- Fundamental concepts of Python objects and classes.
- Structure of classes and object code.
- Real-world examples related to objects and classes.

Introduction to classes and object

Python is an object-oriented programming (OOP) language that uses a paradigm centered around objects and classes.

Let's look at 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, you can create classes 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 your "Car" object, this might include attributes like "color", "speed", and "fuel level".

Behavior

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 because it is a template for explanation and not 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
```

```
class ClassName:
```

Copied!

Class attributes (class_attribute = value)

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

```
1
2
3
```

```
class ClassName:
    # Class attributes (shared by all instances)
    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.

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        pass
        # ...
```

Copied!

Instance attributes (self.attribute1 = attribute1)

- Instance attributes are variables that store data specific to each class instance.
- 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
2
3
4
5
6
7
8
9
```

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...
```

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
2
3
```

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
        # Method logic
        pass
```

Copied!

Using the same steps you can define multiple instance methods.

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

```
class ClassName:
    # Class attributes (shared by all instances)
    class_attribute = value

    # Constructor method (initialize instance attributes)
    def __init__(self, attribute1, attribute2, ...):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
        # ...

    # Instance methods (functions)
    def method1(self, parameter1, parameter2, ...):
        # Method logic
        pass

```

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

```
1
2
3
```

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

Copied!

Calling methods on objects

- In this section, you 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, use the dot notation **(object.method())** to invoke the method on the object directly.
- For example, `result1 = object1.method1(param1_value, param2_value, ...)` calls method1 on object1.

```
1
2
3
4
```

```
# Calling methods on objects
# Method 1: Using dot notation
result1 = object1.method1(param1_value, param2_value, ...)
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**.

Copied!

Accessing object attributes

- Here, you 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
2
```

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

Copied!

Modifying object attributes

- You will modify an object's attribute using dot notation.
- `object1.attribute2 = new_value` sets the attribute **attribute2** of **object1** to the new value **new_value**.

```
1
2
```

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

Copied!

Accessing class attributes (shared by all instances)

- Finally, access a class attribute shared by all class instances.
- `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
2
```

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

Copied!

Real-world example

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

```
class Car:
    # Class attribute (shared by all instances)
    max_speed = 120  # Maximum speed in km/h

    # Constructor method (initialize instance attributes)
    def __init__(self, make, model, color, speed=0):
        self.make = make
        self.model = model
        self.color = color
        self.speed = speed  # Initial speed is set to 0

    # Method for accelerating the car
    def accelerate(self, acceleration):
        if self.speed + acceleration <= Car.max_speed:
            self.speed += acceleration
        else:
            self.speed = Car.max_speed

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

Copied!

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

```
car2 = Car("Honda", "Civic", "Red")
```

Copied!

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

```
1
2
3
```

```
# Accelerate the cars
car1.accelerate(30)
car2.accelerate(20)
```

Copied!

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

```
1
2
3
```

```
# Print the current speeds of the cars
print(f"{car1.make} {car1.model} is currently at {car1.get_speed()} km/h.")
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 data attributes and methods. Objects represent real-world entities and possess their 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.

In the upcoming laboratory session, you can apply the concepts of objects and classes to gain hands-on experience.

Author

Muhammad Faheem Iqbal