Chapter IX 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 defne 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 defne 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 effciently. You'll store your classes in modules and import classes written by other programmers into your own program files.

9.1 Creating and Using a Class

9.1.1 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():

```
class Dog():
    def __init__(self, name, age):
    """Initialize name and age attributes."""
        self.name = name
        self.age = age

    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!")
```

```python
    def __init__(self, name, age):
    """Initialize name and age attributes."""
        self.name = name
        self.age = age
```

A function that’s part of a class is a method.The init() method at w 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 defne the init() method to have three parameters: self, name, and age. 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.

```
my_dog = Dog('willie', 6)
```

Any variable prefxed 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.

9.1.2 Making an Instance from a Class

Let's make an instance representing a specifc dog:

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

my_dog = Dog('willie', 6)

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

1. accessing attributes

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

```
my_dog.name
```

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 we use the same approach to work with the attribute age.。

2. Calling Methods

After we create an instance from the class Dog, we can use dot notation to call any method defned 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.

print

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

3. 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()
```

```
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.
```

TRY IT

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 profle 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

9.2 Working with Classes and Instances

9.2.1 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:

```
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

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = str(self.year) + ' ' + self.make
                                    + ' ' + self.model
        return long_name.title()

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

9.2.2 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 avalue of 0. We’ll also add a method read_odometer() that helps us read each car’s odometer:

```python
class 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):
        --snip--

    def read_odometer(self):
        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()
```

Our car starts with a mileage of 0:

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

9.2.3 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.

1. 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())

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

2. 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--
    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())

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

The only modifcation to Car is the addition of update_odometer() .This method takes in a mileage value and stores it in self.odometer_reading.

3. 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 at❸.At❹we call increment_odometer() and pass it 100 to add the 100 miles that we drove between buying the car and registering it:

Output:

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

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.

9.3 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.

9.3.1 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 specifc 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 specifc 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

```
① class 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
```

```
② class ElectricCar(Car):
      """Represent aspects of a car, specific to electric vehicle
③     def __init__(self, make, model, year):
          """Initialize attributes of the parent class."""
④         super().__init__(make, model, year)

⑤ 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 defnition 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 specifc to electric cars.

9.3.2 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.

9.3.3 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):
   def __init__(self, make, model, year):
      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 specifc 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 specifc to electric vehicles.

9.3.4 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 defne 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.

9.3.5 Instances as Attributes

When modeling something from the real world in code, you may fnd 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 vehicl
      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()
```

At❶ we define a new class called Battery that doesn’t inherit from any other class. The init() method at❷has 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.battery❹.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.
```

9.3.6 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, effcient code, everyone goes through this process.

TRY IT YURDSELF

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.

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.

9.4 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.

9.4.1 Importing a Single Class

Let's create a module containing just the Car class.

car.py

```
class 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):
    """Return a neatly formatted descriptive name."""
    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
```

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

```
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 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.
```

9.4.2 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:

```python
class Car():
    --snip--
class Battery():
    def __init__(self, battery_size=60):
        self.battery_size = battery_size
    def describe_battery(self):
        print("This car has a " + str(self.battery_size) +
                                                "-kWh battery.")
    def get_range(self):
        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):
    def __init__(self, make, model, year):
        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:

```
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.
```

9.4.3 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 ❷and an electric Tesla Roadster at ❸:

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

9.4.4 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:

```
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())
```

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

9.4.5 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 modul.

9.4.6 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

```
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.

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

```
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
```

9.4.7 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.

9.5 The Python Standard Library

The Python standard library is a set of modules included with every Python installation. 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:

```
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 at❶ .At❷we 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 time❸.Now when we loop through favorite_languages at❹, 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.
```

TRY IT YOUSELF

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.

9.6 Styling Classes

- 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.

9.7 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.