Python OOPs Concepts

- Python is also an object-oriented language since its beginning.
- It allows us to develop applications using an Object-Oriented approach.
- We can easily create and use classes and objects.
- 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
- ✓ Method
- ✓ 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.

Syntax

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

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.

  Consider the following example.

Example:

```
class car:
   def __init__(self,modelname, year):
      self.modelname = modelname
      self.year = year
   def display(self):
      print(self.modelname,self.year)

c1 = car("Toyota", 2016)
c1.display()
```

Output:

```
Toyota 2016
```

- ❖ In the above example, we have created the class named car, and it has two attributes modelname and year.
- ❖ We have created a c1 object to access the class attribute.
- ❖ The c1 object will allocate memory for these values.

Method

- The method is a function that is associated with an object.
- In Python, a method is not unique to class instances.
- Any object type can have methods.

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.

For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

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

The difference between object-oriented and procedure-oriented programming is given below:

Index	Object-oriented Programming	Procedural Programming
1.	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.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	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.
4.	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.
5.	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

- In Python, a class can be created by using the keyword class, followed by

the class name.

- The syntax to create a class is given below.

Syntax

```
class ClassName:
   #statement_suite
```

- In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__.**
- A class contains a statement suite including fields, constructor, function, etc. definition.

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

```
class Employee:
   id = 10
   name = "Devansh"
   def display (self):
      print(self.id,self.name)
```

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

The self-parameter

- The self-parameter refers to the current instance of the class and accesses the class variables.
- We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

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 syntax to create the instance of the class is given below.

```
<object-name> = <class-name>(<arguments>)
```

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

Example

```
class Employee:
    id = 10
    name = "John"
    def display (self):
        print("ID: %d \nName: %s"%(self.id,self.name))
# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

Output:

```
ID: 10
Name: John
```

In the above code, we have created the Employee class which has two attributes named id and name and assigned value to them. We can observe we have passed the self as parameter in display function. It is used to refer to the same class attribute.

We have created a new instance object named **emp.** By using it, we can access the attributes of the class.

Delete the Object

We can delete the properties of the object or object itself by using the del keyword. Consider the following example.

Example

```
class Employee:
   id = 10
   name = "John"

   def display(self):
      print("ID: %d \nName: %s" % (self.id, self.name))
   # Creating a emp instance of Employee class

emp = Employee()

# Deleting the property of object
del emp.id
# Deleting the object itself
del emp
emp.display()
```

It will through the Attribute error because we have deleted the object **emp**.

Constructors in Python

- Constructors are generally used for instantiating an object.
- The task of constructors 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.

Syntax of constructor declaration :

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

Types of constructors :

- **default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
- **parameterized constructor:** constructor with parameters is known as 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.

Example of default constructor :

```
class sample:
        # default constructor
        def __init__(self):
                self.a = "Hello World"

        # a method for printing data members
        def display(self):
                print(self.a)


# creating object of the class
obj = sample()

# calling the instance method using the object obj
obj.display()
```

Example of the parameterized constructor :

```
class sample:
        first = 0
        second = 0
        answer = 0

        # parameterized constructor
        def __init__(self, f, s):
```

```
            self.first = f
            self.second = s

        def display(self):
            print("First number = " + str(self.first))
            print("Second number = " + str(self.second))
            print("Addition of two numbers = " + str(self.answer))

        def calculate(self):
            self.answer = self.first + self.second

# creating object of the class
# this will invoke parameterized constructor
obj = sample(1000, 2000)

# perform Addition
obj.calculate()

# display result
obj.display()
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm.

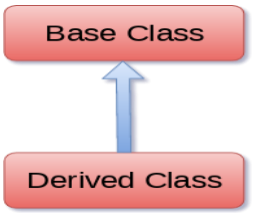
Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

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.



Syntax

```
class derived-class(base class):
   <class-suite>
```

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

Syntax

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):

   <class - suite>
```

Example 1

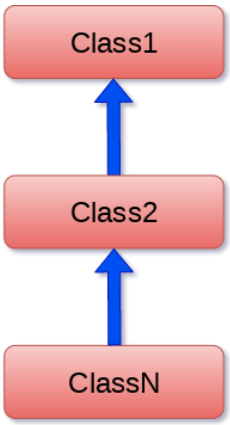
```
class Animal:
   def speak(self):
      print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
   def bark(self):
      print("dog barking")
d = Dog()
d.bark()
d.speak()
```

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.



The syntax of multi-level inheritance is given below.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```

Example

```
class Animal:
  def speak(self):
    print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
  def bark(self):
    print("dog barking")
#The child class Dogchild inherits another child class Dog
ss DogChild(Dog):
```

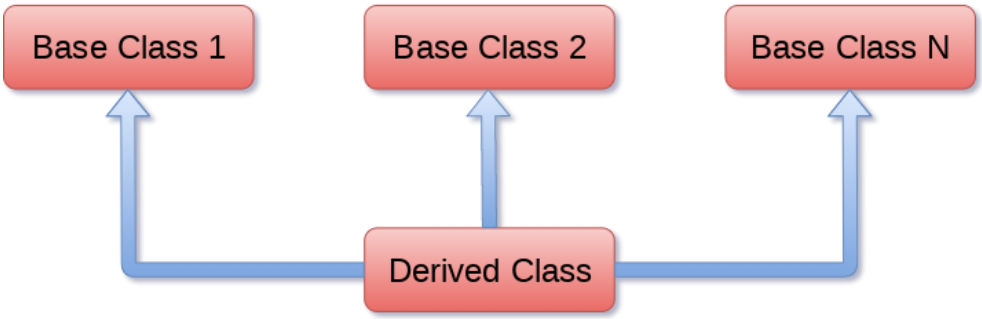
```
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

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

Python Multiple inheritance

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



The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>
```

```
class Derived(Base1, Base2, ..... BaseN):
   <class-suite>
```

Example

```
class Calculation1:
  def Summation(self,a,b):
    return a+b;
class Calculation2:
  def Multiplication(self,a,b):
    return a*b;
class Derived(Calculation1,Calculation2):
  def Divide(self,a,b):
    return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30
200
0.5
```

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes.

It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

Example

```
class Calculation1:
  def Summation(self,a,b):
    return a+b;
class Calculation2:
  def Multiplication(self,a,b):
    return a*b;
```

```
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

Output:

```
True
False
```

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(isinstance(d,Derived))
```

Output:

```
True
```

Method Overriding

- We can provide some specific implementation of the parent class method in our child class.
- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

Example

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

Output:

```
Barking
```

Real Life Example of method overriding

```
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());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

Data Hiding in Python
What is Data Hiding?

- Data hiding is a concept which underlines the hiding of data or information from the user.
- It is one of the key aspects of Object-Oriented programming strategies.
- It includes object details such as data members, internal work.
- Data hiding excludes full data entry to class members and defends object integrity by preventing unintended changes.
- Data hiding also minimizes system complexity for increase robustness by limiting interdependencies between software requirements.
- Data hiding is also known as information hiding. In class, if we declare the data members as private so that no other class can access the data members, then it is a process of hiding data.
- Data hiding in Python is performed using the __ double underscore before done prefix.
- This makes the class members non-public and isolated from the other classes.

Example:

```
class Solution:

    __privateCounter = 0

    def sum(self):
```

```
            self.__privateCounter += 1

            print(self.__privateCounter)

count = Solution()

count.sum()

count.sum()

# Here it will show error because it unable

# to access private member

print(count.__privateCount)
```

To rectify the error, we can access the private member through the class name :

```
class Solution:
    __privateCounter = 0

    def sum(self):
        self.__privateCounter += 1
        print(self.__privateCounter)


count = Solution()
count.sum()
count.sum()

# Here we have accessed the private data
# member through class name.
print(count._Solution__privateCounter)
```