Python Object Oriented Programming

Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach. One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:
Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Class

A class is a blueprint for the object. The class came into existence when it instantiated.

Suppose a class is a prototype of a building. A building contains all the details about the floor, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

On the other hand, the object is the instance of a class. The process of creating an object can be called as instantiation.

Consider the following example to create a class Employee which contains two fields as Employee id, and name. The class also contains a function display() which is used to display the information of the Employee.

Example

```
1.  class Employee:
2.      id = 10;
3.      name = "ayush"
4.      def display (self):
5.          print(self.id,self.name)
```

Here, the self is used as a reference variable which refers to the current class object.
self is always the first argument in the function definition. However, using self is optional in the function call.

Creating an instance of the class

A class needs to be instantiated if we want to use the class attributes in another class or method. A class can be instantiated by calling the class using the class name.

The following example creates the instance of the class Employee defined in the above example.

Example

```
1.  class Employee:
2.      id = 10;
3.      name = "John"
4.      def display (self):
5.          Print(self.id,self.name)
6.  emp = Employee()
7.  emp.display()
```

Output:

```
ID: 10
Name: ayush
```

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. 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.

Creating the constructor in python

Class functions that begins **with double underscore (__)** are called special functions as they have special meaning.

Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Consider the following example to initialize the Employee class attributes.

Example

```
1. class Employee:
2.     def __init__(self,name,id):
3.         self.id = id;
4.         self.name = name;
```

```
5.     def display (self):
6.         Print(self.id,self.name)
7.  emp1 = Employee("John",101)
8.  emp2 = Employee("David",102)
9.
10.#accessing display() method to print employee 1 information
11.
12.emp1.display();
13.
14.#accessing display() method to print employee 2 information
15.emp2.display();
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Python Non-Parameterized Constructor Example

```
1.      class Student:
2.          # Constructor - non parameterized
3.          def __init__(self):
4.              print("This is non parametrized constructor")
5.          def show(self,name):
6.              print("Hello",name)
7.      student = Student()
8.      student.show("John")
```

Output:

```
This is non parametrized constructor
Hello John
```

Python Parameterized Constructor Example

```
1.      class Student:
2.          # Constructor - parameterized
3.          def __init__(self, name):
4.              print("This is parametrized constructor")
5.              self.name = name
6.          def show(self):
7.              print("Hello",self.name)
8.      student = Student("John")
9.      student.show()
```

Output:

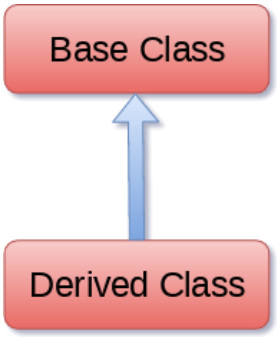
```
This is parametrized constructor
Hello John
```

Python 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.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Example

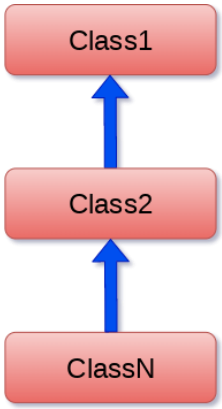
```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



Example

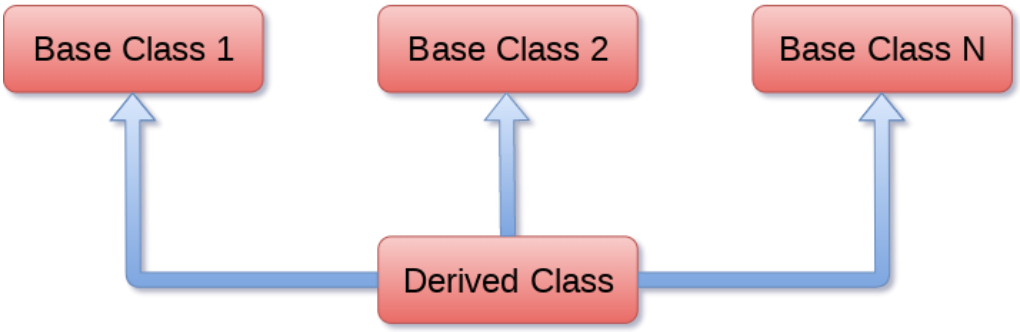
```
1. class Animal:
2.     def speak(self):
3.         print("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. class Dog(Animal):
6.     def bark(self):
7.         print("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. class DogChild(Dog):
10.    def eat(self):
11.        print("Eating bread...")
12.d = DogChild()
13.d.bark()
14.d.speak()
15.d.eat()
```

Output:

```
dog barking
Animal Speaking
Eating bread...
```

Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



Example

```
1. class Calculation1:
2.     def Sum(self,a,b):
3.         return a+b;
4.
5. class Calculation2:
6.     def Multiplication(self,a,b):
7.         return a*b;
8.
9. class Derived(Calculation1,Calculation2):
10.    def Divide(self,a,b):
11.        return a/b;
12.
13.d = Derived()
14.print(d.Sum(10,20))
15.print(d.Multiplication(10,20))
16.print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

Examples:-

Write a Python class named Rectangle constructed by a length and width and a method which will compute the area of a rectangle.

```
class Rectangle():
    def __init__(self, l, w):
        self.length = l
        self.width  = w

    def rectangle_area(self):
        return self.length*self.width

newRectangle = Rectangle(12, 10)
```

```
print(newRectangle.rectangle_area())
```

Write a Python class named Circle constructed by a radius and two methods which will compute the area and the perimeter of a circle.

```
class Circle():
    def __init__(self, r):
        self.radius = r

    def area(self):
        return self.radius**2*3.14

    def perimeter(self):
        return 2*self.radius*3.14

NewCircle = Circle(8)
print(NewCircle.area())
print(NewCircle.perimeter())
```