# Object Oriented Programming in Python

- Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach.
- An object-oriented paradigm is to design the program using classes and objects. The object is related to real-word entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. It is a widespread technique to solve the problem by creating objects.

- Major principles of object-oriented programming system are given below.
    - Class
    - Object
    - Inheritance
    - Polymorphism
    - Data Abstraction
    - Encapsulation
- **Class -** The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.
- **Object -** The object is an entity that has state and behavior. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc. Everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code. When we define a class, it needs to create an object to allocate the memory.
- **Inheritance -** Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object. By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class. It provides the re-usability of the code.
- **Polymorphism -** Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways.
- **Encapsulation -** Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

- **Data Abstraction -** Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation. Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.
- **Object-oriented vs. Procedure-oriented Programming languages –**

| Object-oriented Programming | Procedural Programming |
|---|---|
| Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| It makes the development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when the project becomes lengthy. |
| It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions. |
| It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way for data binding, so it is less secure. |
| Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Example of procedural languages are: C, Fortran, Pascal, VB etc. |

- **Creating classes in Python –**

```
class MyClass:
  x = 5
print(MyClass)
```

- **Creating classes in Python –**

```
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)
```

- **The __init__() Function –** To understand the meaning of classes we have to understand the built-in __init__() function. All classes have a function called __init__(), which is always executed when the class is being initiated. Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created –

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

- **Creating Object Methods -** Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
p1.myfunc()
```

**NOTE : The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.**

**Example –**

```
class Person:
  def __init__(obj, name, age):
    obj.name = name
    obj.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

- **Destroying Object -** We can delete objects by using the del keyword.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)
p1 = Person("John", 36)
del p1
print(p1)
```

- **Python Constructors -** A constructor is a special type of method (function) which is used to initialize the instance members of the class. In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object. Constructors can be of two types -
    - Parameterized Constructor
    - Non-parameterized Constructor

  Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.
- **Non - Parameterized Constructor –**

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")

    def show(self,name):
        print("Hello",name)

student = Student()
student.show("John")
```

- **Parameterized Constructor –**

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name

    def show(self):
        print("Hello",self.name)

student = Student("John")
student.show()
```

- **Default Constructor –** When we do not include the constructor in the class or forget to declare it, then that becomes the default constructor. It does not perform any task but initializes the objects.

```
class Student:
    roll_num = 101
    name = "Joseph"

    def display(self):
        print(self.roll_num,self.name)

st = Student()
st.display()
```

- **Inheritance -** Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

**Parent class** is the class being inherited from, also called **base class**.
**Child class** is the class that inherits from another class, also called **derived class**.

- **Example –**

```
class Animal:
    def speak(self):
        print("Animal Speaking")

#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")

d = Dog()
d.bark()
d.speak()
```

- **Method Overriding –**

```
class Animal:
    def speak(self):
        print("speaking")
class Dog(Animal):
    def speak(self):
        print("Barking")
d = Dog()
d.speak()
```

- **Example –**

```
class Bank:
    def getroi(self):
        return 10;

class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;

b1 = Bank()
b2 = SBI()
b3 = ICICI()

print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

- **Data Abstraction -** Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

- **Encapsulation -** Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single _ or double __.

- **Example –**

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
```

```
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

- **Polymorphism –** Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

- **Example –**

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```