# 9 Clases



*Object-oriented programming (OOP)* es uno de los enfoques más eficaces para escribir software. En la programación orientada a objetos, escribes que representan cosas y situaciones del mundo real, y creas basándote en estas clases. Cuando escribes una clase, defines el comportamiento general que puede tener toda una categoría de objetos. *classes* *objects*

Cuando creas objetos individuales a partir de la clase, cada objeto se equipa automáticamente con el comportamiento general; luego puedes dar a cada objeto los rasgos únicos que desees. Te sorprenderá lo bien que pueden modelarse situaciones del mundo real con la programación orientada a objetos.

Crear un objeto a partir de una clase se llama *instantiation*, y trabajas con *instances* de una clase. En este capítulo escribirás clases y crearás instancias de esas clases. Especificarás el tipo de información que puede almacenarse en las instancias y definirás las acciones que pueden realizarse con ellas. También escribirás clases que amplíen la funcionalidad de clases existentes, de modo que clases similares puedan compartir funcionalidades comunes, y puedas hacer más con menos código. Almacenarás tus clases en módulos e importarás clases escritas por otros programadores en tus propios archivos de programa.

Aprender sobre programación orientada a objetos te ayudará a ver el mundo como lo ve un programador. Te ayudará a entender tu código, no sólo lo que ocurre línea a línea, sino también los conceptos más amplios que hay detrás. Conocer la lógica que hay detrás de las clases te capacitará para pensar lógicamente, de modo que puedas escribir programas que aborden con eficacia casi cualquier problema que encuentres.

Las clases también te harán la vida más fácil a ti y a los demás programadores con los que trabajes cuando os enfrentéis a retos cada vez más complejos. Cuando tú y otros programadores escribáis código basado en el mismo tipo de lógica, podréis entender el trabajo de los demás. Tus programas tendrán sentido para las personas con las que trabajes, lo que permitirá a todos lograr más.

## Crear y utilizar una clase

Puedes modelar casi cualquier cosa utilizando clases. Empecemos escribiendo una clase sencilla, Dog, que represente a un perro, no a un perro en particular, sino a cualquier perro. ¿Qué sabemos de la mayoría de los perros? Bueno, todos tienen un nombre y una edad. También sabemos que la mayoría de los perros se sientan y se dan la vuelta. Esos dos datos (nombre y edad) y esos dos comportamientos (sentarse y darse la vuelta) irán en nuestra clase Dog porque son comunes a la mayoría de los perros. Esta clase le dirá a Python cómo crear un objeto que represente a un perro. Una vez escrita nuestra clase, la utilizaremos para crear instancias individuales, cada una de las cuales representará a un perro concreto.

### Creación de la clase Perro

Cada instancia creada a partir de la clase Dog almacenará un name y un age, y daremos a cada perro la capacidad de sit() y roll\_over():

**dog.py**

❶ class Dog:  
 """A simple attempt to model a dog."""  
  
❷ def \_\_init\_\_(self, name, age):  
 """Initialize name and age attributes."""  
❸ self.name = name  
 self.age = age  
  
❹ def sit(self):  
 """Simulate a dog sitting in response to a command."""  
 print(f"{self.name} is now sitting.")  
  
 def roll\_over(self):  
 """Simulate rolling over in response to a command."""  
 print(f"{self.name} rolled over!")

Hay mucho que notar aquí, pero no te preocupes. Verás esta estructura a lo largo de este capítulo y tendrás mucho tiempo para acostumbrarte a ella. Primero definimos una clase llamada Dog ❶. Por convención, los nombres en mayúsculas se refieren a clases en Python. No hay paréntesis en la definición de la clase porque la estamos creando desde cero. A continuación, escribimos un docstrip que describe lo que hace esta clase.

### El método \_\_init\_\_()

Una función que forma parte de una clase es un *method*. Todo lo que aprendiste sobre las funciones se aplica también a los métodos; la única diferencia práctica por ahora es la forma en que llamaremos a los métodos. El método \_\_init\_\_() ❷ es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Dog. Este método tiene dos guiones bajos iniciales y dos guiones bajos finales, una convención que ayuda a evitar que los nombres de métodos por defecto de Python entren en conflicto con tus nombres de métodos. Asegúrate de utilizar dos guiones bajos a cada lado de \_\_init\_\_(). Si utilizas sólo uno a cada lado, el método no será llamado automáticamente cuando utilices tu clase, lo que puede dar lugar a errores difíciles de identificar.

Definimos el método \_\_init\_\_() para que tenga tres parámetros: self, name, y age. El parámetro self es obligatorio en la definición del método, y debe ir en primer lugar, antes que los demás parámetros. Debe incluirse en la definición porque cuando Python llame a este método más adelante (para crear una instancia de Dog), la llamada al método pasará automáticamente el argumento self. Cada llamada a un método asociado a una instancia pasa automáticamente self, que es una referencia a la propia instancia; da acceso a la instancia individual a los atributos y métodos de la clase. Cuando hagamos una instancia de Dog, Python llamará al método \_\_init\_\_() de la clase Dog. Pasaremos a Dog() un nombre y una edad como argumentos; self se pasa automáticamente, por lo que no necesitamos pasarlo. Siempre que queramos crear una instancia a partir de la clase Dog, proporcionaremos valores sólo para los dos últimos parámetros, name y age.

Las dos variables definidas en el cuerpo del método \_\_init\_\_() tienen cada una el prefijo self ❸. Cualquier variable prefijada con self está disponible para todos los métodos de la clase, y también podremos acceder a estas variables a través de cualquier instancia creada a partir de la clase. La línea self.name = name toma el valor asociado al parámetro name y lo asigna a la variable name, que se adjunta a la instancia que se está creando. El mismo proceso ocurre con self.age = age. Las variables accesibles a través de instancias como ésta se denominan *attributes*.

La clase Dog tiene otros dos métodos definidos: sit() y roll\_over() ❹. Como estos métodos no necesitan información adicional para ejecutarse, sólo los definimos para que tengan un parámetro, self. Las instancias que creemos más adelante tendrán acceso a estos métodos. En otras palabras, podrán sentarse y darse la vuelta. Por ahora, sit() y roll\_over() no hacen gran cosa. Simplemente imprimen un mensaje diciendo que el perro se sienta o se da la vuelta. Pero el concepto puede ampliarse a situaciones realistas: si esta clase formara parte de un juego de ordenador, estos métodos contendrían código para hacer que un perro animado se sentara y se diera la vuelta. Si esta clase se escribiera para controlar un robot, estos métodos dirigirían los movimientos que hacen que un perro robótico se siente y se dé la vuelta.

### Crear una instancia a partir de una clase

Piensa en una clase como un conjunto de instrucciones sobre cómo crear una instancia. La clase Dog es un conjunto de instrucciones que indica a Python cómo crear instancias individuales que representen a perros concretos.

Hagamos una instancia que represente a un perro concreto:

class Dog:  
 --snip--  
  
❶ my\_dog = Dog('Willie', 6)  
  
❷ print(f"My dog's name is {my\_dog.name}.")  
❸ print(f"My dog is {my\_dog.age} years old.")

La clase Dog que utilizamos aquí es la que acabamos de escribir en el ejemplo anterior. Aquí, le decimos a Python que cree un perro cuyo nombre sea 'Willie' y cuya edad sea 6 ❶. Cuando Python lee esta línea, llama al método \_\_init\_\_() en Dog con los argumentos 'Willie' y 6. El método \_\_init\_\_() crea una instancia que representa a este perro concreto y establece los atributos name y age utilizando los valores que le hemos proporcionado. Python devuelve entonces una instancia que representa a este perro. Asignamos esa instancia a la variable my\_dog. La convención de nomenclatura es útil aquí; normalmente podemos asumir que un nombre en mayúsculas como Dog se refiere a una clase, y un nombre en minúsculas como my\_dog se refiere a una única instancia creada a partir de una clase.

#### Acceder a los atributos

Para acceder a los atributos de una instancia, se utiliza la notación con puntos. Accedemos al valor del atributo my\_dog' name ❷ escribiendo:

my\_dog.name

La notación con puntos se utiliza a menudo en Python. Esta sintaxis muestra cómo Python encuentra el valor de un atributo. Aquí, Python busca en la instancia my\_dog y luego encuentra el atributo name asociado a my\_dog. Se trata del mismo atributo denominado self.name en la clase Dog. Utilizamos el mismo enfoque para trabajar con el atributo age ❸.

El resultado es un resumen de lo que sabemos sobre my\_dog:

My dog's name is Willie.  
My dog is 6 years old.

#### Llamada a métodos

Después de crear una instancia de la clase Dog, podemos utilizar la notación punto para llamar a cualquier método definido en Dog. Hagamos que nuestro perro se siente y se dé la vuelta:

class Dog:  
 --snip--  
  
my\_dog = Dog('Willie', 6)  
my\_dog.sit()  
my\_dog.roll\_over()

Para llamar a un método, indica el nombre de la instancia (en este caso, my\_dog) y el método al que quieres llamar, separados por un punto. Cuando Python lee my\_dog.sit(), busca el método sit() en la clase Dog y ejecuta ese código. Python interpreta la línea my\_dog.roll\_over() del mismo modo.

Ahora Willie hace lo que le decimos:

Willie is now sitting.  
Willie rolled over!

Esta sintaxis es bastante útil. Cuando a los atributos y métodos se les han dado nombres adecuadamente descriptivos como name, age, sit(), y roll\_over(), podemos deducir fácilmente lo que se supone que debe hacer un bloque de código, incluso uno que nunca hemos visto antes.

#### Crear varias instancias

Puedes crear tantas instancias de una clase como necesites. Vamos a crear un segundo perro llamado your\_dog:

class Dog:  
 --snip--  
  
my\_dog = Dog('Willie', 6)  
your\_dog = Dog('Lucy', 3)  
  
print(f"My dog's name is {my\_dog.name}.")  
print(f"My dog is {my\_dog.age} years old.")  
my\_dog.sit()  
  
print(f"\nYour dog's name is {your\_dog.name}.")  
print(f"Your dog is {your\_dog.age} years old.")  
your\_dog.sit()

En este ejemplo creamos un perro llamado Willie y un perro llamado Lucy. Cada perro es una instancia independiente con su propio conjunto de atributos, capaz del mismo conjunto de acciones:

My dog's name is Willie.  
My dog is 6 years old.  
Willie is now sitting.  
  
Your dog's name is Lucy.  
Your dog is 3 years old.  
Lucy is now sitting.

Aunque utilizáramos el mismo nombre y la misma edad para el segundo perro, Python seguiría creando una instancia independiente a partir de la clase Dog. Puedes crear tantas instancias de una clase como necesites, siempre que des a cada instancia un nombre de variable único o que ocupe un lugar único en una lista o diccionario.

## Pruébalo tú mismo

9-1. Restaurante: Crea una clase llamada Restaurant. El método \_\_init\_\_() de Restaurant debe almacenar dos atributos: un restaurant\_name y un cuisine\_type. Crea un método llamado describe\_restaurant() que imprima estos dos datos, y un método llamado open\_restaurant() que imprima un mensaje indicando que el restaurante está abierto.

Crea una instancia llamada restaurant a partir de tu clase. Imprime los dos atributos individualmente, y luego llama a ambos métodos.

9-2. Tres Restaurantes: Empieza con tu clase del Ejercicio 9-1. Crea tres instancias diferentes a partir de la clase, y llama a describe\_restaurant() para cada instancia.

9-3. Usuarios: Crea una clase llamada User. Crea dos atributos llamados first\_name y last\_name, y luego crea otros atributos que se suelen almacenar en un perfil de usuario. Crea un método llamado describe\_user() que imprima un resumen de la información del usuario. Crea otro método llamado greet\_user() que imprima un saludo personalizado al usuario.

Crea varias instancias que representen a distintos usuarios, y llama a ambos métodos para cada usuario.

## Trabajar con clases e instancias

Puedes utilizar clases para representar muchas situaciones del mundo real. Una vez que escribas una clase, pasarás la mayor parte del tiempo trabajando con instancias creadas a partir de esa clase. Una de las primeras tareas que querrás hacer es modificar los atributos asociados a una instancia concreta. Puedes modificar los atributos de una instancia directamente o escribir métodos que actualicen los atributos de formas específicas.

### La clase Coche

Escribamos una nueva clase que represente a un coche. Nuestra clase almacenará información sobre el tipo de coche con el que estamos trabajando, y tendrá un método que resuma esta información:

**car.py**

class Car:  
 """A simple attempt to represent a car."""  
  
❶ def \_\_init\_\_(self, make, model, year):  
 """Initialize attributes to describe a car."""  
 self.make = make  
 self.model = model  
 self.year = year  
  
❷ def get\_descriptive\_name(self):  
 """Return a neatly formatted descriptive name."""  
 long\_name = f"{self.year} {self.make} {self.model}"  
 return long\_name.title()  
  
❸ my\_new\_car = Car('audi', 'a4', 2024)  
print(my\_new\_car.get\_descriptive\_name())

En la clase Car, definimos el método \_\_init\_\_() con el parámetro self primero ❶, igual que hicimos con la clase Dog. También le damos otros tres parámetros: make, model, y year. El método \_\_init\_\_() toma estos parámetros y los asigna a los atributos que se asociarán a las instancias creadas a partir de esta clase. Cuando creemos una nueva instancia de Car, tendremos que especificar una marca, un modelo y un año para nuestra instancia.

Definimos un método llamado get\_descriptive\_name() ❷ que pone los datos de un coche year, make, y model en una sola cadena que describe claramente el coche. Esto nos evitará tener que imprimir individualmente el valor de cada atributo. Para trabajar con los valores de los atributos en este método, utilizamos self.make, self.model, y self.year. Fuera de la clase, creamos una instancia de la clase Car y la asignamos a la variable my\_new\_car ❸. Luego llamamos a get\_descriptive\_name() para mostrar qué clase de coche tenemos:

2024 Audi A4

Para hacer la clase más interesante, añadamos un atributo que cambie con el tiempo. Añadiremos un atributo que almacene el kilometraje total del coche.

### Establecer un valor por defecto para un atributo

Cuando se crea una instancia, se pueden definir atributos sin pasarlos como parámetros. Estos atributos pueden definirse en el método \_\_init\_\_(), donde se les asigna un valor por defecto.

Vamos a añadir un atributo llamado odometer\_reading que comience siempre con el valor 0. También añadiremos un método read\_odometer() que nos ayude a leer el cuentakilómetros de cada coche:

class Car:  
  
 def \_\_init\_\_(self, make, model, year):  
 """Initialize attributes to describe a car."""  
 self.make = make  
 self.model = model  
 self.year = year  
❶ self.odometer\_reading = 0  
  
 def get\_descriptive\_name(self):  
 --snip--  
  
❷ def read\_odometer(self):  
 """Print a statement showing the car's mileage."""  
 print(f"This car has {self.odometer\_reading} miles on it.")  
  
my\_new\_car = Car('audi', 'a4', 2024)  
print(my\_new\_car.get\_descriptive\_name())  
my\_new\_car.read\_odometer()

Esta vez, cuando Python llama al método \_\_init\_\_() para crear una nueva instancia, almacena los valores de marca, modelo y año como atributos, como hizo en el ejemplo anterior. A continuación, Python crea un nuevo atributo llamado odometer\_reading y establece su valor inicial en 0 ❶. También tenemos un nuevo método llamado read\_odometer() ❷ que facilita la lectura del kilometraje de un coche.

Nuestro coche empieza con un kilometraje de 0:

2024 Audi A4  
This car has 0 miles on it.

No se venden muchos coches con exactamente 0 kilómetros en el cuentakilómetros, así que necesitamos una forma de cambiar el valor de este atributo.

### Modificar los valores de los atributos

Puedes cambiar el valor de un atributo de tres formas: puedes cambiar el valor directamente a través de una instancia, establecer el valor a través de un método o incrementar el valor (añadirle una determinada cantidad) a través de un método. Veamos cada uno de estos enfoques.

#### Modificar directamente el valor de un atributo

La forma más sencilla de modificar el valor de un atributo es acceder a él directamente a través de una instancia. Aquí fijamos directamente la lectura del cuentakilómetros en 23:

class Car:  
 --snip--  
  
my\_new\_car = Car('audi', 'a4', 2024)  
print(my\_new\_car.get\_descriptive\_name())  
  
my\_new\_car.odometer\_reading = 23  
my\_new\_car.read\_odometer()

Utilizamos la notación con puntos para acceder al atributo odometer\_reading del coche, y establecemos su valor directamente. Esta línea le dice a Python que tome la instancia my\_new\_car, busque el atributo odometer\_reading asociado a ella y establezca el valor de ese atributo en 23:

2024 Audi A4  
This car has 23 miles on it.

A veces querrás acceder directamente a los atributos de esta forma, pero otras veces querrás escribir un método que actualice el valor por ti.

#### Modificar el valor de un atributo mediante un método

Puede ser útil disponer de métodos que actualicen determinados atributos por ti. En lugar de acceder directamente al atributo, pasas el nuevo valor a un método que se encarga internamente de la actualización.

Aquí tienes un ejemplo de un método llamado update\_odometer():

class Car:  
 --snip--  
  
 def update\_odometer(self, mileage):  
 """Set the odometer reading to the given value."""  
 self.odometer\_reading = mileage  
  
my\_new\_car = Car('audi', 'a4', 2024)  
print(my\_new\_car.get\_descriptive\_name())  
  
❶ my\_new\_car.update\_odometer(23)  
my\_new\_car.read\_odometer()

La única modificación de Car es la adición de update\_odometer(). Este método toma un valor de kilometraje y lo asigna a self.odometer\_reading. Utilizando la instancia my\_new\_car, llamamos a update\_odometer() con 23 como argumento ❶. Esto establece la lectura del cuentakilómetros en 23, y read\_odometer() imprime la lectura:

2024 Audi A4  
This car has 23 miles on it.

Podemos ampliar el método update\_odometer() para que realice un trabajo adicional cada vez que se modifique la lectura del cuentakilómetros. Añadamos un poco de lógica para asegurarnos de que nadie intenta hacer retroceder la lectura del cuentakilómetros:

class Car:  
 --snip--  
  
 def update\_odometer(self, mileage):  
 """  
 Set the odometer reading to the given value.  
 Reject the change if it attempts to roll the odometer back.  
 """  
❶ if mileage >= self.odometer\_reading:  
 self.odometer\_reading = mileage  
 else:  
❷ print("You can't roll back an odometer!")

Ahora update\_odometer() comprueba que la nueva lectura tiene sentido antes de modificar el atributo. Si el valor proporcionado para mileage es mayor o igual que el kilometraje existente, self.odometer\_reading, puedes actualizar la lectura del cuentakilómetros al nuevo kilometraje ❶. Si el nuevo kilometraje es menor que el kilometraje existente, recibirás un aviso de que no puedes retroceder un cuentakilómetros ❷.

#### Incrementar el valor de un atributo mediante un método

A veces querrás incrementar el valor de un atributo en una cantidad determinada, en lugar de establecer un valor completamente nuevo. Supongamos que compramos un coche usado y le hacemos 160 km entre el momento en que lo compramos y el momento en que lo matriculamos. Aquí tenemos un método que nos permite pasar esta cantidad incremental y añadir ese valor a la lectura del cuentakilómetros:

class Car:  
 --snip--  
  
 def update\_odometer(self, mileage):  
 --snip--  
  
 def increment\_odometer(self, miles):  
 """Add the given amount to the odometer reading."""  
 self.odometer\_reading += miles  
  
❶ my\_used\_car = Car('subaru', 'outback', 2019)  
print(my\_used\_car.get\_descriptive\_name())  
  
❷ my\_used\_car.update\_odometer(23\_500)  
my\_used\_car.read\_odometer()  
  
my\_used\_car.increment\_odometer(100)  
my\_used\_car.read\_odometer()

El nuevo método increment\_odometer() toma un número de kilómetros, y añade este valor a self.odometer\_reading. Primero, creamos un coche usado, my\_used\_car ❶. Ponemos su cuentakilómetros en 23.500 llamando a update\_odometer() y pasándole 23\_500 ❷. Por último, llamamos a increment\_odometer() y le pasamos 100 para añadir los 160 km que recorrimos entre la compra del coche y su matriculación:

2019 Subaru Outback  
This car has 23500 miles on it.  
This car has 23600 miles on it.

Puedes modificar este método para que rechace incrementos negativos, de modo que nadie utilice también esta función para hacer retroceder un cuentakilómetros.

## Nota

Puedes utilizar métodos como éste para controlar cómo los usuarios de tu programa actualizan valores como la lectura de un cuentakilómetros, pero cualquiera con acceso al programa puede establecer la lectura del cuentakilómetros en cualquier valor accediendo directamente al atributo. Una seguridad eficaz requiere una atención extrema a los detalles, además de comprobaciones básicas como las que se muestran aquí.

## Pruébalo tú mismo

9-4. Número servido: Empieza con tu programa del Ejercicio 9-1(página 162). Añade un atributo llamado number\_served con un valor por defecto de 0. Crea una instancia llamada restaurant a partir de esta clase. Imprime el número de clientes a los que ha servido el restaurante, y luego cambia este valor e imprímelo de nuevo.

Añade un método llamado set\_number\_served() que te permita establecer el número de clientes a los que se ha servido. Llama a este método con un nuevo número e imprime de nuevo el valor.

Añade un método llamado increment\_number\_served() que te permita incrementar el número de clientes que han sido atendidos. Llama a este método con cualquier número que quieras que pueda representar cuántos clientes han sido atendidos en, digamos, un día de negocio.

9-5. Intentos de inicio de sesión: Añade un atributo llamado login\_attempts a tu clase User del Ejercicio 9-3(página 162). Escribe un método llamado increment\_login\_attempts() que incremente el valor de login\_attempts en 1. Escribe otro método llamado reset\_login\_attempts() que restablezca el valor de login\_attempts a 0.

Crea una instancia de la clase User y llama varias veces a increment\_login\_attempts(). Imprime el valor de login\_attempts para asegurarte de que se ha incrementado correctamente, y luego llama a reset\_login\_attempts(). Vuelve a imprimir login\_attempts para asegurarte de que se ha puesto a 0.

## Herencia

No siempre tienes que empezar de cero al escribir una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que escribiste, puedes utilizar *inheritance*. Cuando una clase *inherits* de otra, adopta los atributos y métodos de la primera clase. La clase original se llama *parent class*, y la nueva clase es *child class*. La clase hija puede heredar alguno o todos los atributos y métodos de su clase padre, pero también es libre de definir nuevos atributos y métodos propios.

### El método \_\_init\_\_() de una clase hija

Cuando escribas una nueva clase basada en una clase existente, a menudo querrás llamar al método \_\_init\_\_() de la clase padre. Esto inicializará los atributos definidos en el método \_\_init\_\_() de la clase padre y los pondrá a disposición de la clase hija.

Por ejemplo, modelemos un coche eléctrico. Un coche eléctrico es sólo una clase específica de coche, así que podemos basar nuestra nueva clase ElectricCar en la clase Car que escribimos antes. Así sólo tendremos que escribir código para los atributos y comportamientos específicos de los coches eléctricos.

Empecemos haciendo una versión sencilla de la clase ElectricCar, que hace todo lo que hace la clase Car:

**electric\_car.py**

❶ class Car:  
 """A simple attempt to represent a car."""  
  
 def \_\_init\_\_(self, make, model, year):  
 """Initialize attributes to describe a car.""  
 self.make = make  
 self.model = model  
 self.year = year  
 self.odometer\_reading = 0  
  
 def get\_descriptive\_name(self):  
 """Return a neatly formatted descriptive name."""  
 long\_name = f"{self.year} {self.make} {self.model}"  
 return long\_name.title()  
  
 def read\_odometer(self):  
 """Print a statement showing the car's mileage."""  
 print(f"This car has {self.odometer\_reading} miles on it.")  
  
 def update\_odometer(self, mileage):  
 """Set the odometer reading to the given value."""  
 if mileage >= self.odometer\_reading:  
 self.odometer\_reading = mileage  
 else:  
 print("You can't roll back an odometer!")  
  
 def increment\_odometer(self, miles):  
 """Add the given amount to the odometer reading."""  
 self.odometer\_reading += miles  
  
❷ class ElectricCar(Car):  
 """Represent aspects of a car, specific to electric vehicles."""  
  
❸ def \_\_init\_\_(self, make, model, year):  
 """Initialize attributes of the parent class."""  
❹ super().\_\_init\_\_(make, model, year)  
  
  
❺ my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())

Empezaremos con Car ❶. Cuando creas una clase hija, la clase padre debe formar parte del archivo actual y debe aparecer antes que la clase hija en el archivo. A continuación definimos la clase hija, ElectricCar ❷. El nombre de la clase padre debe incluirse entre paréntesis en la definición de una clase hija. El método \_\_init\_\_() recoge la información necesaria para crear una instancia de Car ❸.

La función super() ❹ es una función especial que te permite llamar a un método de la clase padre. Esta línea indica a Python que llame al método \_\_init\_\_() desde Car, lo que proporciona a una instancia de ElectricCar todos los atributos definidos en ese método. El nombre *super* proviene de la convención de llamar a la clase padre *superclass* y a la clase hija *subclass*.

Comprobamos si la herencia funciona correctamente intentando crear un coche eléctrico con el mismo tipo de información que proporcionaríamos al crear un coche normal. Creamos una instancia de la clase ElectricCar y la asignamos a my\_leaf ❺. Esta línea llama al método \_\_init\_\_() definido en ElectricCar, que a su vez le dice a Python que llame al método \_\_init\_\_() definido en la clase padre Car. Proporcionamos los argumentos 'nissan', 'leaf', y 2024.

Aparte de \_\_init\_\_(), aún no hay atributos ni métodos particulares de un coche eléctrico. En este momento sólo nos estamos asegurando de que el coche eléctrico tiene los comportamientos Car adecuados:

2024 Nissan Leaf

La instancia de ElectricCar funciona igual que una instancia de Car, así que ahora podemos empezar a definir atributos y métodos específicos de los coches eléctricos.

### Definir atributos y métodos para la clase hija

Una vez que tengas una clase hija que hereda de una clase padre, puedes añadir los atributos y métodos nuevos que sean necesarios para diferenciar la clase hija de la clase padre.

Añadamos un atributo específico de los coches eléctricos (una batería, por ejemplo) y un método para informar sobre este atributo. Almacenaremos el tamaño de la batería y escribiremos un método que imprima una descripción de la batería:

class Car:  
 --snip--  
  
class ElectricCar(Car):  
 """Represent aspects of a car, specific to electric vehicles."""  
  
 def \_\_init\_\_(self, make, model, year):  
 """  
 Initialize attributes of the parent class.  
 Then initialize attributes specific to an electric car.  
 """  
 super().\_\_init\_\_(make, model, year)  
❶ self.battery\_size = 40  
  
❷ def describe\_battery(self):  
 """Print a statement describing the battery size."""  
 print(f"This car has a {self.battery\_size}-kWh battery.")  
  
my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())  
my\_leaf.describe\_battery()

Añadimos un nuevo atributo self.battery\_size y establecemos su valor inicial en 40 ❶. Este atributo se asociará a todas las instancias creadas a partir de la clase ElectricCar, pero no se asociará a ninguna instancia de Car. También añadimos un método llamado describe\_battery() que imprime información sobre la pila ❷. Cuando llamamos a este método, obtenemos una descripción claramente específica de un coche eléctrico:

2024 Nissan Leaf  
This car has a 40-kWh battery.

No hay límite en cuanto a la especialización de la clase ElectricCar. Puedes añadir tantos atributos y métodos como necesites para modelar un coche eléctrico con el grado de precisión que necesites. Un atributo o método que podría pertenecer a cualquier coche, en lugar de uno específico de un coche eléctrico, debería añadirse a la clase Car en lugar de a la clase ElectricCar. Así, cualquiera que utilice la clase Car tendrá también disponible esa funcionalidad, y la clase ElectricCar sólo contendrá código para la información y el comportamiento específicos de los vehículos eléctricos.

### Reemplazar métodos de la clase padre

Puedes anular cualquier método de la clase padre que no se ajuste a lo que intentas modelar con la clase hija. Para ello, define un método en la clase hija con el mismo nombre que el método que quieres anular en la clase padre. Python ignorará el método de la clase padre y sólo prestará atención al método que definas en la clase hija.

Supongamos que la clase Car tiene un método llamado fill\_gas\_tank(). Este método no tiene sentido para un vehículo totalmente eléctrico, así que tal vez quieras anular este método. He aquí una forma de hacerlo:

class ElectricCar(Car):  
 --snip--  
  
 def fill\_gas\_tank(self):  
 """Electric cars don't have gas tanks."""  
 print("This car doesn't have a gas tank!")

Ahora, si alguien intenta llamar a fill\_gas\_tank() con un coche eléctrico, Python ignorará el método fill\_gas\_tank() en Car y ejecutará este código en su lugar. Cuando utilizas la herencia, puedes hacer que tus clases hijas retengan lo que necesites y anulen lo que no necesites de la clase padre.

### Instancias como atributos

Al modelar algo del mundo real en código, puedes encontrarte con que vas añadiendo cada vez más detalles a una clase. Te darás cuenta de que tienes una lista cada vez mayor de atributos y métodos y que tus archivos son cada vez más largos. En estas situaciones, puedes reconocer que parte de una clase puede escribirse como una clase independiente. Puedes dividir tu clase grande en clases más pequeñas que funcionen juntas; este enfoque se denomina *composition*.

Por ejemplo, si seguimos añadiendo detalles a la clase ElectricCar, podríamos darnos cuenta de que estamos añadiendo muchos atributos y métodos específicos de la batería del coche. Cuando veamos que esto ocurre, podemos detenernos y trasladar esos atributos y métodos a una clase independiente llamada Battery. Entonces podremos utilizar una instancia de Battery como atributo en la clase ElectricCar:

class Car:  
 --snip--  
  
class Battery:  
 """A simple attempt to model a battery for an electric car."""  
  
❶ def \_\_init\_\_(self, battery\_size=40):  
 """Initialize the battery's attributes."""  
 self.battery\_size = battery\_size  
  
❷ def describe\_battery(self):  
 """Print a statement describing the battery size."""  
 print(f"This car has a {self.battery\_size}-kWh battery.")  
  
  
class ElectricCar(Car):  
 """Represent aspects of a car, specific to electric vehicles."""  
  
 def \_\_init\_\_(self, make, model, year):  
 """  
 Initialize attributes of the parent class.  
 Then initialize attributes specific to an electric car.  
 """  
 super().\_\_init\_\_(make, model, year)  
❸ self.battery = Battery()  
  
my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())  
my\_leaf.battery.describe\_battery()

Definimos una nueva clase llamada Battery que no hereda de ninguna otra clase. El método \_\_init\_\_() ❶ tiene un parámetro, battery\_size, además de self. Se trata de un parámetro opcional que establece el tamaño de la pila en 40 si no se proporciona ningún valor. También se ha trasladado a esta clase el método describe\_battery() ❷.

En la clase ElectricCar, añadimos ahora un atributo llamado self.battery ❸. Esta línea le dice a Python que cree una nueva instancia de Battery (con un tamaño por defecto de 40, porque no estamos especificando un valor) y asigne esa instancia al atributo self.battery. Esto ocurrirá cada vez que se llame al método \_\_init\_\_(); cualquier instancia de ElectricCar tendrá ahora una instancia de Battery creada automáticamente.

Creamos un coche eléctrico y lo asignamos a la variable my\_leaf. Cuando queramos describir la batería, tendremos que trabajar a través del atributo battery del coche:

my\_leaf.battery.describe\_battery()

Esta línea le dice a Python que busque la instancia my\_leaf, encuentre su atributo battery y llame al método describe\_battery() que está asociado a la instancia Battery asignada al atributo.

El resultado es idéntico al que vimos anteriormente:

2024 Nissan Leaf  
This car has a 40-kWh battery.

Esto parece mucho trabajo extra, pero ahora podemos describir la pila con tanto detalle como queramos sin saturar la clase ElectricCar. Añadamos otro método a Battery que informe de la autonomía del coche en función del tamaño de la batería:

class Car:  
 --snip--  
  
class Battery:  
 --snip--  
  
 def get\_range(self):  
 """Print a statement about the range this battery provides."""  
 if self.battery\_size == 40:  
 range = 150  
 elif self.battery\_size == 65:  
 range = 225  
  
 print(f"This car can go about {range} miles on a full charge.")  
  
class ElectricCar(Car):  
 --snip--  
  
my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())  
my\_leaf.battery.describe\_battery()  
❶ my\_leaf.battery.get\_range()

El nuevo método get\_range() realiza algunos análisis sencillos. Si la capacidad de la batería es de 40 kWh, get\_range() establece la autonomía en 240 km, y si la capacidad es de 65 kWh, establece la autonomía en 225 km. A continuación, informa de este valor. Cuando queramos utilizar este método, tendremos que llamarlo de nuevo a través del atributo battery ❶ del coche .

La salida nos indica la autonomía del coche en función del tamaño de su batería:

2024 Nissan Leaf  
This car has a 40-kWh battery.  
This car can go about 150 miles on a full charge.

### Modelar objetos del mundo real

Cuando empieces a modelar cosas más complicadas, como coches eléctricos, te plantearás cuestiones interesantes. ¿La autonomía de un coche eléctrico es una propiedad de la batería o del coche? Si sólo estamos describiendo un coche, probablemente esté bien mantener la asociación del método get\_range() con la clase Battery. Pero si estamos describiendo toda la línea de coches de un fabricante, probablemente queramos trasladar get\_range() a la clase ElectricCar. El método get\_range() seguiría comprobando el tamaño de la batería antes de determinar la autonomía, pero informaría de una autonomía específica para la clase de coche a la que está asociado. Alternativamente, podríamos mantener la asociación del método get\_range() con la batería, pero pasándole un parámetro como car\_model. El método get\_range() informaría entonces de un rango basado en el tamaño de la batería y el modelo de coche.

Esto te lleva a un punto interesante en tu crecimiento como programador. Cuando te enfrentas a preguntas como éstas, estás pensando a un nivel lógico superior en lugar de a un nivel centrado en la sintaxis. No estás pensando en Python, sino en cómo representar el mundo real en código. Cuando llegues a este punto, te darás cuenta de que a menudo no hay enfoques correctos o incorrectos para modelar situaciones del mundo real. Algunos enfoques son más eficientes que otros, pero se necesita práctica para encontrar las representaciones más eficientes. Si tu código funciona como quieres, ¡lo estás haciendo bien! No te desanimes si descubres que estás desmontando tus clases y reescribiéndolas varias veces utilizando distintos enfoques. En la búsqueda de un código preciso y eficiente, todo el mundo pasa por este proceso.

## Pruébalo tú mismo

9-6. Puesto de helados: Un puesto de helados es una clase específica de restaurante. Escribe una clase llamada IceCreamStand que herede de la clase Restaurant que escribiste en el Ejercicio 9-1(página 162) o en el Ejercicio 9-4(página 166). Cualquiera de las dos versiones de la clase funcionará; simplemente elige la que más te guste. Añade un atributo llamado flavors que almacene una lista de sabores de helado. Escribe un método que muestre estos sabores. Crea una instancia de IceCreamStand, y llama a este método.

9-7. Administrador: Un administrador es una clase especial de usuario. Escribe una clase llamada Admin que herede de la clase User que escribiste en el Ejercicio 9-3(página 162) o en el Ejercicio 9-5(página 167). Añade un atributo, privileges, que almacene una lista de cadenas como "can add post", "can delete post", "can ban user", etc. Escribe un método llamado show\_privileges() que enumere el conjunto de privilegios del administrador. Crea una instancia de Admin, y llama a tu método.

9-8. Privilegios: Escribe una clase Privileges separada. La clase debe tener un atributo, privileges, que almacene una lista de cadenas como la descrita en el Ejercicio 9-7. Traslada el método show\_privileges() a esta clase. Crea una instancia de Privileges como atributo en la clase Admin. Crea una nueva instancia de Admin y utiliza su método para mostrar sus privilegios.

9-9. Actualización de la Batería: Utiliza la versión final de *electric\_car.py* de esta sección. Añade un método a la clase Battery llamado upgrade\_battery(). Este método debería comprobar el tamaño de la batería y establecer la capacidad en 65 si aún no lo está. Crea un coche eléctrico con un tamaño de batería por defecto, llama a get\_range() una vez y, a continuación, llama a get\_range() una segunda vez después de actualizar la batería. Deberías ver un aumento de la autonomía del coche.

## Importar clases

A medida que añades más funcionalidad a tus clases, tus archivos pueden hacerse largos, incluso cuando utilizas la herencia y la composición adecuadamente. De acuerdo con la filosofía general de Python, querrás mantener tus archivos lo más despejados posible. Para ayudarte, Python te permite almacenar clases en módulos y luego importar las clases que necesites en tu programa principal.

### Importar una sola clase

Creemos un módulo que contenga sólo la clase Car. Esto plantea un sutil problema de nomenclatura: ya tenemos un archivo llamado *car.py* en este capítulo, pero este módulo debería llamarse *car.py* porque contiene código que representa un coche. Resolveremos este problema de nomenclatura almacenando la clase Car en un módulo llamado *car.py*, en sustitución del archivo *car.py* que utilizábamos anteriormente. A partir de ahora, cualquier programa que utilice este módulo necesitará un nombre de archivo más específico, como *my\_car.py*. Aquí tienes *car.py* con sólo el código de la clase Car:

**car.py**

❶ """A class that can be used to represent a car."""  
  
class Car:  
 """A simple attempt to represent a car."""  
  
 def \_\_init\_\_(self, make, model, year):  
 """Initialize attributes to describe a car."""  
 self.make = make  
 self.model = model  
 self.year = year  
 self.odometer\_reading = 0  
  
 def get\_descriptive\_name(self):  
 """Return a neatly formatted descriptive name."""  
 long\_name = f"{self.year} {self.make} {self.model}"  
 return long\_name.title()  
  
 def read\_odometer(self):  
 """Print a statement showing the car's mileage."""  
 print(f"This car has {self.odometer\_reading} miles on it.")  
  
 def update\_odometer(self, mileage):  
 """  
 Set the odometer reading to the given value.  
 Reject the change if it attempts to roll the odometer back.  
 """  
 if mileage >= self.odometer\_reading:  
 self.odometer\_reading = mileage  
 else:  
 print("You can't roll back an odometer!")  
  
 def increment\_odometer(self, miles):  
 """Add the given amount to the odometer reading."""  
 self.odometer\_reading += miles

Incluimos un docstring a nivel de módulo que describe brevemente el contenido de este módulo ❶. Debes escribir un docstring para cada módulo que crees.

Ahora creamos un archivo independiente llamado *my\_car.py*. Este archivo importará la clase Car y luego creará una instancia a partir de esa clase:

**my\_car.py**

❶ from car import Car  
  
my\_new\_car = Car('audi', 'a4', 2024)  
print(my\_new\_car.get\_descriptive\_name())  
  
my\_new\_car.odometer\_reading = 23  
my\_new\_car.read\_odometer()

La sentencia import ❶ indica a Python que abra el módulo car e importe la clase Car. Ahora podemos utilizar la clase Car como si estuviera definida en este archivo. El resultado es el mismo que vimos antes:

2024 Audi A4  
This car has 23 miles on it.

Importar clases es una forma eficaz de programar. Imagina lo largo que sería este archivo de programa si se incluyera toda la clase Car. En cambio, si trasladas la clase a un módulo e importas el módulo, sigues obteniendo toda la misma funcionalidad, pero mantienes tu archivo de programa principal limpio y fácil de leer. También almacenas la mayor parte de la lógica en archivos separados; una vez que tus clases funcionen como quieres, puedes dejar esos archivos tranquilos y centrarte en la lógica de alto nivel de tu programa principal.

### Almacenar varias clases en un módulo

Puedes almacenar tantas clases como necesites en un solo módulo, aunque cada clase de un módulo debe estar relacionada de alguna manera. Las clases Battery y ElectricCar ayudan a representar coches, así que añadámoslas al módulo *car.py*.

**car.py**

"""A set of classes used to represent gas and electric cars."""  
  
class Car:  
 --snip--  
  
class Battery:  
 """A simple attempt to model a battery for an electric car."""  
  
 def \_\_init\_\_(self, battery\_size=40):  
 """Initialize the battery's attributes."""  
 self.battery\_size = battery\_size  
  
 def describe\_battery(self):  
 """Print a statement describing the battery size."""  
 print(f"This car has a {self.battery\_size}-kWh battery.")  
  
 def get\_range(self):  
 """Print a statement about the range this battery provides."""  
 if self.battery\_size == 40:  
 range = 150  
 elif self.battery\_size == 65:  
 range = 225  
  
 print(f"This car can go about {range} miles on a full charge.")  
  
class ElectricCar(Car):  
 """Models aspects of a car, specific to electric vehicles."""  
  
 def \_\_init\_\_(self, make, model, year):  
 """  
 Initialize attributes of the parent class.  
 Then initialize attributes specific to an electric car.  
 """  
 super().\_\_init\_\_(make, model, year)  
 self.battery = Battery()

Ahora podemos crear un nuevo archivo llamado *my\_electric\_car.py*, importar la clase ElectricCar y crear un coche eléctrico:

**my\_electric\_car.py**

from car import ElectricCar  
  
my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())  
my\_leaf.battery.describe\_battery()  
my\_leaf.battery.get\_range()

Esto tiene el mismo resultado que vimos antes, aunque la mayor parte de la lógica esté oculta en un módulo:

2024 Nissan Leaf  
This car has a 40-kWh battery.  
This car can go about 150 miles on a full charge.

### Importar varias clases de un módulo

Puedes importar tantas clases como necesites en un archivo de programa. Si queremos hacer un coche normal y un coche eléctrico en el mismo archivo, necesitamos importar ambas clases, Car y ElectricCar:

**my\_cars.py**

❶ from car import Car, ElectricCar  
  
❷ my\_mustang = Car('ford', 'mustang', 2024)  
print(my\_mustang.get\_descriptive\_name())  
❸ my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())

Puedes importar varias clases de un módulo separando cada clase con una coma ❶. Una vez que hayas importado las clases necesarias, puedes crear tantas instancias de cada clase como necesites.

En este ejemplo hacemos un Ford Mustang ❷ de gasolina y luego un Nissan Leaf ❸ eléctrico :

2024 Ford Mustang  
2024 Nissan Leaf

### Importar un módulo entero

También puedes importar un módulo entero y luego acceder a las clases que necesites utilizando la notación de puntos. Este método es sencillo y da como resultado un código fácil de leer. Como cada llamada que crea una instancia de una clase incluye el nombre del módulo, no tendrás conflictos de nombres con los nombres utilizados en el archivo actual.

Este es el aspecto que tiene importar todo el módulo car y luego crear un coche normal y un coche eléctrico:

**my\_cars.py**

❶ import car  
  
❷ my\_mustang = car.Car('ford', 'mustang', 2024)  
print(my\_mustang.get\_descriptive\_name())  
  
❸ my\_leaf = car.ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())

Primero importamos todo el módulo car ❶. A continuación, accedemos a las clases que necesitamos mediante la sintaxis nombre\_módulo.Nombre\_clase. Creamos de nuevo un Ford Mustang ❷, y un Nissan Leaf ❸.

### Importar todas las clases de un módulo

Puedes importar todas las clases de un módulo utilizando la siguiente sintaxis:

from module\_name import \*

Este método no es recomendable por dos razones. En primer lugar, es útil poder leer las declaraciones import al principio de un archivo y tener una idea clara de qué clases utiliza un programa. Con este enfoque no está claro qué clases estás utilizando del módulo. Este enfoque también puede llevar a confusiones con los nombres en el archivo. Si importas accidentalmente una clase con el mismo nombre que otra en el archivo de tu programa, puedes crear errores difíciles de diagnosticar. Muestro esto aquí porque, aunque no es un enfoque recomendado, es probable que lo veas en el código de otras personas en algún momento.

Si necesitas importar muchas clases de un módulo, es mejor que importes todo el módulo y utilices la sintaxis nombre\_módulo.Nombre\_clase. No verás todas las clases utilizadas en la parte superior del archivo, pero verás claramente dónde se utiliza el módulo en el programa. También evitarás los posibles conflictos de nombres que pueden surgir cuando importas todas las clases de un módulo.

### Importar un módulo dentro de otro módulo

A veces querrás distribuir tus clases en varios módulos para evitar que un archivo crezca demasiado y para evitar almacenar clases no relacionadas en el mismo módulo. Cuando almacenas tus clases en varios módulos, puedes encontrar que una clase de un módulo depende de una clase de otro módulo. Cuando esto ocurra, puedes importar la clase necesaria en el primer módulo.

Por ejemplo, almacenemos la clase Car en un módulo y las clases ElectricCar y Battery en otro módulo. Crearemos un nuevo módulo llamado *electric\_car.py*-reemplazando el archivo *electric\_car.py* que creamos antes- y copiaremos sólo las clases Battery y ElectricCar en este archivo:

**electric\_car.py**

"""A set of classes that can be used to represent electric cars."""  
  
from car import Car  
  
class Battery:  
 --snip--  
  
class ElectricCar(Car):  
 --snip--

La clase ElectricCar necesita acceder a su clase padre Car, así que importaremos Car directamente al módulo. Si olvidamos esta línea, Python emitirá un error cuando intentemos importar el módulo electric\_car. También tenemos que actualizar el módulo Car para que sólo contenga la clase Car:

**car.py**

"""A class that can be used to represent a car."""  
  
class Car:  
 --snip--

Ahora podemos importar de cada módulo por separado y crear la clase que necesitemos:

**my\_cars.py**

from car import Car  
from electric\_car import ElectricCar  
  
my\_mustang = Car('ford', 'mustang', 2024)  
print(my\_mustang.get\_descriptive\_name())  
  
my\_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my\_leaf.get\_descriptive\_name())

Importamos Car de su módulo, y ElectricCar de su módulo. A continuación, creamos un coche normal y un coche eléctrico. Ambos coches se crean correctamente:

2024 Ford Mustang  
2024 Nissan Leaf

### Utilizar alias

Como viste en el Capítulo 8, los alias pueden ser muy útiles cuando utilizas módulos para organizar el código de tus proyectos. También puedes utilizar alias al importar clases.

Como ejemplo, considera un programa en el que quieras fabricar un montón de coches eléctricos. Podría resultar tedioso escribir (y leer) ElectricCar una y otra vez. Puedes dar a ElectricCar un alias en la sentencia import:

from electric\_car import ElectricCar as EC

Ahora puedes utilizar este alias siempre que quieras fabricar un coche eléctrico:

my\_leaf = EC('nissan', 'leaf', 2024)

También puedes dar un alias a un módulo. Aquí tienes cómo importar todo el módulo electric\_car utilizando un alias:

import electric\_car as ec

Ahora puedes utilizar este alias de módulo con el nombre completo de la clase:

my\_leaf = ec.ElectricCar('nissan', 'leaf', 2024)

### Encontrar tu propio flujo de trabajo

Como puedes ver, Python te da muchas opciones sobre cómo estructurar el código en un proyecto grande. Es importante que conozcas todas estas posibilidades para que puedas determinar las mejores formas de organizar tus proyectos, así como entender los proyectos de otras personas.

Cuando estés empezando, mantén una estructura de código sencilla. Intenta hacerlo todo en un solo archivo y mover tus clases a módulos separados una vez que todo funcione. Si te gusta cómo interactúan los módulos y los archivos, intenta almacenar tus clases en módulos cuando empieces un proyecto. Encuentra un enfoque que te permita escribir código que funcione, y sigue a partir de ahí.

## Pruébalo tú mismo

9-10. Restaurante importado: Utilizando tu última clase Restaurant, almacénala en un módulo. Crea un archivo aparte que importe Restaurant. Crea una instancia de Restaurant y llama a uno de los métodos de Restaurantpara demostrar que la sentencia import funciona correctamente.

9-11. Admin. importado: Empieza con tu trabajo del Ejercicio 9-8(página 173). Guarda las clases User, Privileges, y Admin en un módulo. Crea un archivo separado, crea una instancia Admin y llama a show\_privileges() para mostrar que todo funciona correctamente.

9-12. Módulos múltiples: Almacena la clase User en un módulo, y almacena las clases Privileges y Admin en un módulo aparte. En un archivo separado, crea una instancia de Admin y llama a show\_privileges() para demostrar que todo sigue funcionando correctamente.

## La biblioteca estándar de Python

La *Python standard library* es un conjunto de módulos incluidos en cada instalación de Python. Ahora que tienes una comprensión básica de cómo funcionan las funciones y las clases, puedes empezar a utilizar módulos como éstos que han escrito otros programadores. Puedes utilizar cualquier función o clase de la biblioteca estándar incluyendo una simple sentencia import al principio de tu archivo. Veamos un módulo, random, que puede ser útil para modelar muchas situaciones del mundo real.

Una función interesante del módulo aleatorio es randint(). Esta función toma dos argumentos enteros y devuelve un número entero seleccionado aleatoriamente entre esos números (e incluyéndolos).

He aquí cómo generar un número aleatorio entre 1 y 6:

>>> from random import randint  
>>> randint(1, 6)  
3

Otra función útil es choice(). Esta función toma una lista o tupla y devuelve un elemento elegido al azar:

>>> from random import choice  
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']  
>>> first\_up = choice(players)  
>>> first\_up  
'florence'

El módulo random no debería utilizarse para crear aplicaciones relacionadas con la seguridad, pero funciona bien para muchos proyectos divertidos e interesantes.

## Nota

También puedes descargar módulos de fuentes externas. Verás varios de estos ejemplos en la Parte II, donde necesitaremos módulos externos para completar cada proyecto.

## Pruébalo tú mismo

9-13. Dados: Crea una clase Die con un atributo llamado sides, que tiene un valor por defecto de 6. Escribe un método llamado roll\_die() que imprima un número aleatorio entre 1 y el número de caras que tiene el dado. Haz un dado de 6 caras y lánzalo 10 veces.

Crea un dado de 10 caras y otro de 20 caras. Tira cada dado 10 veces.

9-14. Lotería: Haz una lista o tupla que contenga una serie de 10 números y 5 letras. Selecciona al azar 4 números o letras de la lista e imprime un mensaje diciendo que cualquier boleto que coincida con esos 4 números o letras gana un premio.

9-15. Análisis de la Lotería: Puedes utilizar un bucle para ver lo difícil que puede ser ganar el tipo de lotería que acabas de modelar. Haz una lista o tupla llamada my\_ticket. Escribe un bucle que siga sacando números hasta que gane el boleto. Imprime un mensaje informando de cuántas veces ha tenido que ejecutarse el bucle para darte un boleto ganador.

9-16. Módulo Python de la Semana: Un recurso excelente para explorar la biblioteca estándar de Python es un sitio llamado *Python Module of the Week*. Entra en <https://pymotw.com> y mira la tabla de contenidos. Busca un módulo que te parezca interesante y lee sobre él, quizás empezando por el módulo random.

## Clases de estilo

Merece la pena aclarar algunas cuestiones de estilo relacionadas con las clases, especialmente a medida que tus programas se complican.

Los nombres de las clases deben escribirse en *CamelCase*. Para ello, escribe en mayúscula la primera letra de cada palabra del nombre y no utilices guiones bajos. Los nombres de instancias y módulos deben escribirse en minúsculas, con guiones bajos entre las palabras.

Cada clase debe tener un docstring inmediatamente después de la definición de la clase. La docstring debe ser una breve descripción de lo que hace la clase, y debes seguir las mismas convenciones de formato que utilizaste para escribir docstrings en funciones. Cada módulo debe tener también una docstring que describa para qué pueden utilizarse las clases de un módulo.

Puedes utilizar líneas en blanco para organizar el código, pero no las utilices en exceso. Dentro de una clase puedes utilizar una línea en blanco entre métodos, y dentro de un módulo puedes utilizar dos líneas en blanco para separar clases.

Si necesitas importar un módulo de la biblioteca estándar y un módulo escrito por ti, coloca primero la sentencia import del módulo de la biblioteca estándar. A continuación, añade una línea en blanco y la sentencia import del módulo que has escrito. En programas con varias declaraciones de importación, esta convención facilita ver de dónde proceden los distintos módulos utilizados en el programa.

## Resumen

En este capítulo has aprendido a escribir tus propias clases. Aprendiste a almacenar información en una clase utilizando atributos y a escribir métodos que den a tus clases el comportamiento que necesitan. Aprendiste a escribir métodos \_\_init\_\_() que crean instancias a partir de tus clases con exactamente los atributos que deseas. Viste cómo modificar los atributos de una instancia directamente y a través de métodos. Aprendiste que la herencia puede simplificar la creación de clases relacionadas entre sí, y aprendiste a utilizar instancias de una clase como atributos en otra clase para mantener la sencillez de cada clase.

Viste cómo almacenar las clases en módulos e importar las clases que necesitas en los archivos donde se utilizarán puede mantener organizados tus proyectos. Empezaste a aprender sobre la biblioteca estándar de Python, y viste un ejemplo basado en el módulo random. Por último, aprendiste a dar estilo a tus clases utilizando las convenciones de Python.

En el Capítulo 10, aprenderás a trabajar con archivos para poder guardar el trabajo que has hecho en un programa y el trabajo que has permitido hacer a los usuarios. También aprenderás sobre *exceptions*, una clase especial de Python diseñada para ayudarte a responder a los errores cuando surjan.