Object-Oriented Programming (OOPS-1)

Introduction to OOPS

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of *"objects"* and *"classes"* that contain data and methods. The primary purpose of OOP is to increase the flexibility and maintainability of programs. It is used to structure a software program into simple, reusable pieces of code **blueprints** (called *classes*) which are used to create individual instances of *objects*.

Python supports a variety of programming approaches. One of the most useful and popular programming approaches is OOPS.

What is an Object?

The object is an entity that has a state and a behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
Integers, strings, floating-point numbers, even arrays and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number **12** is an object, the string **"Hello, world"** is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

What is a Class?

A class is a **blueprint** that defines the variables and the methods (Characteristics) common to all objects of a certain kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like 4 wheels, 1 steering wheel, windows, breaks etc. Maruti 800 (The **Car** object) has all these features.

Classes vs Objects (Or Instances)

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.

A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Python

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon(**:**). Any code that is indented below the class definition is considered part of the class's body.
Here is an example of a **Car** class:

```
class Car:
    pass
```

The body of the **Car** class consists of a single statement: the `pass` keyword. As we have discussed earlier, `pass` is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note: Python class names are written in *CapitalizedWords* notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Python, the `.__init__()` method is called the **constructor** and is always called when an object is created.

- **Note:** Names that have leading and trailing double underscores are reserved for special use like the **__init__** method for object constructors. These methods are known as **dunder methods**.

Syntax of Constructor Declaration

```
def __init__(self):
    # body of the constructor
```

Types of constructors

- **Default Constructor:** The default constructor is a simple constructor that doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed known as `self`.
- **Parameterized Constructor:** A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as `self` and the rest of the arguments are provided by the programmer.

The properties that all **Car** objects must have been defined in `.__init__()`. Every time a new **Car** object is created, `.__init__()` sets the initial state of the object by assigning the values of the object's properties. That is, `.__init__()` initializes each new instance of the class.
When a new class instance is created, the instance is automatically passed to the self parameter in `.__init__()` so that new attributes can be defined on the object.

The `self` Parameter

- The `self` parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class.

- You can give `.__init__()` any number of parameters, but the first parameter will always be a variable called `self`.

Let's update the Car class with the `.__init__()` method that creates `name` and `topSpeed` attributes:

```
class Car:
    def __init__(self, name, topSpeed):
        self.name = name
        self.topSpeed= topSpeed
```

Note: The `.__init__()` method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the **Car** class.

In the body of `.__init__()`, two statements are using the self variable:

1. `self.name = name` creates an attribute called `name` and assigns to it the value of the `name` parameter.
2. `self.topSpeed= topSpeed` creates an attribute called `topSpeed` and assigns to it the value of the `topSpeed` parameter.

Instance Attributes

Attributes created in `.__init__()` are called **instance** attributes. An instance attribute's value is specific to a particular instance of the class. All **Car** objects have a `name` and a `topSpeed`, but the values for the `name` and `topSpeed` attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Class Attributes

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following **Car** class has a class attribute called `color` with the value `"Black"`:

```
class Car:
    # Class attribute
    color = "Black"

    def __init__(self, name, topSpeed):
        self.name = name
        self.topSpeed= topSpeed
```

- Class attributes are defined directly beneath the first line of the class name and are indented by four spaces.
- They must always be assigned an initial value.
- When an instance of the class is created, the class attributes are automatically created and assigned to their initial values.

You should use class attributes to define properties that should have the same value for every class instance and you must use instance attributes for properties that vary from one instance to another.

Now that we have a **Car** class, let's create some cars!

Instantiating an Object in Python

Creating a new object from a class is called instantiating an object. Consider the previous simpler version of our **Car** class:

```
>>> class Car()
...     pass
```

You can instantiate a new **Car** object by typing the name of the class, followed by opening and closing parentheses:

```
>>> Car()
<__main__.Car object at 0x106702d30>
```

You now have a new **Car** object at `0x106702d30`. This string of letters and numbers is a memory address that indicates where the **Car** object is stored in your computer's memory. Note that the address you see on your screen will be different.

Now, instantiate a second **Car** object:

```
>>> Car()
<__main__.Car object at 0x0004ccc90>
```

The new **Car** instance is located at a different memory address. That's because it's an entirely new instance and is completely different from the first **Car** object that you instantiated.

Consider the following code snippet:

```
>>> a = Car()
>>> b = Car()
>>> a == b
False
```

In this code, you create two new **Car** objects and assign them to the variables **a** and **b**. When you compare **a** and **b** using the **==** operator, the result is **False**. This is because even though **a** and **b** are both instances of the **Car** class, they represent two distinct objects in memory.

Class and Instance Attributes

Now, consider the **Car** class we created with class attribute **color** and instance attributes `name` and `topSpeed`.

```
>>> class Car:
...     color = "Black"
...     def __init__(self, name, topSpeed):
...         self.name = name
...         self.topSpeed= topSpeed
```

To instantiate objects of this **Car** class, you need to provide values for the `name` and `topSpeed`. If you don't, then Python raises a `TypeError`.

```
>>> Car()
TypeError: __init__() missing 2 required positional arguments: 'name'
and 'topSpeed'
```

To pass arguments to the `name` and `topSpeed` parameters, put values into the parentheses after the class name.

```
>>> c1 = Car("Creta", 200)
>>> c1 = Car("Wagon R", 190)
```

This creates two new **Car** instances. Now, observe that the **Car** class's `.__init__()` method has three parameters, so why are only two arguments passed to it in the example?

The reason is that when you instantiate a **Car** object, Python creates a new instance and passes it to the first parameter of `.__init__().` This essentially removes the `self` parameter, so you only need to worry about the `name` and `topSpeed` parameters.

After you create the **Car** instances, you can access their instance attributes using dot notation:

```
>>> c1.name
'Creta'
>>> c2.topSpeed
190
```

You can access class attributes the same way:

```
>>> c1.color
'Black'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. The values of these attributes ***can*** be changed dynamically:

```
>>> c1.topSpeed= 250
>>> c1.topSpeed
250
>>> c2.color = "Red"
>>> c2.color
'Red'
```

In this example, you change the **`topSpeed`** attribute of the **`c1`** object to **250**. Then you change the **`color`** attribute of the **`c2`** object to `"Red"`.

Note:

- The key takeaway here is such custom objects are **mutable** by default i.e. their states can be modified.
- Further, the value of the class attributes will be the same for all instances of that class **initially**. However, we can modify the values of these class attributes for individual objects.

- For instance, in the given example the value of `color` of `c2` object is now `"Red"`, however the value of `color` for `c1` is still `"Black"`.