



Python Objects and Classes Lab

Estimated time needed: 10 minutes

Objective:

By the end of this reading, you should be able to:

1. Learn the fundamental concepts of Python objects and classes.
2. Understand the Structure of Classes and object code
3. Understand the concept with real-world example

Introduction to Classes and object

Python is an object-oriented programming (OOP) language, which means it uses a paradigm centered around objects and classes. Here's an explanation of these fundamental concepts:

Classes:

A class is a blueprint or template for creating objects. It defines the structure and behavior that its objects will have.

Think of a class as a cookie cutter, and objects as the cookies cut from that template.

In Python, classes are created using the `class` keyword.

Creating Classes:

When you create a class, you specify the attributes(data) and methods (functions) that objects of that class will have.

Attributes are defined as variables within the class, and methods are defined as functions.

For example, you can design a "Car" class with attributes such as "color" and "speed," along with methods like "accelerate."

Objects:

An *object* is a fundamental unit in Python that represents a real-world entity or concept. Objects can be tangible (like a car) or abstract (like a student's grade).

Every object has two main characteristics:

State:

The *attributes or data* that describe the object. For our "Car" object, this might include attributes like "color", "speed", and "fuel level".

Behavior:

The *actions or methods* that the object can perform. In Python, methods are functions that belong to objects and can change the object's state or perform specific operations.

Instantiating Objects:

- Once you've defined a class, you can create individual objects (instances) based on that class.
- Each object is independent and has its own set of attributes and methods.
- To create an object, you use the class name followed by parentheses, so: "my_car = Car()"

Interacting with Objects:

You interact with objects by calling their methods or accessing their attributes using dot notation.

For example, if you have a Car object named **my_car**, you can set its color with **my_car.color = "blue"** and accelerate it with **my_car.accelerate()** if there's an accelerate method defined in the class.

Structure of classes and object code

Please don't directly copy and use this code as it's meant as a template for explanation and isn't tailored for specific results.

Class Declaration (class ClassName):

- The class keyword is used to declare a class in Python.
- ClassName is the name of the class, typically following CamelCase naming conventions.

```
1. 1
```

```
1. class ClassName:
```

Copied!

Class Attributes (class_attribute = value):

- Class attributes are variables that are shared among all instances (objects) of the class.
- They are defined within the class but outside of any methods.

```
1. 1
```

```
2. 2
```

```
3. 3
```

```
1. class ClassName:
```

```
2.     # Class attributes (shared by all instances)
```

```
3.     class_attribute = value
```

Copied!

Constructor Method (def init(self, attribute1, attribute2, ...):):

- The `__init__` method is a special method known as the constructor.
- It initializes the **instance attributes** (also called instance variables) when an object is created.
- The `self` parameter is the first parameter of the constructor, referring to the instance being created.
- **attribute1**, **attribute2**, and so on are parameters passed to the constructor when creating an object.
- Inside the constructor, `self.attribute1`, `self.attribute2`, and so on are used to assign values to instance attributes.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
```

```
1. class ClassName:
2.     # Class attributes (shared by all instances)
3.     class_attribute = value
4.
5.     # Constructor method (initialize instance attributes)
6.     def __init__(self, attribute1, attribute2, ...):
7.         pass
8.         # ...
```

Copied!

Instance Attributes (`self.attribute1 = attribute1`):

- Instance attributes are variables that store data specific to each instance of the class.
- They are initialized within the **init** method using the `self` keyword followed by the attribute name.
- These attributes hold unique data for each object created from the class.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
```

```
1. class ClassName:
2.     # Class attributes (shared by all instances)
3.     class_attribute = value
4.
5.     # Constructor method (initialize instance attributes)
6.     def __init__(self, attribute1, attribute2, ...):
7.         self.attribute1 = attribute1
8.         self.attribute2 = attribute2
9.         # ...
```

Copied!

Instance Methods (`def method1(self, parameter1, parameter2, ...):`):

- Instance methods are functions defined within the class.
- They operate on the instance's data (instance attributes) and can perform actions specific to instances.

- The **self** parameter is required in instance methods, allowing them to access instance attributes and call other methods within the class.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
```

```
1. class ClassName:
2.     # Class attributes (shared by all instances)
3.     class_attribute = value
4.
5.     # Constructor method (initialize instance attributes)
6.     def __init__(self, attribute1, attribute2, ...):
7.         self.attribute1 = attribute1
8.         self.attribute2 = attribute2
9.         # ...
10.
11.     # Instance methods (functions)
12.     def method1(self, parameter1, parameter2, ...):
13.         # Method logic
14.         pass
15.
```

Copied!

Using same steps you can define multiple instance methods.

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
```

```
1. class ClassName:
2.     # Class attributes (shared by all instances)
3.     class_attribute = value
4.
5.     # Constructor method (initialize instance attributes)
6.     def __init__(self, attribute1, attribute2, ...):
```

```
7.         self.attribute1 = attribute1
8.         self.attribute2 = attribute2
9.         # ...
10.
11.     # Instance methods (functions)
12.     def method1(self, parameter1, parameter2, ...):
13.         # Method logic
14.         pass
15.
16.     def method2(self, parameter1, parameter2, ...):
17.         # Method logic
18.         pass
```

Copied!

Note:- Now, we have successfully created a dummy class.

Creating Objects (Instances):

- To create objects (instances) of the class, you call the class like a function and provide arguments required by the constructor.
- Each object is a distinct instance of the class, with its own set of instance attributes and the ability to call methods defined in the class.

```
1. 1
2. 2
3. 3

1. # Create objects (instances) of the class
2. object1 = ClassName(arg1, arg2, ...)
3. object2 = ClassName(arg1, arg2, ...)
```

Copied!

Calling methods on objects:

- In this section we will call methods on objects, specifically `object1` and `object2`.
- The methods **`method1`** and **`method2`** are defined in the **`ClassName`** **class**, and you're calling them on **`object1`** and **`object2`** respectively.
- You pass values **`param1_value`** and **`param2_value`** as arguments to these methods. These arguments are used within the method's logic.

Method 1: Using dot notation:

- This is the most straightforward way to call an object's method. In this we use the dot notation (**`object.method()`**) to directly invoke the method on the object.
- For example, `result1 = object1.method1(param1_value, param2_value, ...)` calls `method1` on `object1`.

```
1. 1
2. 2
3. 3
4. 4

1. # Calling methods on objects
2. # Method 1: Using dot notation
3. result1 = object1.method1(param1_value, param2_value, ...)
4. result2 = object2.method2(param1_value, param2_value, ...)
```

Copied!

Method 2: Assigning object methods to variables:

- Here's an alternative way to call an object's method by assigning the method reference to a variable.
- `method_reference = object1.method1` assigns the method **method1** of **object1** to the variable **method_reference**.
- Later, we call the method using the variable like this: **result3 = method_reference(param1_value, param2_value, ...)**.

```
1. 1
2. 2
3. 3
```

```
1. # Method 2: Assigning object methods to variables
2. method_reference = object1.method1 # Assign the method to a variable
3. result3 = method_reference(param1_value, param2_value, ...)
```

Copied!

Accessing object attributes:

- Here, we are accessing an object's attribute using dot notation.
- `attribute_value = object1.attribute1` retrieves the value of the attribute **attribute1** from **object1** and assigns it to the variable **attribute_value**.

```
1. 1
2. 2
```

```
1. # Accessing object attributes
2. attribute_value = object1.attribute1 # Access the attribute using dot notation
```

Copied!

Modifying object attributes:

- We are modifying an object's attribute using dot notation.
- `object1.attribute2 = new_value` sets the attribute **attribute2** of **object1** to the new value **new_value**.

```
1. 1
2. 2
```

```
1. # Modifying object attributes
2. object1.attribute2 = new_value # Change the value of an attribute using dot notation
```

Copied!

Accessing class attributes (shared by all instances):

- Finally, we access a class attribute, which is shared by all instances of the class.
- `class_attr_value = ClassName.class_attribute` accesses the class attribute `class_attribute` from the `ClassName` class and assigns its value to the variable `class_attr_value`.

```
1. 1
2. 2
```

1. # Accessing class attributes (shared by all instances)
2. class_attr_value = ClassName.class_attribute

Copied!

Real-world example

Let's write a python program that simulates a simple car class, allowing you to create car instances, accelerate them, and display their current speeds.

1. Let's start by defining a Car class that includes the following attributes and methods:

- Class attribute `max_speed`, which is set to **120 km/h**.
- Constructor method `__init__` that takes parameters for the **car's make, model, color, and an optional speed (defaulting to 0)**. This method initializes instance attributes for make, model, color, and speed.
- Method `accelerate(self, acceleration)` that allows the car to accelerate. If the acceleration does not exceed the `max_speed`, update the **car's speed** attribute. Otherwise, set the speed to the **max_speed**.
- Method `get_speed(self)` that returns the current speed of the car.

```

1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21

```

```

1. class Car:
2.     # Class attribute (shared by all instances)
3.     max_speed = 120 # Maximum speed in km/h
4.
5.     # Constructor method (initialize instance attributes)
6.     def __init__(self, make, model, color, speed=0):
7.         self.make = make
8.         self.model = model
9.         self.color = color
10.        self.speed = speed # Initial speed is set to 0
11.
12.    # Method for accelerating the car
13.    def accelerate(self, acceleration):
14.        if self.speed + acceleration <= Car.max_speed:
15.            self.speed += acceleration
16.        else:
17.            self.speed = Car.max_speed

```

```
18.  
19.     # Method to get the current speed of the car  
20.     def get_speed(self):  
21.         return self.speed
```

Copied!

2. Now, we will instantiate two objects of the Car class, each with the following characteristics:

- **car1: Make = "Toyota", Model = "Camry", Color = "Blue"**
- **car2: Make = "Honda", Model = "Civic", Color = "Red"**

```
1. 1  
2. 2  
3. 3
```

```
1. # Create objects (instances) of the Car class  
2. car1 = Car("Toyota", "Camry", "Blue")  
3. car2 = Car("Honda", "Civic", "Red")
```

Copied!

3. Using the accelerate method, we will increase the speed of car1 by 30 km/h and car2 by 20 km/h.

```
1. 1  
2. 2  
3. 3  
4. 4
```

```
1.  
2. # Accelerate the cars  
3. car1.accelerate(30)  
4. car2.accelerate(20)
```

Copied!

4. Lastly, we will display the current speed of each car by utilizing the `get_speed` method.

```
1. 1  
2. 2  
3. 3  
4. 4
```

```
1.  
2. # Print the current speeds of the cars  
3. print(f"{car1.make} {car1.model} is currently at {car1.get_speed()} km/h.")  
4. print(f"{car2.make} {car2.model} is currently at {car2.get_speed()} km/h.")
```

Copied!

Next Steps

In conclusion, this reading provides a fundamental understanding of objects and classes in Python, essential concepts in object-oriented programming. Classes serve as blueprints for creating objects, encapsulating both data attributes and methods. Objects represent real-world entities and possess their own unique state and behavior. The structured code example presented in the reading outlines the key elements of a class, including class attributes, the constructor method for initializing instance attributes, and instance methods for defining object-specific functionality.

You can apply the concepts of objects and classes in the upcoming laboratory session to gain hands-on experience.

Author

[Akansha Yadav](#)

Changelog

Date	Version	Changed by	Change Description
2023-09-05	1.0	Akansha Yadav	Created reading