Python: An Object Oriented Language

- Python is an **object-oriented language** since its beginning. It allows us to develop applications using an Object-Oriented approach. In Python, we can easily create and use classes and objects.
- The object is related to real-word entities such as book, house, pencil, etc.
- The Object Oriented Programming System (OOPS) concept focuses on writing the **reusable** code.

OOPS Terminology

Major principles of object-oriented programming system are given below:

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

OOPS Terminology

Class

- A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. It can be seen as a blueprint of an object. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- A class comes into existence when it is instantiated.
- It is a logical entity that has some specific attributes and methods.
- Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details.

OOPS Terminology

Object

- The object is an entity that has state and behavior.
- Everything in Python is an object, and almost everything has attributes and methods.

```
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()
```

- When we define a class, it is needed to create class object/objects to allocate the memory.
- Here, the class named car 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.

OOP Principles

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. A method binds the state of an instance to its behavior.

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.

OOP Principles

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.

OOP Principles

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

S.N.	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

- A class can be created by using the keyword class, followed by the class name.
- 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.

```
class Employee:
    name = 'TCS'
    id = "C123"
    def display (self):
        print(self.id,self.name)
```

- The **self** Parameter

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

Creating Objects

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

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

Instantiating class 'Employee'

```
class Employee:
    id = 'C123'
    name = "tcs"
    def display (self):
        print("ID:  ",self.id," Name:  ",self.name)
# Creating a emp instance of Employee class
emp = Employee()
emp.display()
```

- 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.

Class Variables vs Instance Variables

- **Data member**

  A class variable or instance variable that holds data associated with a class and its objects.

- **Class variable**

  A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.

- **Instance variable**

  A variable that is defined inside a method and belongs only to the current instance of a class.

Class Vs Instance Variables

- Consider the example:

```
class Student:
    name='TCS'  # Class variable
    id='C123'   # Class variable
    def display_details(self):
        print("Name: ",self.id," ID: ",self.name ," age: ",self.age)
s = Student()
s.name = 'amit' # Instance variable
s.id = 112      # Instance
s.age = 23      # Instance
s.display_details()
print(Student.name)
print(Student.id)
```

```
OUTPUT

Name:  amit  id:  112  age:  23
TCS
C123
```

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:
  - 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.
- In C++ or Java, the constructor has the same name as its class, but Python treats constructor definition in a different way.
- In Python, the method **__init__()** simulates the constructor of the class. This method is called when the class is instantiated. It accepts the **self** keyword as a first argument which allows accessing the attributes or method of the class.

Python Constructor

- We can pass any number of arguments at the time of creating the class object, depending upon the __init__() definition.

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("Gagan", 101)
emp2 = Employee("Kavita", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information
emp2.display()
```

Output:
ID: 101 Name: Gagan
ID: 102 Name: Kavita

Non-parameterized Constructor

- The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument. Consider the following example.

```
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("Susheel")
```

Python 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 = 999999 # say a default roll no
    name = "Anonymous"  # anonymous name

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

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

- Note that the display() method refers to the class variables in this case because no instance variables exist.

Multiple Constructors in a Class

- What happen if we declare the two same constructors in the class?

```
class Student:
    def __init__(self):
        print("The First Constructor")
    def __init__(self):
        print("The second contructor")

st = Student()
```

- The object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the st object.
- the object of the class will always call the last constructor if the class has multiple constructors irrespective of whether the constructor is parameterized or non-parameterized
- The constructor overloading is **not allowed** in Python.

Multiple Constructors in a Class

- What happen if we declare the two same constructors in the class?

```
class Student:
      def __init__(self):
        print("The first contructor")
        self.name='LK'
        self.id=184
      def __init__(self,name,id):
        print("The second contructor")
        self.name=name
        self.id=id
      def display(self):
        print("Name: ",self.name)
        print("ID: ",self.id)
#st1 = Student()  This line will cause error
#st1.display() This line will cause error
st2 = Student('ajay',1234)
st2.display()
```

- In this code too, since the second constructor is accepted, it would be mandatory to pass two arguments while object creation.

Multiple Constructors in a Class

- Can you count the no of instances created for a class? **Use class/static variables**

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students: ",Student.count)
```

- Since count is static variable its value is common for all instances of Student.

```
OUTPUT

The number of students: 3
```

Built-In Class Functions

- Give or set the meta-data of the class

SN	Function	Description
1	getattr(obj, name, default)	It is used to access the attribute of the object.
2	setattr(obj, name, value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Built-In Class Functions

- Consider the example:

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

    # creates the object of the class Stude
nt
s = Student("Amit", 101, 22)

# prints the attribute name of the object s

print(getattr(s, 'name'))

# reset the value of attribute age to 23
setattr(s, "age", 23)

# prints the modified value of age
print(getattr(s, 'age'))

# prints true if the student contains the a
ttribute with name id

print(hasattr(s, 'id'))
# deletes the attribute age
delattr(s, 'age')

# this will give an error since the attribu
te age has been deleted
print(s.age)
```

```
OUTPUT

Amit

23

True

AttributeError: 'Student' object has no
attribute 'age'
```

Built-In Class Attributes

- Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.

Built-In Class Attributes

- Consider the example:

```
class Student:
    def __init__(self,name,id,age):
        self.name = name;
        self.id = id;
        self.age = age
    def display_details(self):
        print("Name:%s, ID:%d, age:%d"%(self.name,self.id))

s = Student("Amit",101,22)
print(s.__doc__)
print(s.__dict__)
print(s.__module__)
```

```
OUTPUT

None

{'name': 'Amit', 'id': 101, 'age': 22}

__main__
```