodos build() y clear(). Pero los métodos tienen que funcionar de forma diferente porque los árboles tienen un aspecto distinto al de las casas. Así que tienes que crear una clase que herede de la clase Edificio y anular los métodos build( ) y clear( ).

Para empezar, he cogido la función que crea un árbol del archivo *forest.py*[(página 153](ch08.xhtml#page_153)) y la he puesto en [el Listado 12-6](ch12.xhtml#ch12ex6). Cópiala en un nuevo archivo llamado *ghostTree.py* en la carpeta de *clases*.

*ghostTree.py*

from mcpi.minecraft import Minecraft  
mc = Minecraft.create()  
  
# Paste the ghostHouse.py program here  
# Create a Tree class here  
  
➊ def growTree(x, y, z):  
""" Creates a tree at the coordinates given """  
wood = 17  
leaves = 18  
  
# Trunk  
mc.setBlocks(x, y, z, x, y + 5, z, wood)  
  
# Leaves  
mc.setBlocks(x - 2, y + 6, z - 2, x + 2, y + 6, z + 2, leaves)  
mc.setBlocks(x - 1, y + 7, z - 1, x + 1, y + 7, z + 1, leaves)  
  
# Create build() and clear() methods for the Tree class here

*Listado 12-6: Una función para crear un árbol*

Para terminar el programa, copia y pega el código de la clase Edificio de *ghostHouse.* py en el nuevo archivo. A continuación, crea una nueva clase llamada Árbol que herede de la clase Edificio. Dentro de la clase Árbol, añade un método build() y un método clear() para anular los métodos de la clase Edificio y construir un árbol en lugar de una casa. Asegúrate de incluir el argumento self delante de los atributos en el método final growTree() ➊.

Cuando hayas creado el programa, crea un objeto Árbol llamado árbol fantasma. Llama al método build( ) para que aparezca el árbol, espera un poco y luego haz que desaparezca utilizando clear().

[La Figura 12-7](ch12.xhtml#ch12fig7) muestra el resultado de mi programa.



*Figura 12-7: ¡Es un árbol espeluznante!*

**OBJETIVO EXTRA: BOSQUE FANTASMA**

Modifica el código de *ghostTree.py* para construir un bosque fantasma. ¿Qué tipo de tesoros crees que podrías encontrar en un bosque fantasma?

### **Lo que has aprendido**

Acabas de aprender los fundamentos de uno de los conceptos más importantes de la programación actual: ¡la programación orientada a objetos! Has aprendido a escribir una clase y a crear objetos, y has aprendido a utilizar la herencia para personalizar las clases y el comportamiento de los objetos. Podrás aplicar esta habilidad tan útil no sólo en Minecraft, ¡sino también en cualquier tipo de aventura de programación que decidas emprender a continuación!