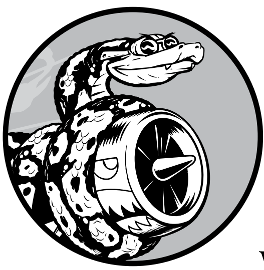
9

CLASSES



Object-oriented programming is one of the most effective approaches to writing software. In object-oriented programming you write *classes* that represent real-world things and situations, and you create *objects* based on these classes. When you write a class, you define the general behavior that a whole category of objects can have.

When you create individual objects from the class, each object is automatically equipped with the general behavior; you can then give each object whatever unique traits you desire. You'll be amazed how well real-world situations can be modeled with object-oriented programming.

Making an object from a class is called *instantiation*, and you work with *instances* of a class. In this chapter you'll write classes and create instances of those classes. You'll specify the kind of information that can be stored in instances, and you'll define actions that can be taken with these instances. You'll also write classes that extend the functionality of existing classes, so similar classes can share code efficiently. You'll store your classes in modules and import classes written by other programmers into your own program files.

Understanding object-oriented programming will help you see the world as a programmer does. It'll help you really know your code, not just what's happening line by line, but also the bigger concepts behind it. Knowing the logic behind classes will train you to think logically so you can write programs that effectively address almost any problem you encounter.

Classes also make life easier for you and the other programmers you'll need to work with as you take on increasingly complex challenges. When you and other programmers write code based on the same kind of logic, you'll be able to understand each other's work. Your programs will make sense to many collaborators, allowing everyone to accomplish more.

Creating and Using a Class

You can model almost anything using classes. Let's start by writing a simple class, `Dog`, that represents a dog—not one dog in particular, but any dog. What do we know about most pet dogs? Well, they all have a name and age. We also know that most dogs sit and roll over. Those two pieces of information (name and age) and those two behaviors (sit and roll over) will go in our `Dog` class because they're common to most dogs. This class will tell Python how to make an object representing a dog. After our class is written, we'll use it to make individual instances, each of which represents one specific dog.

Creating the Dog Class

Each instance created from the `Dog` class will store a `name` and an `age`, and we'll give each dog the ability to `sit()` and `roll_over()`:

dog.py

```
❶ class Dog():
❷     """A simple attempt to model a dog."""

❸     def __init__(self, name, age):
          """Initialize name and age attributes."""
❹         self.name = name
          self.age = age

❺     def sit(self):
          """Simulate a dog sitting in response to a command."""
          print(self.name.title() + " is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(self.name.title() + " rolled over!")
```

There's a lot to notice here, but don't worry. You'll see this structure throughout this chapter and have lots of time to get used to it. At ❶ we define a class called `Dog`. By convention, capitalized names refer to classes in Python. The parentheses in the class definition are empty because we're creating this class from scratch. At ❷ we write a docstring describing what this class does.

The __init__() Method

A function that's part of a class is a *method*. Everything you learned about functions applies to methods as well; the only practical difference for now is the way we'll call methods. The `__init__()` method at ❸ is a special method Python runs automatically whenever we create a new instance based on the `Dog` class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names.

We define the `__init__()` method to have three parameters: `self`, `name`, and `age`. The `self` parameter is required in the method definition, and it must come first before the other parameters. It must be included in the definition because when Python calls this `__init__()` method later (to create an instance of `Dog`), the method call will automatically pass the `self` argument. Every method call associated with a class automatically passes `self`, which is a reference to the instance itself; it gives the individual instance access to the attributes and methods in the class. When we make an instance of `Dog`, Python will call the `__init__()` method from the `Dog` class. We'll pass `Dog()` a name and an age as arguments; `self` is passed automatically, so we don't need to pass it. Whenever we want to make an instance from the `Dog` class, we'll provide values for only the last two parameters, `name` and `age`.

The two variables defined at ❹ each have the prefix `self`. Any variable prefixed with `self` is available to every method in the class, and we'll also be able to access these variables through any instance created from the class. `self.name = name` takes the value stored in the parameter `name` and stores it in the variable `name`, which is then attached to the instance being created. The same process happens with `self.age = age`. Variables that are accessible through instances like this are called *attributes*.

The `Dog` class has two other methods defined: `sit()` and `roll_over()` ❺. Because these methods don't need additional information like a name or age, we just define them to have one parameter, `self`. The instances we create later will have access to these methods. In other words, they'll be able to sit and roll over. For now, `sit()` and `roll_over()` don't do much. They simply print a message saying the dog is sitting or rolling over. But the concept can be extended to realistic situations: if this class were part of an actual computer game, these methods would contain code to make an animated dog sit and roll over. If this class was written to control a robot, these methods would direct movements that cause a dog robot to sit and roll over.

Creating Classes in Python 2.7

When you create a class in Python 2.7, you need to make one minor change. You include the term `object` in parentheses when you create a class:

```
class ClassName(object):
    --snip--
```

This makes Python 2.7 classes behave more like Python 3 classes, which makes your work easier overall.

The `Dog` class would be defined like this in Python 2.7:

```
class Dog(object):
    --snip--
```

Making an Instance from a Class

Think of a class as a set of instructions for how to make an instance. The class `Dog` is a set of instructions that tells Python how to make individual instances representing specific dogs.

Let's make an instance representing a specific dog:

```
class Dog():
    --snip--

❶ my_dog = Dog('willie', 6)

❷ print("My dog's name is " + my_dog.name.title() + ".")
❸ print("My dog is " + str(my_dog.age) + " years old.")
```

The `Dog` class we're using here is the one we just wrote in the previous example. At ❶ we tell Python to create a dog whose name is `'willie'` and whose age is `6`. When Python reads this line, it calls the `__init__()` method in `Dog` with the arguments `'willie'` and `6`. The `__init__()` method creates an instance representing this particular dog and sets the `name` and `age` attributes using the values we provided. The `__init__()` method has no explicit `return` statement, but Python automatically returns an instance representing this dog. We store that instance in the variable `my_dog`. The naming convention is helpful here: we can usually assume that a capitalized name like `Dog` refers to a class, and a lowercase name like `my_dog` refers to a single instance created from a class.

Accessing Attributes

To access the attributes of an instance, you use dot notation. At ❷ we access the value of `my_dog`'s attribute `name` by writing:

```
my_dog.name
```

Dot notation is used often in Python. This syntax demonstrates how Python finds an attribute's value. Here Python looks at the instance `my_dog` and then finds the attribute name associated with my_dog. This is the same attribute referred to as self.name in the class Dog. At ❸ we use the same approach to work with the attribute age. In our first print statement, my_dog.name.title() makes 'willie', the value of my_dog's name attribute, start with a capital letter. In the second print statement, str(my_dog.age) converts 6, the value of my_dog's age attribute, to a string.

The output is a summary of what we know about my_dog:

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

Calling Methods

After we create an instance from the class Dog, we can use dot notation to call any method defined in Dog. Let's make our dog sit and roll over:

```
class Dog():
    --snip--

my_dog = Dog('willie', 6)
my_dog.sit()
my_dog.roll_over()
```

To call a method, give the name of the instance (in this case, my_dog) and the method you want to call, separated by a dot. When Python reads my_dog.sit(), it looks for the method sit() in the class Dog and runs that code. Python interprets the line my_dog.roll_over() in the same way.

Now Willie does what we tell him to:

```
Willie is now sitting.
Willie rolled over!
```

This syntax is quite useful. When attributes and methods have been given appropriately descriptive names like name, age, sit(), and roll_over(), we can easily infer what a block of code, even one we've never seen before, is supposed to do.

Creating Multiple Instances

You can create as many instances from a class as you need. Let's create a second dog called your_dog:

```
class Dog():
    --snip--

my_dog = Dog('willie', 6)
your_dog = Dog('lucy', 3)

print("My dog's name is " + my_dog.name.title() + ".")
print("My dog is " + str(my_dog.age) + " years old.")
my_dog.sit()
```

```
print("\nYour dog's name is " + your_dog.name.title() + ".")
print("Your dog is " + str(your_dog.age) + " years old.")
your_dog.sit()
```

In this example we create a dog named Willie and a dog named Lucy. Each dog is a separate instance with its own set of attributes, capable of the same set of actions:

```
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.
```

Even if we used the same name and age for the second dog, Python would still create a separate instance from the `Dog` class. You can make as many instances from one class as you need, as long as you give each instance a unique variable name or it occupies a unique spot in a list or dictionary.

TRY IT YOURSELF

9-1. Restaurant: Make a class called `Restaurant`. The `__init__()` method for `Restaurant` should store two attributes: a `restaurant_name` and a `cuisine_type`. Make a method called `describe_restaurant()` that prints these two pieces of information, and a method called `open_restaurant()` that prints a message indicating that the restaurant is open.

Make an instance called `restaurant` from your class. Print the two attributes individually, and then call both methods.

9-2. Three Restaurants: Start with your class from Exercise 9-1. Create three different instances from the class, and call `describe_restaurant()` for each instance.

9-3. Users: Make a class called `User`. Create two attributes called `first_name` and `last_name`, and then create several other attributes that are typically stored in a user profile. Make a method called `describe_user()` that prints a summary of the user's information. Make another method called `greet_user()` that prints a personalized greeting to the user.

Create several instances representing different users, and call both methods for each user.

Working with Classes and Instances

You can use classes to represent many real-world situations. Once you write a class, you'll spend most of your time working with instances created from that class. One of the first tasks you'll want to do is modify the attributes associated with a particular instance. You can modify the attributes of an instance directly or write methods that update attributes in specific ways.

The Car Class

Let's write a new class representing a car. Our class will store information about the kind of car we're working with, and it will have a method that summarizes this information:

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 = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
```

At ❶ in the `Car` class, we define the `__init__()` method with the `self` parameter first, just like we did before with our `Dog` class. We also give it three other parameters: `make`, `model`, and `year`. The `__init__()` method takes in these parameters and stores them in the attributes that will be associated with instances made from this class. When we make a new `Car` instance, we'll need to specify a make, model, and year for our instance.

At ❷ we define a method called `get_descriptive_name()` that puts a car's `year`, `make`, and `model` into one string neatly describing the car. This will spare us from having to print each attribute's value individually. To work with the attribute values in this method, we use `self.make`, `self.model`, and `self.year`. At ❸ we make an instance from the `Car` class and store it in the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have:

```
2016 Audi A4
```

To make the class more interesting, let's add an attribute that changes over time. We'll add an attribute that stores the car's overall mileage.

Setting a Default Value for an Attribute

Every attribute in a class needs an initial value, even if that value is 0 or an empty string. In some cases, such as when setting a default value, it makes sense to specify this initial value in the body of the `__init__()` method; if you do this for an attribute, you don't have to include a parameter for that attribute.

Let's add an attribute called `odometer_reading` that always starts with a value of 0. We'll also add a method `read_odometer()` that helps us read each car's odometer:

```
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("This car has " + str(self.odometer_reading) + " miles on it.")

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

This time when Python calls the `__init__()` method to create a new instance, it stores the make, model, and year values as attributes like it did in the previous example. Then Python creates a new attribute called `odometer_reading` and sets its initial value to 0 ❶. We also have a new method called `read_odometer()` at ❷ that makes it easy to read a car's mileage.

Our car starts with a mileage of 0:

```
2016 Audi A4
This car has 0 miles on it.
```

Not many cars are sold with exactly 0 miles on the odometer, so we need a way to change the value of this attribute.

Modifying Attribute Values

You can change an attribute's value in three ways: you can change the value directly through an instance, set the value through a method, or increment the value (add a certain amount to it) through a method. Let's look at each of these approaches.

Modifying an Attribute's Value Directly

The simplest way to modify the value of an attribute is to access the attribute directly through an instance. Here we set the odometer reading to 23 directly:

```
class Car():
    --snip--

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

❶ my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

At ❶ we use dot notation to access the car's `odometer_reading` attribute and set its value directly. This line tells Python to take the instance `my_new_car`, find the attribute `odometer_reading` associated with it, and set the value of that attribute to 23:

```
2016 Audi A4
This car has 23 miles on it.
```

Sometimes you'll want to access attributes directly like this, but other times you'll want to write a method that updates the value for you.

Modifying an Attribute's Value Through a Method

It can be helpful to have methods that update certain attributes for you. Instead of accessing the attribute directly, you pass the new value to a method that handles the updating internally.

Here's an example showing a method called `update_odometer()`:

```
class Car():
    --snip--

❶   def update_odometer(self, mileage):
        """Set the odometer reading to the given value."""
        self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

❷ my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

The only modification to `Car` is the addition of `update_odometer()` at ❶. This method takes in a mileage value and stores it in `self.odometer_reading`. At ❷ we call `update_odometer()` and give it `23` as an argument (corresponding to the mileage parameter in the method definition). It sets the odometer reading to 23, and read_odometer() prints the reading:

```
2016 Audi A4
This car has 23 miles on it.
```

We can extend the method update_odometer() to do additional work every time the odometer reading is modified. Let's add a little logic to make sure no one tries to roll back the odometer reading:

```
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!")
```

Now update_odometer() checks that the new reading makes sense before modifying the attribute. If the new mileage, mileage, is greater than or equal to the existing mileage, self.odometer_reading, you can update the odometer reading to the new mileage ❶. If the new mileage is less than the existing mileage, you'll get a warning that you can't roll back an odometer ❷.

Incrementing an Attribute's Value Through a Method

Sometimes you'll want to increment an attribute's value by a certain amount rather than set an entirely new value. Say we buy a used car and put 100 miles on it between the time we buy it and the time we register it. Here's a method that allows us to pass this incremental amount and add that value to the odometer reading:

```
class Car():
    --snip--

    def update_odometer(self, mileage):
        --snip--

❶   def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

❷ my_used_car = Car('subaru', 'outback', 2013)
print(my_used_car.get_descriptive_name())

❸ my_used_car.update_odometer(23500)
my_used_car.read_odometer()
```

```
❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

The new method increment_odometer() at ❶ takes in a number of miles, and adds this value to self.odometer_reading. At ❷ we create a used car, my_used_car. We set its odometer to 23,500 by calling update_odometer() and passing it 23500 at ❸. At ❹ we call increment_odometer() and pass it 100 to add the 100 miles that we drove between buying the car and registering it:

```
2013 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

You can easily modify this method to reject negative increments so no one uses this function to roll back an odometer.

NOTE *You can use methods like this to control how users of your program update values such as an odometer reading, but anyone with access to the program can set the odometer reading to any value by accessing the attribute directly. Effective security takes extreme attention to detail in addition to basic checks like those shown here.*

TRY IT YOURSELF

9-4. Number Served: Start with your program from Exercise 9-1 (page 166). Add an attribute called number_served with a default value of 0. Create an instance called restaurant from this class. Print the number of customers the restaurant has served, and then change this value and print it again.

Add a method called set_number_served() that lets you set the number of customers that have been served. Call this method with a new number and print the value again.

Add a method called increment_number_served() that lets you increment the number of customers who've been served. Call this method with any number you like that could represent how many customers were served in, say, a day of business.

9-5. Login Attempts: Add an attribute called login_attempts to your User class from Exercise 9-3 (page 166). Write a method called increment_login_attempts() that increments the value of login_attempts by 1. Write another method called reset_login_attempts() that resets the value of login_attempts to 0.

Make an instance of the User class and call increment_login_attempts() several times. Print the value of login_attempts to make sure it was incremented properly, and then call reset_login_attempts(). Print login_attempts again to make sure it was reset to 0.

Inheritance

You don't always have to start from scratch when writing a class. If the class you're writing is a specialized version of another class you wrote, you can use *inheritance*. When one class *inherits* from another, it automatically takes on all the attributes and methods of the first class. The original class is called the *parent class*, and the new class is the *child class*. The child class inherits every attribute and method from its parent class but is also free to define new attributes and methods of its own.

The __init__() Method for a Child Class

The first task Python has when creating an instance from a child class is to assign values to all attributes in the parent class. To do this, the `__init__()` method for a child class needs help from its parent class.

As an example, let's model an electric car. An electric car is just a specific kind of car, so we can base our new `ElectricCar` class on the `Car` class we wrote earlier. Then we'll only have to write code for the attributes and behavior specific to electric cars.

Let's start by making a simple version of the `ElectricCar` class, which does everything the `Car` class does:

electric_car.py

```
❶ class Car():
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        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_tesla = ElectricCar('tesla', 'model s', 2016)
  print(my_tesla.get_descriptive_name())
```

At ❶ we start with Car. When you create a child class, the parent class must be part of the current file and must appear before the child class in the file. At ❷ we define the child class, ElectricCar. The name of the parent class must be included in parentheses in the definition of the child class. The __init__() method at ❸ takes in the information required to make a Car instance.

The super() function at ❹ is a special function that helps Python make connections between the parent and child class. This line tells Python to call the __init__() method from ElectricCar's parent class, which gives an ElectricCar instance all the attributes of its parent class. The name *super* comes from a convention of calling the parent class a *superclass* and the child class a *subclass*.

We test whether inheritance is working properly by trying to create an electric car with the same kind of information we'd provide when making a regular car. At ❺ we make an instance of the ElectricCar class, and store it in my_tesla. This line calls the __init__() method defined in ElectricCar, which in turn tells Python to call the __init__() method defined in the parent class Car. We provide the arguments 'tesla', 'model s', and 2016.

Aside from __init__(), there are no attributes or methods yet that are particular to an electric car. At this point we're just making sure the electric car has the appropriate Car behaviors:

```
2016 Tesla Model S
```

The ElectricCar instance works just like an instance of Car, so now we can begin defining attributes and methods specific to electric cars.

Inheritance in Python 2.7

In Python 2.7, inheritance is slightly different. The ElectricCar class would look like this:

```
class Car(object):
    def __init__(self, make, model, year):
        --snip--

class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(make, model, year)
        --snip--
```

The super() function needs two arguments: a reference to the child class and the self object. These arguments are necessary to help Python make proper connections between the parent and child classes. When you use inheritance in Python 2.7, make sure you define the parent class using the object syntax as well.

Defining Attributes and Methods for the Child Class

Once you have a child class that inherits from a parent class, you can add any new attributes and methods necessary to differentiate the child class from the parent class.

Let's add an attribute that's specific to electric cars (a battery, for example) and a method to report on this attribute. We'll store the battery size and write a method that prints a description of the battery:

```
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 = 70

❷   def describe_battery(self):
        """Print a statement describing the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

At ❶ we add a new attribute self.battery_size and set its initial value to, say, 70. This attribute will be associated with all instances created from the ElectricCar class but won't be associated with any instances of Car. We also add a method called describe_battery() that prints information about the battery at ❷. When we call this method, we get a description that is clearly specific to an electric car:

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

There's no limit to how much you can specialize the ElectricCar class. You can add as many attributes and methods as you need to model an electric car to whatever degree of accuracy you need. An attribute or method that could belong to any car, rather than one that's specific to an electric car, should be added to the Car class instead of the ElectricCar class. Then anyone who uses the Car class will have that functionality available as well, and the ElectricCar class will only contain code for the information and behavior specific to electric vehicles.

Overriding Methods from the Parent Class

You can override any method from the parent class that doesn't fit what you're trying to model with the child class. To do this, you define a method in the child class with the same name as the method you want to override in the parent class. Python will disregard the parent class method and only pay attention to the method you define in the child class.

Say the class Car had a method called fill_gas_tank(). This method is meaningless for an all-electric vehicle, so you might want to override this method. Here's one way to do that:

```
def ElectricCar(Car):
    --snip--

    def fill_gas_tank():
        """Electric cars don't have gas tanks."""
        print("This car doesn't need a gas tank!")
```

Now if someone tries to call fill_gas_tank() with an electric car, Python will ignore the method fill_gas_tank() in Car and run this code instead. When you use inheritance, you can make your child classes retain what you need and override anything you don't need from the parent class.

Instances as Attributes

When modeling something from the real world in code, you may find that you're adding more and more detail to a class. You'll find that you have a growing list of attributes and methods and that your files are becoming lengthy. In these situations, you might recognize that part of one class can be written as a separate class. You can break your large class into smaller classes that work together.

For example, if we continue adding detail to the ElectricCar class, we might notice that we're adding many attributes and methods specific to the car's battery. When we see this happening, we can stop and move those attributes and methods to a separate class called Battery. Then we can use a Battery instance as an attribute in the ElectricCar class:

```
class Car():
    --snip--

❶ class Battery():
    """A simple attempt to model a battery for an electric car."""

❷     def __init__(self, battery_size=70):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size
```

```
❸     def describe_battery(self):
          """Print a statement describing the battery size."""
          print("This car has a " + str(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)
❹         self.battery = Battery()


my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

At ❶ we define a new class called `Battery` that doesn't inherit from any other class. The `__init__()` method at ❷ has one parameter, `battery_size`, in addition to `self`. This is an optional parameter that sets the battery's size to 70 if no value is provided. The method `describe_battery()` has been moved to this class as well ❸.

In the `ElectricCar` class, we now add an attribute called `self.battery` ❹. This line tells Python to create a new instance of `Battery` (with a default size of 70, because we're not specifying a value) and store that instance in the attribute `self.battery`. This will happen every time the `__init__()` method is called; any `ElectricCar` instance will now have a `Battery` instance created automatically.

We create an electric car and store it in the variable `my_tesla`. When we want to describe the battery, we need to work through the car's `battery` attribute:

```
my_tesla.battery.describe_battery()
```

This line tells Python to look at the instance `my_tesla`, find its `battery` attribute, and call the method `describe_battery()` that's associated with the `Battery` instance stored in the attribute.

The output is identical to what we saw previously:

```
2016 Tesla Model S
This car has a 70-kWh battery.
```

This looks like a lot of extra work, but now we can describe the battery in as much detail as we want without cluttering the `ElectricCar` class. Let's add another method to `Battery` that reports the range of the car based on the battery size:

```
class Car():
    --snip--

class Battery():
    --snip--

❶   def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

class ElectricCar(Car):
    --snip--

my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

The new method `get_range()` at ❶ performs some simple analysis. If the battery's capacity is 70 kWh, `get_range()` sets the range to 240 miles, and if the capacity is 85 kWh, it sets the range to 270 miles. It then reports this value. When we want to use this method, we again have to call it through the car's `battery` attribute at ❷.

The output tells us the range of the car based on its battery size:

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

Modeling Real-World Objects

As you begin to model more complicated items like electric cars, you'll wrestle with interesting questions. Is the range of an electric car a property of the battery or of the car? If we're only describing one car, it's probably fine to maintain the association of the method `get_range()` with the `Battery` class. But if we're describing a manufacturer's entire line of cars, we probably want to move `get_range()` to the `ElectricCar` class. The `get_range()` method would still check the battery size before determining the range, but it would report a range specific to the kind of car it's associated with. Alternatively, we could maintain the association of the `get_range()` method with the battery but pass it a parameter such as `car_model`. The `get_range()` method would then report a range based on the battery size and car model.

This brings you to an interesting point in your growth as a programmer. When you wrestle with questions like these, you're thinking at a higher logical level rather than a syntax-focused level. You're thinking not about Python, but about how to represent the real world in code. When you reach this point, you'll realize there are often no right or wrong approaches to modeling real-world situations. Some approaches are more efficient than others, but it takes practice to find the most efficient representations. If your code is working as you want it to, you're doing well! Don't be discouraged if you find you're ripping apart your classes and rewriting them several times using different approaches. In the quest to write accurate, efficient code, everyone goes through this process.

TRY IT YOURSELF

9-6. Ice Cream Stand: An ice cream stand is a specific kind of restaurant. Write a class called `IceCreamStand` that inherits from the `Restaurant` class you wrote in Exercise 9-1 (page 166) or Exercise 9-4 (page 171). Either version of the class will work; just pick the one you like better. Add an attribute called `flavors` that stores a list of ice cream flavors. Write a method that displays these flavors. Create an instance of `IceCreamStand`, and call this method.

9-7. Admin: An administrator is a special kind of user. Write a class called `Admin` that inherits from the `User` class you wrote in Exercise 9-3 (page 166) or Exercise 9-5 (page 171). Add an attribute, `privileges`, that stores a list of strings like `"can add post"`, `"can delete post"`, `"can ban user"`, and so on. Write a method called `show_privileges()` that lists the administrator's set of privileges. Create an instance of `Admin`, and call your method.

9-8. Privileges: Write a separate `Privileges` class. The class should have one attribute, `privileges`, that stores a list of strings as described in Exercise 9-7. Move the `show_privileges()` method to this class. Make a `Privileges` instance as an attribute in the `Admin` class. Create a new instance of `Admin` and use your method to show its privileges.

9-9. Battery Upgrade: Use the final version of *electric_car.py* from this section. Add a method to the `Battery` class called `upgrade_battery()`. This method should check the battery size and set the capacity to 85 if it isn't already. Make an electric car with a default battery size, call `get_range()` once, and then call `get_range()` a second time after upgrading the battery. You should see an increase in the car's range.

Importing Classes

As you add more functionality to your classes, your files can get long, even when you use inheritance properly. In keeping with the overall philosophy of Python, you'll want to keep your files as uncluttered as possible. To help, Python lets you store classes in modules and then import the classes you need into your main program.

Importing a Single Class

Let's create a module containing just the `Car` class. This brings up a subtle naming issue: we already have a file named *car.py* in this chapter, but this module should be named *car.py* because it contains code representing a car. We'll resolve this naming issue by storing the `Car` class in a module named *car.py*, replacing the *car.py* file we were previously using. From now on, any program that uses this module will need a more specific filename, such as *my_car.py*. Here's *car.py* with just the code from the class `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 = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print("This car has " + str(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
```

At ❶ we include a module-level docstring that briefly describes the contents of this module. You should write a docstring for each module you create.

Now we make a separate file called *my_car.py*. This file will import the Car class and then create an instance from that class:

my_car.py

```
❶ from car import Car

my_new_car = Car('audi', 'a4', 2016)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

The import statement at ❶ tells Python to open the car module and import the class Car. Now we can use the Car class as if it were defined in this file. The output is the same as we saw earlier:

```
2016 Audi A4
This car has 23 miles on it.
```

Importing classes is an effective way to program. Picture how long this program file would be if the entire Car class were included. When you instead move the class to a module and import the module, you still get all the same functionality, but you keep your main program file clean and easy to read. You also store most of the logic in separate files; once your classes work as you want them to, you can leave those files alone and focus on the higher-level logic of your main program.

Storing Multiple Classes in a Module

You can store as many classes as you need in a single module, although each class in a module should be related somehow. The classes Battery and ElectricCar both help represent cars, so let's add them to the module *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=60):
        """Initialize the batteery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print("This car has a " + str(self.battery_size) + "-kWh battery.")

    def get_range(self):
```

```
        """Print a statement about the range this battery provides."""
        if self.battery_size == 70:
            range = 240
        elif self.battery_size == 85:
            range = 270

        message = "This car can go approximately " + str(range)
        message += " miles on a full charge."
        print(message)

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()
```

Now we can make a new file called *my_electric_car.py*, import the `ElectricCar` class, and make an electric car:

my_electric_car.py

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

This has the same output we saw earlier, even though most of the logic is hidden away in a module:

```
2016 Tesla Model S
This car has a 70-kWh battery.
This car can go approximately 240 miles on a full charge.
```

Importing Multiple Classes from a Module

You can import as many classes as you need into a program file. If we want to make a regular car and an electric car in the same file, we need to import both classes, `Car` and `ElectricCar`:

my_cars.py

```
❶ from car import Car, ElectricCar

❷ my_beetle = Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

You import multiple classes from a module by separating each class with a comma ❶. Once you've imported the necessary classes, you're free to make as many instances of each class as you need.

In this example we make a regular Volkswagen Beetle at ❷ and an electric Tesla Roadster at ❸:

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

Importing an Entire Module

You can also import an entire module and then access the classes you need using dot notation. This approach is simple and results in code that is easy to read. Because every call that creates an instance of a class includes the module name, you won't have naming conflicts with any names used in the current file.

Here's what it looks like to import the entire car module and then create a regular car and an electric car:

my_cars.py

```
❶ import car

❷ my_beetle = car.Car('volkswagen', 'beetle', 2016)
  print(my_beetle.get_descriptive_name())

❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2016)
  print(my_tesla.get_descriptive_name())
```

At ❶ we import the entire car module. We then access the classes we need through the *module_name.class_name* syntax. At ❷ we again create a Volkswagen Beetle, and at ❸ we create a Tesla Roadster.

Importing All Classes from a Module

You can import every class from a module using the following syntax:

```
from module_name import *
```

This method is not recommended for two reasons. First, it's helpful to be able to read the import statements at the top of a file and get a clear sense of which classes a program uses. With this approach it's unclear which classes you're using from the module. This approach can also lead to confusion with names in the file. If you accidentally import a class with the same name as something else in your program file, you can create errors that are hard to diagnose. I show this here because even though it's not a recommended approach, you're likely to see it in other people's code.

If you need to import many classes from a module, you're better off importing the entire module and using the *module_name.class_name* syntax.

You won't see all the classes used at the top of the file, but you'll see clearly where the module is used in the program. You'll also avoid the potential naming conflicts that can arise when you import every class in a module.

Importing a Module into a Module

Sometimes you'll want to spread out your classes over several modules to keep any one file from growing too large and avoid storing unrelated classes in the same module. When you store your classes in several modules, you may find that a class in one module depends on a class in another module. When this happens, you can import the required class into the first module.

For example, let's store the `Car` class in one module and the `ElectricCar` and `Battery` classes in a separate module. We'll make a new module called *electric_car.py*—replacing the *electric_car.py* file we created earlier—and copy just the `Battery` and `ElectricCar` classes into this file:

electric_car.py

```
"""A set of classes that can be used to represent electric cars."""

❶ from car import Car

class Battery():
    --snip--

class ElectricCar(Car):
    --snip--
```

The class `ElectricCar` needs access to its parent class `Car`, so we import `Car` directly into the module at ❶. If we forget this line, Python will raise an error when we try to make an `ElectricCar` instance. We also need to update the `Car` module so it contains only the `Car` class:

car.py

```
"""A class that can be used to represent a car."""

class Car():
    --snip--
```

Now we can import from each module separately and create whatever kind of car we need:

my_cars.py

```
❶ from car import Car
from electric_car import ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2016)
print(my_tesla.get_descriptive_name())
```

At ❶ we import `Car` from its module, and `ElectricCar` from its module. We then create one regular car and one electric car. Both kinds of cars are created correctly:

```
2016 Volkswagen Beetle
2016 Tesla Roadster
```

Finding Your Own Workflow

As you can see, Python gives you many options for how to structure code in a large project. It's important to know all these possibilities so you can determine the best ways to organize your projects as well as understand other people's projects.

When you're starting out, keep your code structure simple. Try doing everything in one file and moving your classes to separate modules once everything is working. If you like how modules and files interact, try storing your classes in modules when you start a project. Find an approach that lets you write code that works, and go from there.

TRY IT YOURSELF

9-10. Imported Restaurant: Using your latest `Restaurant` class, store it in a module. Make a separate file that imports `Restaurant`. Make a `Restaurant` instance, and call one of `Restaurant`'s methods to show that the `import` statement is working properly.

9-11. Imported Admin: Start with your work from Exercise 9-8 (page 178). Store the classes `User`, `Privileges`, and `Admin` in one module. Create a separate file, make an `Admin` instance, and call `show_privileges()` to show that everything is working correctly.

9-12. Multiple Modules: Store the `User` class in one module, and store the `Privileges` and `Admin` classes in a separate module. In a separate file, create an `Admin` instance and call `show_privileges()` to show that everything is still working correctly.

The Python Standard Library

The *Python standard library* is a set of modules included with every Python installation. Now that you have a basic understanding of how classes work, you can start to use modules like these that other programmers have written. You can use any function or class in the standard library by including a simple `import` statement at the top of your file. Let's look at one class, `OrderedDict`, from the module `collections`.

Dictionaries allow you to connect pieces of information, but they don't keep track of the order in which you add key-value pairs. If you're creating a dictionary and want to keep track of the order in which key-value pairs are added, you can use the `OrderedDict` class from the `collections` module. Instances of the `OrderedDict` class behave almost exactly like dictionaries except they keep track of the order in which key-value pairs are added.

Let's revisit the *favorite_languages.py* example from Chapter 6. This time we'll keep track of the order in which people respond to the poll:

favorite_languages.py

```
❶ from collections import OrderedDict

❷ favorite_languages = OrderedDict()

❸ favorite_languages['jen'] = 'python'
  favorite_languages['sarah'] = 'c'
  favorite_languages['edward'] = 'ruby'
  favorite_languages['phil'] = 'python'

❹ for name, language in favorite_languages.items():
      print(name.title() + "'s favorite language is " +
          language.title() + ".")
```

We begin by importing the `OrderedDict` class from the module `collections` at ❶. At ❷ we create an instance of the `OrderedDict` class and store this instance in `favorite_languages`. Notice there are no curly brackets; the call to `OrderedDict()` creates an empty ordered dictionary for us and stores it in `favorite_languages`. We then add each name and language to `favorite_languages` one at a time ❸. Now when we loop through `favorite_languages` at ❹, we know we'll always get responses back in the order they were added:

```
Jen's favorite language is Python.
Sarah's favorite language is C.
Edward's favorite language is Ruby.
Phil's favorite language is Python.
```

This is a great class to be aware of because it combines the main benefit of lists (retaining your original order) with the main feature of dictionaries (connecting pieces of information). As you begin to model real-world situations that you care about, you'll probably come across a situation where an ordered dictionary is exactly what you need. As you learn more about the standard library, you'll become familiar with a number of modules like this that help you handle common situations.

NOTE *You can also download modules from external sources. You'll see a number of these examples in Part II, where we'll need external modules to complete each project.*

TRY IT YOURSELF

9-13. OrderedDict Rewrite: Start with Exercise 6-4 (page 108), where you used a standard dictionary to represent a glossary. Rewrite the program using the `OrderedDict` class and make sure the order of the output matches the order in which key-value pairs were added to the dictionary.

9-14. Dice: The module `random` contains functions that generate random numbers in a variety of ways. The function `randint()` returns an integer in the range you provide. The following code returns a number between 1 and 6:

```
from random import randint
x = randint(1, 6)
```

Make a class `Die` with one attribute called `sides`, which has a default value of 6. Write a method called `roll_die()` that prints a random number between 1 and the number of sides the die has. Make a 6-sided die and roll it 10 times.

Make a 10-sided die and a 20-sided die. Roll each die 10 times.

9-15. Python Module of the Week: One excellent resource for exploring the Python standard library is a site called *Python Module of the Week*. Go to *http://pymotw.com/* and look at the table of contents. Find a module that looks interesting to you and read about it, or explore the documentation of the `collections` and `random` modules.

Styling Classes

A few styling issues related to classes are worth clarifying, especially as your programs become more complicated.

Class names should be written in *CamelCaps*. To do this, capitalize the first letter of each word in the name, and don't use underscores. Instance and module names should be written in lowercase with underscores between words.

Every class should have a docstring immediately following the class definition. The docstring should be a brief description of what the class does, and you should follow the same formatting conventions you used for writing docstrings in functions. Each module should also have a docstring describing what the classes in a module can be used for.

You can use blank lines to organize code, but don't use them excessively. Within a class you can use one blank line between methods, and within a module you can use two blank lines to separate classes.

If you need to import a module from the standard library and a module that you wrote, place the import statement for the standard library module first. Then add a blank line and the import statement for the module you wrote. In programs with multiple import statements, this convention makes it easier to see where the different modules used in the program come from.

Summary

In this chapter you learned how to write your own classes. You learned how to store information in a class using attributes and how to write methods that give your classes the behavior they need. You learned to write `__init__()` methods that create instances from your classes with exactly the attributes you want. You saw how to modify the attributes of an instance directly and through methods. You learned that inheritance can simplify the creation of classes that are related to each other, and you learned to use instances of one class as attributes in another class to keep each class simple.

You saw how storing classes in modules and importing classes you need into the files where they'll be used can keep your projects organized. You started learning about the Python standard library, and you saw an example based on the `OrderedDict` class from the `collections` module. Finally, you learned to style your classes using Python conventions.

In Chapter 10 you'll learn to work with files so you can save the work you've done in a program and the work you've allowed users to do. You'll also learn about *exceptions*, a special Python class designed to help you respond to errors when they arise.