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 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(f"{self.name} is now sitting.")

      def roll_over(self):
          """Simulate rolling over in response to a command."""
          print(f"{self.name} 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. There are no parentheses in the class definition 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 that 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. Make sure to use two underscores on each side of `__init__()`. If you use just one on each side, the method won't be called automatically when you use your class, which can result in errors that are difficult to identify.

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 method later (to create an instance of `Dog`), the method call will automatically pass the `self` argument. Every method call associated with an instance 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. The line `self.name = name` takes the value associated with the parameter `name` and assigns it to 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 to run, 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 robotic dog to sit and roll over.

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(f"My dog's name is {my_dog.name}.")
❸ print(f"My dog is {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. Python then returns an instance representing this dog. We assign that instance to 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`.

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(f"My dog's name is {my_dog.name}.")
print(f"My dog is {my_dog.age} years old.")
my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")
print(f"Your dog is {your_dog.age} years old.")
your_dog.sit()
```

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 = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

❸ my_new_car = Car('audi', 'a4', 2019)
  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 assigns them to 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 assign it to the variable `my_new_car`. Then we call `get_descriptive_name()` to show what kind of car we have:

```
2019 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

When an instance is created, attributes can be defined without being passed in as parameters. These attributes can be defined in the `__init__()` method, where they are assigned a default value.

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(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2019)
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:

```
2019 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', 2019)
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:

```
2019 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', 2019)
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 assigns it to `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:

```
2019 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', 2015)
  print(my_used_car.get_descriptive_name())

❸ my_used_car.update_odometer(23_500)
  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 `23_500` 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:

```
2015 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 162). 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 162). 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 takes on 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 can inherit any or all of the attributes and methods of its parent class, but it's also free to define new attributes and methods of its own.

The `__init__()` Method for a Child Class

When you're writing a new class based on an existing class, you'll often want to call the `__init__()` method from the parent class. This will initialize any attributes that were defined in the parent `__init__()` method and make them available in the child 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 = f"{self.year} {self.manufacturer} {self.model}"
        return long_name.title()

    def read_odometer(self):
        print(f"This car has {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', 2019)
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 a 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 allows you to call a method from the parent class. This line tells Python to call the `__init__()` method from `Car`, which gives an `ElectricCar` instance all the attributes defined in that method. 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 assign it to `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 `2019`.

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:

```
2019 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.

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 = 75

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

my_tesla = ElectricCar('tesla', 'model s', 2019)
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, `75`. 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