PREV
Lesson 30. Making your own object types

NEXT
Lesson 32. Working with your own object types

Lesson 31. Creating a class for an object type

After reading lesson 31, you'll be able to

- Define a Python class
- Define data properties for a class
- Define operations for a class
- Use a class to create objects of that type and perform operations

You can create your own types of objects to suit whatever your program needs. Except for atomic object types (`int`, `float`, `bool`), any object that you create is made up of other preexisting objects. As someone who implements a new object type, you get to define the properties that make up the object and the behaviors that you'll allow an object to have (on its own or when interacting with other objects).

You usually define your own objects in order to have customized properties and behaviors, so that you can reuse them. In this lesson, you'll view code you write from two points of view, just as when you wrote your own functions. You'll separate yourself from a programmer/writer of a new object type and from the programmer/user of a newly created object type.

Before defining an object type by using a class, you should have a general idea of how you'll implement it by answering two questions:

- What is your object made up of (its characteristics, or properties)?
- What do you want your object to do (its behaviors, or operations)?

Consider this

Lists and integers are two types of objects. Name some operations you can do

- On a list
- On one or more numbers

Do you notice anything different between the majority of the operations you can do on each?

Answer:

- `Append, extend, pop, index, remove, in`
- `+, -, *, /, %, negate, convert to a string`

`Most of the operations on lists are done with dot notation, but the operations on numbers use mathematical symbols.`

31.1. IMPLEMENTING A NEW OBJECT TYPE BY USING A CLASS

The first part of creating your own object type is to define the class. You use the `class` keyword to do this. A simple object type you may want to create is an object representing a circle. You tell Python that you want to define a new object type through a class. Consider the following line:

```
class Circle(object):
```

The keyword `class` starts the definition. The word `Circle` is the name of your class as well as the name of the object type that you want to define. In the parentheses, the word `object` means that your class is going to be a Python object. All classes you define are going to be Python objects. As such, objects created using your class will inherit all basic behaviors and functionality that any Python object has—for example, binding a variable to your ob-

Quick check 31.1

Write a line to define a class for the following objects:

1

A person

2

A car

3

A computer

31.2. DATA ATTRIBUTES AS OBJECT PROPERTIES

After you start defining the class, you'll have to decide how your object will be initialized. For the most part, this involves deciding how you'll represent your object and the data that will define it. You'll initialize these objects. The object properties are called *data attributes* of the object.

31.2.1. Initializing an object with __init__

To initialize your object, you have to implement a special operation, the `__init__` operation (notice the double underscores before and after the word `init`):

```
class Circle(object):
    def __init__(self):
       # code here
```

The `__init__` definition looks like a function, except it's defined inside a class. Any function defined inside a class is named a *method*.

Definition

A method is a function defined inside a class, and defines an operation you can do on an object of that type.

The code inside the `__init__` method generally initializes the data attributes that define the object. You decide that your circle class initializes a circle with radius 0 when first created.

31.2.2. Creating an object property inside __init__

A data attribute of one object is another object. Your object may be defined by more than one data attribute. To tell Python that you want to define a data attribute of the object, you use a variable named `self` with a dot after it. In the `Circle` class, you initialize a radius as the data attribute of a circle, and initialize it to 0:

```
class Circle(object):
    def __init__(self):
        self.radius = 0
```

Notice that in the definition of `__init__`, you take one parameter named `self`. Then, inside the method, you use `self.` to set a data attribute of your circle. The variable `self` is used to tell Python that you'll be using this variable to refer to any object you'll create of the type `Circle`. Any circle you create will have its own radius accessible through `self.radius`. At this point, notice that you're still defining the class and haven't created any specific object yet. You can think of `self` as a placeholder variable for any object of type `Circle`.

Inside `__init__`, you use `self.radius` to tell Python that the variable `radius` belongs to an object of type `Circle`. Every object you create of type `Circle` will have its own variable named `radius`, whose value can differ between objects. Every variable defined using `self.` refers to a data attribute of the object.

Quick check 31.2

Q1:

Write an `__init__` method that contains data attribute initializations for each of the following scenarios:

- A car
- A computer

31.3. METHODS AS OBJECT OPERATIONS AND BEHAVIORS

Your object has behaviors defined by operations you can do with or on the object. You implement operations via methods. For a circle, you can change its radius by writing another method:

```
class Circle(object):
    def __init__(self):
        self.radius = 0
    def change_radius(self, radius):
        self.radius = radius
```

A method looks like a function. As in the `__init__` method, you use `self` as the first parameter to the method. The method definition says this is a method named `change_radius`, and it takes one parameter named `ra-dius`.

Inside the method is one line. Because you want to modify a data attribute of the class, you use `self.` to access the radius inside the method and change its value.

Another behavior for the circle object is to tell you its radius:

```
class Circle(object):
    def __init__(self):
        self.radius = 0
    def change_radius(self, radius):
        self.radius = radius
    def get_radius(self):
        return self.radius
```

Again, this is a method, and it takes no parameters besides `self`. All it does is return the value of its data attribute `radius`. As before, you use `self` to access the data attribute.

Quick check 31.3

Q1:

Suppose you create a `Door` object type with the following initialization method:

```
class Door(object):
    def __init__(self):
        self.width = 1
        self.height = 1
        self.open = False
```

- Write a method that returns whether the door is open.
- Write a method that returns the area of the door.

31.4. USING AN OBJECT TYPE YOU DEFINED

You've already been using object types written by someone else every time you've created an object: for example, `int = 3` or `L = []`. These are shorthand notations instead of using the name of the class.

The following are equivalent in Python: `L = []` and `L = list()`. Here, `list` is the name of the `list` class that someone implemented for others to use.

Now, you can do the same with your own object types. For the `Circle` class, you create a new Circle object as follows:

```
one_circle = Circle()
```

We say that the variable `one_circle` is bound to an object that's an in-stance of the `Circle` class. In other words, `one_circle` is a `Circle`.

Definition

An instance is a specific object of a certain object type.

You can create as many instances as you like by calling the class name and binding the new object to another variable name:

```
one_circle = Circle()
another_circle = Circle()
```

After you create instances of the class, you can perform operations on the objects. On a `Circle` instance, you can do only two operations: change its radius or get the object to tell you its radius.

Recall that the dot notation means that the operation acts on a particular object. For example,

```
one_circle.change_radius(4)
```

Notice that you pass in one actual parameter (4) to this function, whereas the definition had two formal parameters (`self` and `radius`). Python always automatically assigns the value for `self` to be the object on which the method is called (`one_circle`, in this case). The object on which the method is called is the object right before the dot. This code changes the radius of only this instance, named `one_circle`, to `4`. All other instances of the object that may have been created in a program remain unchanged. Say you ask for the radius values as shown here:

```
print(one_circle.get_radius())
print(another_circle.get_radius())
```

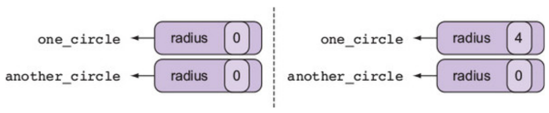
This prints the following:

```
4
0
```

Here, `one_circle`'s radius was changed to 4, but you didn't change the radius of `another_circle`. How do you know this? Because the radius of a circle was a data attribute and defined with `self`.

This is shown in figure 31.1: each object has its own data attribute for the radius, and changing one doesn't affect the other.

Figure 31.1. On the left are the data attributes of two circle objects. On the right, you can see that one data attribute changed after using dot notation on it to change the value.



Quick check 31.4

Q1:

Suppose you create a `Door` object type in the following way:

```
class Door(object):
    def __init__(self):
        self.width = 1
        self.height = 1
        self.open = False
    def change_state(self):
        self.open = not self.open
    def scale(self, factor):
        self.height *= factor
        self.width *= factor
```

- Write a line that creates a new `Door` object and binds it to a variable named `square_door`.
- Write a line that changes the state of the `square_door`.
- Write a line that scales the door to be three times bigger.

31.5. CREATING A CLASS WITH PARAMETERS IN __INIT__

Now, you want to make another class to represent a rectangle. The following listing shows the code.

Listing 31.1. A `Rectangle` class

```
        self.length = length
        self.width = width
    def set_length(self, length):
        self.length = length
    def set_width(self, width):
        self.width = width
```

This code presents a couple of new ideas. First, you have two parameters in `__init__` besides `self`. When you create a new `Rectangle` object, you'll have to initialize it with two values: one for the length and one for the width.

You can do that this way:

```
a_rectangle = Rectangle(2,4)
```

Say you don't put in two parameters and do this:

```
bad_rectangle = Rectangle(2)
```

Then Python gives you an error saying that it's expecting two parameters when you initialize the object but you gave it only one:

```
TypeError: __init__() missing 1 required positional a
```

The other thing to notice in this `__init__` is that the parameters and the data attributes have the same name. They don't have to be the same, but often they are. Only the attribute names matter when you want to access the values of object properties using the class methods. The parameters to the methods are formal parameters to pass data in to initialize the object, and are temporary; they last until the method call ends, while data attributes persist throughout the life of the object instance.

31.6. DOT NOTATION ON THE CLASS NAME, NOT ON AN OBJECT

You've been initializing and using objects by leaving out the `self` parameter and letting Python automatically decide what the value for `self` should be. This is a nice feature of Python, which allows programmers to write more-concise code.

There's a more explicit way to do this in the code, by giving a parameter for `self` directly, without relying on Python to detect what it should be.

Going back to the `Circle` class you defined, you can again initialize an object, set the radius, and print the radius as follows:

```
c = Circle()
c.change_radius(2)
r = c.get_radius()
print(r)
```

After initializing an object, a more explicit way of doing the operations on the object is by using the class name and object directly, like this:

```
c = Circle()
Circle.change_radius(c, 2)
r = Circle.get_radius(c)
print(r)
```

Notice that you're calling the methods on the class name. Additionally, you're now passing two parameters to `change_radius`:

- `c` is the object you want to do the operation on and is assigned to `self`.
- `2` is the value for the new radius.

If you call the method on the object directly, as in `c.change_radius(2)`, Python knows that the parameter for `self` is to be `c`, infers that `c` is an object of type `Circle`, and translates the line behind the scenes to be `Circle.change_radius(c, 2)`.

Quick check 31.5

Q1:

You have the following lines of code. Convert the ones noted to use the explicit way of calling methods (using dot notation on the class name):

```
a = Rectangle(1,1)
b = Rectangle(1,1)
a.set_length(4)      # change this
b.set_width(4)       # change this
```

SUMMARY

In this lesson, my objective was to teach you how to define a class in Python. Here are the major takeaways:

- A class defines an object type.
- A class defines data attributes (properties) and methods (operations).
- `self` is a variable name conventionally used to refer to a generic instance of the object type.
- An `__init__` method is a special operation that defines how to initialize an object. It's called when an object is created.
- You can define other methods (for example, functions inside a class) to do other operations.
- When using a class, the dot notation on an object accesses data attributes and methods.

Let's see if you got this...

Q31.1

Write a method for the circle class named `get_area`. It returns the area of a circle by using the formula 3.14 * radius^2. Test your method by creating an object and printing the result of the method call.

Q31.2

Write two methods for the `Rectangle` class named `get_area` and `get_perimeter`. Test your methods by creating an object and printing the result of the method calls:

- `get_area` returns the area of a rectangle by using the formula length * width.
- `get_perimeter` returns the perimeter of a rectangle by using 2 * length + 2 * width.

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 30. Making your own object types

NEXT
Lesson 32. Working with your own object types