# Object Oriented Programming Using Python:

## INDEX

1. Introduction to Object Oriented Programming in Python
2. Difference between object and procedural oriented programming
3. What are Classes and Objects
4. Object-Oriented Programming methodologies:
   - Inheritance
   - Polymorphism
   - Encapsulation
   - Abstraction

# Object Oriented Programming Using Python:

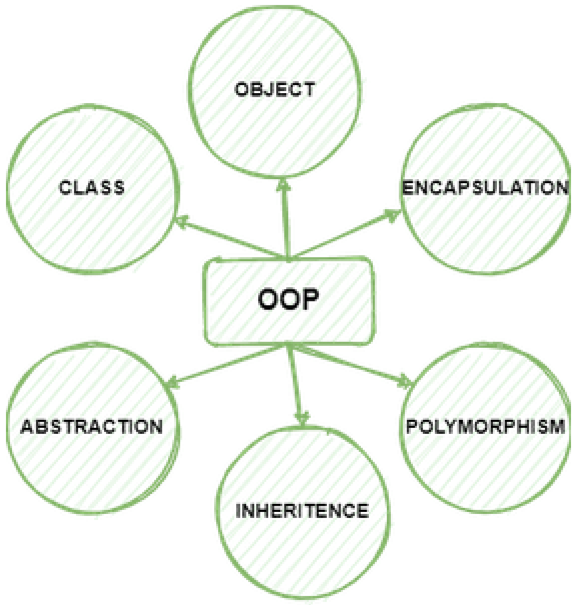
## 1. Introduction to Object Oriented Programming in Python:

Object Oriented Programming is a way of computer programming using the idea of “objects” to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one.

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

## Main Concepts of Object-Oriented Programming (OOPs)

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction



## Difference between object and procedural oriented programming:

| **Procedural Oriented Programming** | **Object-Oriented Programming** |
|---|---|
| In procedural programming, the program is divided into small parts called functions. | In object-oriented programming, the program is divided into small parts called objects. |
| Procedural programming follows a top-down approach. | Object-oriented programming follows a bottom-up approach. |
| There is no access specifier in procedural programming. | Object-oriented programming has access specifiers like private, public, protected, etc. |
| Adding new data and functions is not easy. | Adding new data and function is easy. |
| Procedural programming does not have any proper way of hiding data so it is less secure. | Object-oriented programming provides data hiding so it is more secure. |
| In procedural programming, overloading is not possible. | Overloading is possible in object-oriented programming. |
| In procedural programming, there is no concept of data hiding and inheritance. | In object-oriented programming, the concept of data hiding and inheritance is used. |
| In procedural programming, the function is more important than the data. | In object-oriented programming, data is more important than function. |
| Procedural programming is based on the unreal world. | Object-oriented programming is based on the real world. |
| Procedural programming is used for designing medium-sized programs. | Object-oriented programming is used for designing large and complex programs. |
| Procedural programming uses the concept of procedure abstraction. | Object-oriented programming uses the concept of data abstraction. |
| Code reusability absent in procedural programming, | Code reusability present in object-oriented programming. |
| Examples: C, FORTRAN, Pascal, Basic, etc. | Examples: C++, Java, Python, C#, etc. |

# Object Oriented Programming Using Python:

## Class :

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

### Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  Eg.: Myclass. Myattribute

### Class Definition Syntax:

```
class ClassName:
  # Statement-1

  # Statement-N
```

### Example: Creating an empty Class in Python

```
# Python3 program to
# demonstrate defining
# a class

class Dog:
        pass
```

In the above example, we have created a class named dog using the class keyword.

# Object Oriented Programming Using Python:

## Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a 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.

### An object consists of :

- State: It is represented by the attributes of an object. It also reflects the properties of an object.
- Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- Identity: It gives a unique name to an object and enables one object to interact with other objects.

### To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

**Example: Creating an object:**

```
obj = Dog()
```

This will create an object named obj of the class Dog defined above. Before diving deep into objects and class let us understand some basic keywords that will we used while working with objects and classes.

## The self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

## The __init__ method :

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

Now let us define a class and create some objects using the self and __init__ method.

### Example 1: Creating a class and object with class and instance attributes

```
class Dog:
                      # class attribute
                      attr1 = "mammal"

                      # Instance attribute
                      def __init__(self, name):
                          self.name = name
# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

**Output**

- Rodger is a mammal
- Tommy is also a mammal
- My name is Rodger
- My name is Tommy

## Example 2: Creating Class and objects with methods:

```
class Dog:
    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))
# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

**Output**

- My name is Rodger
- My name is Tommy

## Inheritance:

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

## Types of Inheritance –

1. **Single Inheritance:**
   - ➢ Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

2. **Multilevel Inheritance:**
   - ➢ Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

3. **Hierarchical Inheritance:**
   - ➢ Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

4. **Multiple Inheritance:**
   - ➢ Multiple level inheritance enables one derived class to inherit properties from more than one base class.

## Example: Inheritance in Python:

```
# Python code to demonstrate how parent constructors
# are called.
# parent class
class Person(object):
        # __init__ is known as the constructor
        def __init__(self, name, idnumber):
            self.name = name
            self.idnumber = idnumber

        def display(self):
            print(self.name)
            print(self.idnumber)

        def details(self):
            print("My name is {}".format(self.name))
            print("IdNumber: {}".format(self.idnumber))
# child class
class Employee(Person):
        def __init__(self, name, idnumber, salary, post):
            self.salary = salary
            self.post = post

            # invoking the __init__ of the parent class
            Person.__init__(self, name, idnumber)

        def details(self):
            print("My name is {}".format(self.name))
            print("IdNumber: {}".format(self.idnumber))
            print("Post: {}".format(self.post))

# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")
# calling a function of the class Person using
# its instance
a.display()
a.details()
```

## Output

- Rahul
- 886012
- My name is Rahul
- IdNumber: 886012
- Post: Intern

## Polymorphism:

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

### Example: Polymorphism in Python:

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")
class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")
class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")
obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

### Output

- There are many types of birds.
- Most of the birds can fly but some cannot.
- There are many types of birds.
- Sparrows can fly.
- There are many types of birds.
- Ostriches cannot fly.

## Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object’s variable can only be changed by an object’s method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

### Example: Encapsulation in Python:

```
# Python program to
# demonstrate private members
# Creating a Base class
class Base:
                                def __init__(self):
                                    self.a = "GeeksforGeeks"
                                    self.__c = "GeeksforGeeks"
# Creating a derived class
class Derived(Base):
                                def __init__(self):

                                    # Calling constructor of
                                    # Base class
                                    Base.__init__(self)
                                    print("Calling private member of base class: ")
                                    print(self.__c)
# Driver code
obj1 = Base()
print(obj1.a)
# Uncommenting print(obj1.c) will
# raise an AttributeError
# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
# is called inside derived class
```

## Data Abstraction :

It hides the unnecessary code details from the user. Also,  when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved through creating abstract classes.

## Reference :

- Simplilearn
- Analytical Vidhya
- Greeks for greeks
- sklearn