Desconocido

## 8 Cómo utilizar clases y objetos



¿Por qué una jirafa es como una acera? Porque tanto una jirafa como una acera son *cosas* , que en inglés se conocen como nouns y en Python como *objects .* En programación, los objetos son una forma de organizar el código y descomponer las cosas para trabajar más fácilmente con ideas complejas. (Utilizamos un objeto en el [Capítulo 4](ch04.xhtml#ch04) cuando trabajamos con el objeto Turtle del módulo turtle ).

Para entender bien cómo funcionan los objetos en Python, tenemos que pensar en los tipos de objetos. Empecemos con las jirafas y las aceras.

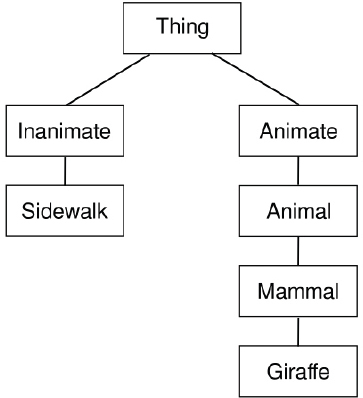
Una jirafa es un tipo de mamífero, que es un tipo de animal. Una jirafa también es un objeto animado: está viva.

No hay mucho que decir sobre una acera, aparte de que no es un ser vivo. Llamémosla objeto inanimado (en otras palabras, no está viva). Los términos *mamífero* , *animal* , *animado* e *inanimado* son formas de clasificar las cosas.



### Clasificación de las cosas

En Python, los objetos se definen mediante *clases* , que clasifican los objetos en grupos. Por ejemplo, el diagrama de árbol de [la Figura 8-1](ch08.xhtml#ch08fig01) muestra las clases en las que encajan las jirafas y las aceras basándonos en nuestras definiciones anteriores.



Figura*8-1: Diagrama en árbol de algunas* clases

La clase principal es Thing . Debajo de la clase Thing , tenemos Inanimate y Animate . Estas clases se dividen a su vez en Sidewalk para Inanimate , y Animal , Mammal , y Giraffe para Animate .

Podemos utilizar clases para organizar fragmentos de código. Por ejemplo, considera el módulo turtle . Todas las cosas que puede hacer el módulo tortuga de Python -como avanzar, retroceder, girar a la izquierda y a la derecha- son funciones de la clase Turtle .  Un objeto es un miembro de una clase, y podemos crear cualquier número de objetos para una clase, a lo que nos referiremos en breve.

Ahora vamos a crear el mismo conjunto de clases que aparece en nuestro diagrama de árbol, empezando por arriba. Definimos las clases utilizando la palabra clave class seguida de un nombre. Thing es la clase más amplia, así que la crearemos primero:

>>> class Thing:  
 pass

Llamamos a la clase Thing y utilizamos la sentencia pass para que Python sepa que no vamos a dar más información. La palabra clave pass se utiliza cuando queremos proporcionar una clase o función pero no queremos dar más detalles por el momento.

A continuación, añadiremos las demás clases y construiremos algunas relaciones entre ellas.

### Hijos y padres

Si una clase forma parte de otra clase, se considera *hija* de esa clase, y la otra clase es su *padre .* Las clases pueden ser a la vez *hijas* y *padres de* otras clases. En nuestro diagrama de árbol, la clase que está encima de otra clase es su padre y la que está debajo es su hijo. Por ejemplo, Inanimate y Animate son ambas hijas de la clase Thing , lo que significa que Thing es su padre.

Para indicar a Python que una clase es hija de otra clase, añadimos el nombre de la clase padre entre paréntesis después del nombre de nuestra nueva clase, así

>>> class Inanimate(Thing):  
 pass  
  
>>> class Animate(Thing):  
 pass

Aquí creamos una clase llamada Inanimate y le decimos a Python que su clase padre es Thing . A continuación, creamos una clase llamada Animate y le decimos a Python que su clase padre es también Thing .

Vamos a crear la clase Sidewalk con la clase padre Inanimate :

>>> class Sidewalk(Inanimate):  
 pass

Podemos organizar las clases Animal , Mammal , y Giraffe utilizando también sus clases padre:

>>> class Animal(Animate):  
 pass  
  
>>> class Mammal(Animal):  
 pass  
  
>>> class Giraffe(Mammal):  
 pass

### Añadir objetos a las clases

Ahora tenemos un montón de clases, pero ¿qué tal si añadimos algo más de información a esas clases? Digamos que tenemos una jirafa llamada Reginald. Sabemos que pertenece a la clase Giraffe , pero ¿qué utilizamos -en términos de programación- para describir a la jirafa llamada Reginald?

Llamamos a Reginald *objeto* (también conocido como *instancia* ) de la clase Giraffe . Para "introducir" a Reginald en Python, utilizamos este pequeño fragmento de código:

>>> reginald = Giraffe()

Este código le dice a Python que cree un objeto de la clase Giraffe y lo asigne a la variable reginald . Como cuando llamamos a una función, el nombre de la clase va seguido de paréntesis. Más adelante en este capítulo, veremos cómo crear objetos y utilizar parámetros entre paréntesis.

Pero, ¿qué hace el objeto reginald ? Pues, de momento, nada. Para que nuestros objetos sean útiles, cuando creamos nuestras clases, también tenemos que definir funciones que puedan utilizarse con los objetos de esa clase. En lugar de limitarnos a utilizar la palabra clave pass inmediatamente después de definir la clase, podemos añadir definiciones de funciones.

### Definir funciones de clases

[El Capítulo 7](ch07.xhtml#ch07) introdujo las funciones como una forma de reutilizar código. Cuando definimos una función asociada a una clase, lo hacemos del mismo modo que definimos cualquier otra función, salvo que la sangramos debajo de la definición de la clase. Por ejemplo, aquí tienes una función normal que no está asociada a una clase:

>>> def this\_is\_a\_normal\_function():  
 print('I am a normal function')

Y aquí tienes un par de funciones que se definen para una clase:

>>> class ThisIsMySillyClass:  
 def this\_is\_a\_class\_function():  
 print('I am a class function')  
 def this\_is\_also\_a\_class\_function():  
 print('I am also a class function. See?')

### Añadir características de clase

Considera las clases hijas de la clase Animate que definimos en la sección Hijos y padres. Podemos añadir características a cada clase que describan lo que es y lo que puede hacer. Una *característica* es un rasgo que comparten todos los miembros de la clase (y sus hijos).

Por ejemplo, ¿qué tienen en común todos los animales? Para empezar, respiran, se mueven y comen. ¿Y los mamíferos? Los mamíferos alimentan a sus crías con leche, y también respiran, se mueven y comen. Sabemos que las jirafas comen hojas de lo alto de los árboles. Y como todos los mamíferos, alimentan a sus crías con leche, respiran, se mueven y comen. Cuando añadimos estas características a nuestro diagrama de árbol, obtenemos algo como [la Figura 8-2 .](ch08.xhtml#ch08fig02)

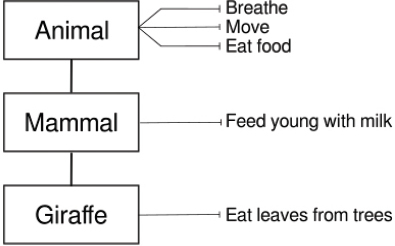


Figura 8-2*: Clases con* características

Estas características pueden considerarse acciones o funciones, cosas que puede hacer un objeto de esa clase.

Para añadir una función a una clase, utilizamos la palabra clave def . Así, la clase Animal tendrá este aspecto:

>>> class Animal(Animate):  
 def breathe(self):  
 pass  
 def move(self):  
 pass  
 def eat\_food(self):  
 pass

En la primera línea de este listado, definimos la clase como antes, pero en lugar de utilizar pass en la línea siguiente, definimos una función llamada breathe y le damos el parámetro self . El parámetro self es una forma de que una función de la clase llame a otra función de la clase (y de la clase padre). Veremos cómo se utiliza este parámetro más adelante.



En la línea siguiente, pass indica a Python que no vamos a dar más información sobre la función breathe porque no va a hacer nada por ahora. A continuación, añadimos las funciones move y eat\_food y utilizamos la palabra clave pass para ambas. En breve recrearemos nuestras clases y pondremos código adecuado en las funciones. Esta es una forma habitual de desarrollar programas.

**NOTA**

*A menudo, los programadores crean clases con funciones que no hacen nada, como forma de averiguar qué debe hacer la clase antes de entrar en los detalles de las funciones individuales.*

También podemos añadir funciones a las clases Mammal y Giraffe . Cada clase podrá utilizar las características (o funciones) de su clase madre, lo que significa que no necesitas crear una clase realmente complicada. En su lugar, puedes poner tus funciones en la clase padre más alta en la que se aplique la característica. (Esto hace que tus clases sean más sencillas y fáciles de entender).

>>> class Mammal(Animal):  
 def feed\_young\_with\_milk(self):  
 pass  
  
>>> class Giraffe(Mammal):  
 def eat\_leaves\_from\_trees(self):  
 pass

En el código anterior, la clase Mammal proporciona una función feed \_young\_with\_milk . La clase Giraffe es una clase hija (o *subclase* ) de Mammal y proporciona otra función: eat\_leaves\_from\_trees .

### ¿Por qué utilizar clases y objetos?

Ahora hemos añadido funciones a nuestras clases, pero ¿por qué utilizar clases y objetos cuando podrías escribir funciones normales llamadas breathe , move , eat\_food , etc.?

Para responder a esa pregunta, utilizaremos nuestra jirafa llamada Reginald, que creamos antes como un objeto de la clase Giraffe , de la siguiente manera:

>>> reginald = Giraffe()

Como reginald es un objeto, podemos llamar (o ejecutar) funciones proporcionadas por la clase Giraffe y sus clases padre. Llamamos a funciones de un objeto utilizando el operador punto ( . ) y el nombre de la función. Para decirle a Reginald que se mueva o que coma, podemos llamar a las funciones de la siguiente manera:

>>> reginald = Giraffe()  
>>> reginald.move()  
>>> reginald.eat\_leaves\_from\_trees()

Supongamos que Reginald tiene una amiga jirafa llamada Harriet. Vamos a crear otro objeto Giraffe llamado harriet :

>>> harriet = Giraffe()

Como estamos utilizando objetos y clases, podemos decirle a Python de qué jirafa estamos hablando cuando queramos ejecutar la función move . Por ejemplo, si queremos hacer que Harriet se mueva pero dejar a Reginald en su sitio, podríamos llamar a la función move utilizando nuestro objeto harriet , así:

>>> harriet.move()

En este caso, sólo se movería Harriet.

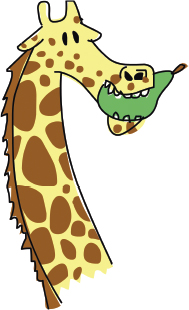
Cambiemos un poco nuestras clases para que esto sea más evidente. Añade una sentencia print a cada función en lugar de utilizar pass :

>>> class Animal(Animate):  
 def breathe(self):  
 print('breathing')  
 def move(self):  
 print('moving')  
 def eat\_food(self):  
 print('eating food')  
>>> class Mammal(Animal):  
 def feed\_young\_with\_milk(self):  
 print('feeding young')  
  
>>> class Giraffe(Mammal):  
 def eat\_leaves\_from\_trees(self):  
 print('eating leaves')

Ahora, cuando creamos nuestros objetos reginald y harriet y llamamos a funciones sobre ellos, podemos ver que realmente ocurre algo:

>>> reginald = Giraffe()  
>>> harriet = Giraffe()  
>>> reginald.move()  
moving  
>>> harriet.eat\_leaves\_from\_trees()  
eating leaves

En las dos primeras líneas, creamos las variables reginald y harriet , que son objetos de la clase Giraffe . A continuación, llamamos a la función move sobre reginald , y Python imprime moving en la línea siguiente. Del mismo modo, llamamos a la función eat\_leaves\_from\_trees sobre harriet , y Python imprime eating leaves . Si se tratara de jirafas reales, en lugar de objetos de un ordenador, una jirafa estaría caminando y la otra comiendo.



**NOTA**

*Las funciones definidas para las clases se llaman en realidad methods . Los términos son casi intercambiables, salvo que los métodos sólo pueden invocarse sobre objetos de una clase. Otra forma de decir esto es que un método está asociado a una clase, pero una función no. Dado que son casi lo mismo, en este libro utilizaremos el término function* .

### Objetos y clases en imágenes

Intentemos adoptar un enfoque más gráfico de los objetos y las clases y volvamos al módulo turtle con el que jugamos en el [Capítulo 4 .](ch04.xhtml#ch04) Cuando utilizamos turtle.Turtle() , Python crea un objeto de la clase Turtle que proporciona el módulo turtle (similar a nuestros objetos reginald y harriet ). Podemos crear dos objetos Turtle (llamados Avery y Kate) igual que creamos dos jirafas:

>>> import turtle  
>>> avery = turtle.Turtle()  
>>> kate = turtle.Turtle()

Cada objeto tortuga ( avery y kate ) es miembro de la clase Turtle .

Ahora es cuando los objetos empiezan a ser poderosos. Una vez creados nuestros objetos Turtle , podemos llamar a funciones en cada uno de ellos, y se dibujarán independientemente. Prueba este código:

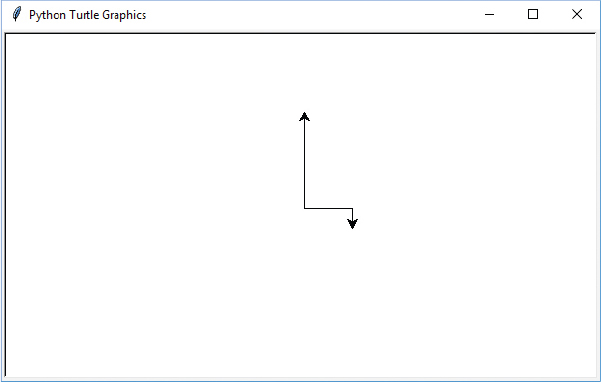
>>> avery.forward(50)  
>>> avery.right(90)  
>>> avery.forward(20)

Con esta serie de instrucciones, le decimos a Avery que avance 50 píxeles, gire 90 grados a la derecha y avance 20 píxeles para que termine mirando hacia abajo. Recuerda que las tortugas siempre empiezan mirando hacia la derecha.

Ahora es el momento de mover a Kate.

>>> kate.left(90)  
>>> kate.forward(100)

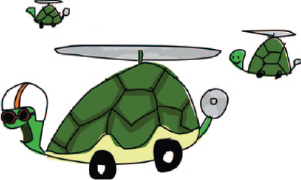
Le decimos a Kate que gire 90 grados a la izquierda y luego que avance 100 píxeles para que termine mirando hacia arriba. Hasta ahora, tenemos una línea con puntas de flecha que se mueven en dos direcciones distintas, y la punta de cada flecha representa un objeto tortuga distinto: Avery apunta hacia abajo y Kate hacia arriba (ver [Figura 8-3](ch08.xhtml#ch08fig03) ).



Figura*8-3: Kate y Avery*

Ahora vamos a añadir otra tortuga, Jacob, y a moverla sin molestar a Kate ni a Avery:

>>> jacob = turtle.Turtle()  
>>> jacob.left(180)  
>>> jacob.forward(80)



Primero, creamos un nuevo objeto Turtle llamado jacob ; luego lo giramos 180 grados a la izquierda y lo movemos 80 píxeles hacia delante. Nuestro dibujo con tres tortugas debería parecerse a [la Figura 8-4 .](ch08.xhtml#ch08fig04)

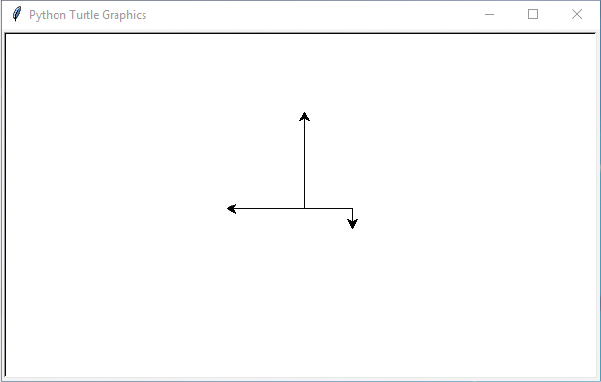


Figura 8-4*: Kate, Avery y Jacob*

Cada vez que llamamos a turtle.Turtle() para crear una tortuga, añadimos un objeto nuevo e independiente. Cada objeto sigue siendo una instancia de la clase Turtle , y podemos utilizar las mismas funciones en cada objeto. Pero como estamos utilizando objetos, podemos mover cada tortuga de forma independiente. Al igual que nuestros objetos independientes Giraffe (Reginald y Harriet), Avery, Kate y Jacob son objetos independientes Turtle . Si creamos un nuevo objeto con el mismo nombre de variable que un objeto que ya hemos creado, el objeto antiguo no desaparecerá necesariamente.

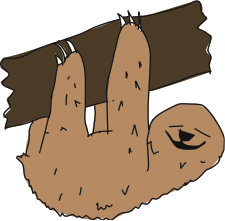
### Otras características útiles de los objetos y las clases

Las clases y los objetos facilitan la agrupación de funciones. También son muy útiles cuando queremos pensar en un programa en trozos más pequeños.

Por ejemplo, considera una aplicación de software enorme, como un procesador de textos o un juego de ordenador en 3D. Para la mayoría de la gente es casi imposible comprender completamente programas grandes como éstos, porque contienen mucho código. Pero si divides estos programas monstruosos en partes más pequeñas, cada una de ellas empezará a tener sentido, siempre que conozcas su lenguaje de programación, por supuesto.

Al escribir un programa grande, dividirlo también te permite repartir el trabajo entre otros programadores. Los programas más complicados (como tu navegador web) fueron escritos por muchas personas, o equipos de personas, trabajando en diferentes partes al mismo tiempo en todo el mundo.

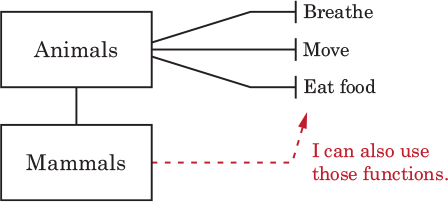
Imagina que queremos ampliar algunas de las clases que hemos creado en este capítulo ( Animal , Mammal , y Giraffe ), pero tenemos demasiado trabajo y nuestros amigos se ofrecen a ayudarnos. Podríamos dividir el trabajo de escribir el código de modo que una persona trabaje en la clase Animal , otra en la clase Mammal y otra en la clase Giraffe .



### Funciones heredadas

Puedes darte cuenta de que quien acabe trabajando en la clase Giraffe tiene suerte, porque cualquier función creada por las personas que trabajan en las clases Animal y Mammal también puede ser utilizada por la clase Giraffe . La clase Giraffe *hereda* funciones de la clase Mammal , que, a su vez, hereda funciones de la clase Animal . En otras palabras, cuando creamos un objeto Giraffe , podemos utilizar funciones definidas en la clase Giraffe , así como funciones definidas en las clases Mammal y Animal . Y, del mismo modo, si creamos un objeto Mammal , podemos utilizar funciones definidas en la clase Mammal , así como en su clase padre, Animal .

Observa de nuevo las relaciones entre las clases Animal , Mammal y Giraffe . La clase Animal es la clase padre de la clase Mammal , y Mammal es la clase padre de la clase Giraffe ( [Figura 8-5](ch08.xhtml#ch08fig05) ).



*Figura 8-5: Clases y funciones heredadas*

Aunque reginald sea un objeto de la clase Giraffe , podemos llamar a la función move que definimos en la clase Animal , porque las funciones definidas en cualquier clase padre están disponibles para sus clases hijas:

>>> reginald = Giraffe()  
>>> reginald.move()  
moving

De hecho, todas las funciones que definimos en las clases Animal y Mammal pueden llamarse desde nuestro objeto reginald porque las funciones son heredadas:

>>> reginald = Giraffe()  
>>> reginald.breathe()  
breathing  
>>> reginald.eat\_food()  
eating food  
>>> reginald.feed\_young\_with\_milk()  
feeding young

En este código, creamos un objeto de la clase Giraffe llamado reginald . Cuando llamamos a cada función, imprime un mensaje independientemente de si la función está definida en Giraffe o en una clase padre.

### Funciones que llaman a otras funciones

Cuando llamamos a funciones de un objeto, utilizamos el nombre de la variable del objeto. Por ejemplo, podemos llamar a la función move sobre la jirafa Reginald de la siguiente manera:

>>> reginald.move()

Para que una función de la clase Giraffe llame a la función move , utilizaríamos el parámetro self . El parámetro self es una forma de que una función de la clase llame a otra función. Por ejemplo, supongamos que añadimos una función llamada find\_food a la clase Giraffe :

>>> class Giraffe(Mammal):  
 def find\_food(self):  
 self.move()  
 print('I\'ve found food!')  
 self.eat\_food()

Hemos creado una función que combina otras dos funciones, lo que es bastante habitual en programación. A menudo, escribes una función que hace algo útil, que luego puedes utilizar dentro de otra función. (Haremos esto en [el Capítulo 11](ch11.xhtml#ch11) , donde escribiremos funciones más complejas para crear un juego).

Utilicemos self para añadir algunas funciones a la clase Giraffe :

>>> class Giraffe(Mammal):  
 def find\_food(self):  
 self.move()  
 print('I\'ve found leaves!')  
 self.eat\_food()  
 def eat\_leaves\_from\_trees(self):  
 print('tear leaves from branches')  
 self.eat\_food()  
 def dance\_a\_jig(self):  
 self.move()  
 self.move()  
 self.move()  
 self.move()

Utilizamos las funciones eat\_food y move de la clase padre Animal para definir eat \_leaves\_from\_trees y dance\_a\_jig para la clase Giraffe , porque son funciones heredadas. Al añadir funciones que llaman a otras funciones, cuando creamos objetos de estas clases, podemos llamar a una única función que hace más de una cosa. Mira lo que ocurre cuando llamamos a la función dance\_a\_jig :



>>> reginald = Giraffe()  
>>> reginald.dance\_a\_jig()  
moving  
moving  
moving  
moving

En este código, nuestra jirafa se mueve cuatro veces (es decir, el texto moving se imprime cuatro veces).

Si llamamos a la función find\_food , se imprimen tres líneas:

>>> reginald.find\_food()  
moving  
I've found leaves!  
eating food

### Inicializar un objeto

A veces, al crear un objeto, queremos establecer algunos valores (también llamados *propiedades* ) para su uso posterior. Cuando *inicializamos* un objeto, lo estamos preparando para ser utilizado.

Por ejemplo, supongamos que queremos establecer el número de manchas de nuestros objetos jirafa cuando se crean (o inicializan). Para ello, creamos una función \_\_init\_\_ (fíjate en los dos caracteres de subrayado a cada lado, para un total de cuatro). La función init establece las propiedades de un objeto cuando éste se crea por primera vez. Python llamará automáticamente a esta función cuando creemos un nuevo objeto. A continuación te explicamos cómo utilizarla.

>>> class Giraffe(Mammal):  
 def \_\_init\_\_(self, spots):  
 self.giraffe\_spots = spots

En primer lugar, definimos la función \_\_init\_\_ con los parámetros self y spots . Al igual que las otras funciones que hemos definido en la clase, la función \_\_init\_\_ también necesita tener self como primer parámetro. A continuación, establecemos el parámetro spots en una variable de objeto (su propiedad) llamada giraffe\_spots utilizando el parámetro self . Puedes pensar que esta línea de código dice: "Toma el valor del parámetro spots y guárdalo para más tarde (utilizando la variable de objeto giraffe\_spots )". Del mismo modo que una función de una clase puede llamar a otra función utilizando el parámetro self , también se accede a las variables de la clase utilizando self .

A continuación, si creamos un par de nuevos objetos jirafa (llamados ozwald y gertrude ) y mostramos su número de manchas, podrás ver la función de inicialización en acción:

>>> ozwald = Giraffe(100)  
>>> gertrude = Giraffe(150)  
>>> print(ozwald.giraffe\_spots)  
100  
>>> print(gertrude.giraffe\_spots)  
150

En primer lugar, creamos una instancia de la clase Giraffe utilizando el valor 100 como parámetro. Esto tiene el efecto de llamar a la función \_\_init\_\_ y utilizar 100 para el valor del parámetro spots . A continuación, creamos otra instancia de la clase Giraffe con un valor de 150. Por último, imprimimos la variable de objeto giraffe\_spots para cada uno de nuestros objetos jirafa, y vemos que los resultados son 100 y 150. ¡Ha funcionado!

Recuerda que cuando creamos un objeto de una clase, como ozwald en este ejemplo, podemos referirnos a sus variables o funciones utilizando el operador punto y el nombre de la variable o función que queremos utilizar (por ejemplo, ozwald.giraffe\_spots ). Pero cuando creamos funciones dentro de una clase, nos referimos a esas mismas variables (y a otras funciones) utilizando el parámetro self ( self.giraffe\_spots ).

### Lo que has aprendido

En este capítulo, hemos utilizado clases para crear categorías de cosas y hemos creado objetos (o instancias) de esas clases. Aprendiste cómo el hijo de una clase hereda las funciones de su padre, y que aunque dos objetos sean de la misma clase, no son necesariamente clones. Por ejemplo, dos objetos jirafa pueden tener un número distinto de manchas.

Aprendiste cómo llamar (o ejecutar) funciones en un objeto y cómo las variables de objeto son una forma de guardar valores en esos objetos. Por último, utilizamos el parámetro self en funciones para referirnos a otras funciones y variables. Estos conceptos son fundamentales en Python, y los verás varias veces a lo largo del resto de este libro.

### Rompecabezas de programación

Prueba los siguientes ejemplos para experimentar con la creación de tus propias funciones. Puedes encontrar las soluciones en http://python-for-kids.com [*.*](http://python-for-kids.com)

#### #nº 1: El barajado de la jirafa

Añade funciones a la clase Giraffe para mover los pies izquierdo y derecho de la jirafa hacia delante y hacia atrás. Una función para mover la pata izquierda hacia delante podría ser así

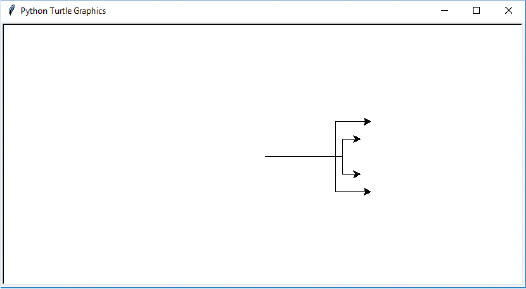
>>> def left\_foot\_forward(self):  
 print('left foot forward')

A continuación, crea una función llamada dance para enseñar a bailar a nuestras jirafas (la función llamará a las cuatro funciones de pies que acabas de crear). El resultado de llamar a esta nueva función será un simple baile:

>>> harriet = Giraffe()  
>>> harriet.dance()  
left foot forward  
left foot back  
right foot forward  
right foot back  
left foot back  
right foot back  
right foot forward  
left foot forward

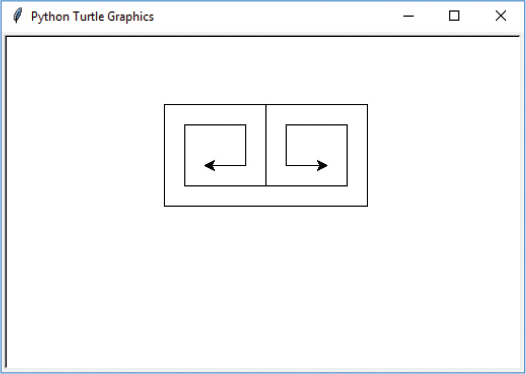
#### #nº 2: Horquilla de tortuga

Crea la siguiente imagen de una horquilla lateral utilizando cuatro objetos Turtle (la longitud exacta de las líneas no es importante). ¡Recuerda importar primero el módulo turtle !



#### #3: Dos pequeñas espirales

Crea la siguiente imagen de dos pequeñas espirales utilizando dos objetos Turtle (de nuevo, el tamaño exacto de las espirales no es importante).



#### #4: Cuatro espirales pequeñas

Tomemos las dos espirales que hemos creado en el código anterior y hagamos una imagen especular para crear cuatro espirales, que deberían parecerse a la imagen siguiente.

