Introduction to Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Example

A person is an object which has certain properties such as height, gender, age, etc. It also has certain methods such as move, talk, and so on.

Features of Object-oriented Programming

Object - This is the basic unit of object-oriented programming. That is both data and function that operate on data are bundled as a unit called an object.

Class - A Class is a blueprint or a template for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

OOP has four basic concepts on which it is totally based. Let's have a look at them individually:

1. **Abstraction:** It refers to, providing only essential information to the outside world and hiding their background details.
   Example - A web server hides how it processes data it receives, the end user just hits the endpoints and gets the data back.

2. **Encapsulation:** Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit. It is also a way of restricting access to certain properties or component.
   Example - class.

3. **Inheritance:** The ability to create a new class from an existing class is called Inheritance. Using inheritance, we can create a Child class from a Parent class such that it inherits the properties and methods of the parent class and can have its own additional properties and methods.
   Example - If we have a class Vehicle that has properties like Color, Price, etc, we can create 2 classes like Bike and Car from it that have those 2 properties and additional properties that are specialized for them like a car has numberOfWindows while a bike cannot. Same is applicable to methods.

4. **Polymorphism:** The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Classes and Objects

Object - An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

Class – A class can be defined as a blueprint or template from which we can create an individual object. Class doesn't consume any space.

- A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Syntax: Creating a Class

```
class <Class_Name>:
      Statement-1
      Statement-2
           …
      Statement-n
```

Syntax: Creating an Object for a Class

```
Object_name = Class_Name( )
```

Syntax: For accessing a Class member through the class object.

```
Object_name.Class_member_name
```

Example:

```
class Train:
    name = "Venkatadri Express"          # Class Variable
    no = 51057                           # Class Variable
    price = 1500                         # Class Variable


T1 = Train()                           # Object Creation / Instatiation
print("Type of Train: ",type(Train))
print("Type of T1: ",type(T1))
print("T1 Name: ",T1.name)
print("T1 no: ",T1.no)
print("price: ",T1.price)
```

Output:

```
Type of Train:  <class 'type'>
Type of T1:  <class '__main__.Train'>
T1 Name:  Venkatadri Express
T1 no:  51057
price:  1500
```

Class Methods (Instance Methods) - Class methods are similar to methods (Functions) must have the first argument named as **self**.

- **self** is the first argument that is added to the beginning of the parameter list.
- The **self** argument refers to the object itself.
- **self** argument will pass all the class variables to class methods

Syntax

```
class  <class_name>:
      list of class variables (attributes)
      def method_name1(self, arguement1, argument2, …):
            statement-1
            statement-2
                …
            Statement-n
```

Example

```
class Train:
    # class attributes
    name=input("Enter Train Name: ")
    no=input("Enter Train Number: ")
    price=int(input("Enter Price: "))

    # class methods
    def disp(self):
        print("inside the class method..")
    def status(self,source,destination):
        print("Train starts from ",source," and ends at ",destination)

# Creating an object for Train Class
T1=Train()

# Accessing variables
print("T1 Name: ",T1.name)
print("T1 no: ",T1.no)
print("price: ",T1.price)

# Calling a Class methods
T1.disp()
T1.status('Hyderabad','Tirupati')
```

Output:

```
Enter Train Name: Venkatadri Express
Enter Train Number: 12345
Enter Price: 1500
T1 Name:  Venkatadri Express
T1 no:  12345
price:  1500
inside the class method..
Train starts from  Hyderabad  and ends at  Tirupati
```

Example – Self argument

```
class ABC:
    A=10
    B=20
    def display(self):
        print(self.A, " and ", self.B)
        #print(A ," and ", B)    # Error - A and B undefined

a1=ABC()
a1.display()
```

Output:

```
10  and  20
```

The _ _init_ _() Method (The Class Constructor)

- Constructor is a special method in a class, this constructor method is automatically executed when an object of a class is created.
- The name of the constructor should be _ _init_ _() (prefixed as well as suffixed by double underscores)
- The method is useful to initialize the variables of the class object.
- __init__() method returns “None”. We cannot return a value from the constructor.

Example

```
class ABC:
    def disp(self):
        print("inside the class method..")

    def __init__(self):
        print("I am a Constructor method for a ABC class.")


a1=ABC()
a1.disp()
```

Output:

```
I am a Constructor method for a ABC class.
inside the class method..
```

Example

```
class ABC:
    A=10
    def __init__(self,A):
        print("class Variable A=",self.A)   # class Variable A= 10
        print("instance variable A=",A)     # instance variable A= 2

a1=ABC(2)
```

The _ _del_ _() method

- The __del__() method is just an opposite of __init__(). The __del__() method is automatically called/invoked when an object is going out of scope.
- This is the time when object will no longer be used and its occupied resources are returned back to the system so that they can be reused as and when required. You can also explicitly do the same using the del keyword.

Example:

```
class ABC:
    def __init__(self):
        print("__init__() is invoked automatically when an object is created")
        print("started..")

    def __del__(self):
        print("__del__() is invoked automatically when an object is going out of scope")
        print("ending..")

obj1 = ABC()
obj2 = ABC()

print("Bye..")
```

Output

```
__init__() is invoked automatically when an object is created
started..
__del__() is invoked automatically when an object is going out of scope
ending..
__init__() is invoked automatically when an object is created
started..
__del__() is invoked automatically when an object is going out of scope
ending..
Bye..
```

Class Variables & Object Variables

Variables are of two types- class variables and object variables. Class variables are owned by the class and object variables are owned by each object.

- If a class has n objects, then there will be n separate copies of the object variable as each object will have its own object variable.
- The object variable is not shared between objects.
- A change made to the object variable by one object will not be reflected in other objects. If a class has one class variable, then there will be one copy only for that variable. All the objects of that class will share the class variable.
- Since there exists a single copy of the class variable, any change made to the class variable by an object will be reflected to all other objects.

Example

```
class ABC:
    x=0         #Class Variable
    def __init__(self,y):
        ABC.x +=1      # Accessing Class Var x
        print("class var x value: ",ABC.x)
        print("object Variable y value: ",y)     # Object Var y

obj1=ABC(10)
obj2=ABC(20)
obj3=ABC(30)
```

Output:

```
class var x value:  1
object Variable y value:  10
class var x value:  2
object Variable y value:  20
class var x value:  3
object Variable y value:  30
```

Example

```
class ABC:
    x=10
    y=20
    def addition(self):
        c = self.x + self.y
        print("sum = ", c)

obj1 = ABC()
obj1.addition()
```

Output:

```
sum =  30
```

Note: x & y are class variables and c is object variable

Calling a Class Method from another Class Method

Example:

```
class ABC:
    sum=0
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def display(self):
        print("sum  = ", self.sum)
    def addition(self):
        self.sum=self.x+self.y
        self.display()  #Calling a Class Method from Another Class Method

obj1=ABC(10,20)
obj1.addition()
```

Output:

```
sum  =  30
```

Example: Write a Python program that reads student details: rollno,name and marks in six subjects and computes average marks. Use classes and objects.

```
class Student:
    def __init__(self,r,n,m):
        print("roll no = ", r)
        print("name = " ,n)
        tot=0
        for i in range(len(m)):
            print(m[i])
            tot=tot+m[i]
        print("total= ", tot)
        avg=tot/6
        print("average= ",avg)


roll=input("enter roll : ")
name=input("enter name : ")
marks=[]
print("enter marks : ")
for i in range(1,7):
    marks.append(int(input()))
print(marks)

s1=Student(roll,name,marks)
```

Output:

```
enter roll : 1208
enter name : Eeswar
enter marks :
49
39
40
60
90
57

roll no =  1208
name =  Eeswar
49
39
40
60
90
57
total=  335
average=  55.833333333333336
```

Public and Private Data Members

Public variables: Public variables are those variables that are defined in the class and can be accessed from anywhere in the program, of course using the dot operator.

Private variables: Private variables are those variables that are defined in the class with a double underscore prefix (_ _). These variables can be accessed only from within the class and from nowhere outside the class.

Example:

```
class PrivateMem:
    x=10    # Public Variable
    __y=20  # Private Variable
    def fun1(self):
        print("x in fun1 = ", self.x)
        print("y in fun1 = ", self.__y)

obj1=PrivateMem()
obj1.fun1()
print("x out of class = ",obj1.x)
# print("y out of class = ",obj1.__y)     # Error
```

Output:

```
x in fun1 =  10
y in fun1 =  20
x out of class =  10
```

Private Methods

Like private attributes, we cannot use private method from anywhere outside the class.

However, if it is very necessary to access them from outside the class, then they are accessed with a small difference. A private method can be accessed using the object name as well as the class name from outside the class.

The syntax for accessing the private method

objectname._classname_ _privatemethodname( )

Example

```
class PrivateMem:
    x=10    # Public Variable
    __y=20  # Private Variable
    def fun1(self):
        print("x in fun1 = ", self.x)
        print("y in fun1 = ", self.__y)
    def __fun2(self):      # Private Method
        print("This is a private method")

obj1=PrivateMem()
obj1.fun1()
print("x out of class = ",obj1.x)
# print("y out of class = ",obj1.__y)     # Error
obj1._PrivateMem__fun2()        # Calling a private method
```

Output:

```
x in fun1 =  10
y in fun1 =  20
x out of class =  10
This is a private method
```

Python In-built class functions

The in-built functions defined in the class are described in the following table.

S.No.	Function	Description
1	getattr(obj,name[,default])	It is used to access the attribute of the object. **Note:** Consider name of variable as a string **Note:** getattr(obj, 'var') is same as writing obj.var.
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.

Example

```
class Person:
    name=""
    age=0
    def __init__(self,n,a):
        self.name=n
        self.age=a

obj1=Person("Eeswar",10)
print(getattr(obj1,'name'))
print(getattr(obj1,'height','-1'))

setattr(obj1,'name',"Banala")
print(obj1.name)

print(hasattr(obj1,'name'))

delattr(obj1,'name')
print("obj1 name=", obj1.name)
```

Output:

```
Eeswar
-1
Banala
True
obj1 name=
```

Garbage Collection (Destroying Objects)

Python performs automatic garbage collection. This means that it deletes all the objects (built-in types or user defined like class objects) automatically that are no longer needed and that have gone out of scope to free the memory space. The process by which Python periodically reclaims unwanted memory is known as garbage collection.

Python's garbage collector runs in the background during program execution. It immediately takes action (of reclaiming memory) as soon as an object's reference count reaches zero.

Example

```
var1 = 10      # Create object var1
var2 = var1    # Increase ref. count  of var1 - object assignment
var3 = [var2]  # Increase ref. count  of var1 - object used in a list
var2 = 50      # Decrease ref. count of var1 - reassignment
var3[0] = -1   # Decrease ref. count  of var1 - removal from list
del var1       # Decrease ref. count  of var1 - object deleted
```

Class methods

Class methods are little different from these ordinary methods. First, they are called by a class (not by instance of the class). Second, the first argument of the classmethod is **cls** not the **self**.

Class methods are widely used for factory methods, which instantiate an instance of a class, using different parameters than those usually passed to the class constructor.

Syntax:

```
class  <class_name>:
     @classmethod(cls, arg1, arg2,…):
          Statement-1
          Statement-2
                …
```

Example:

```
class Shape:
    def rectArea(self,l,b):
        return(l*b)
    @classmethod
    def sqArea(cls,s):
        return(s*s)

s=Shape()
print("Area of Rectangle = ",s.rectArea(2,4))
#calling a classmethod
print("Area of Square = ",Shape.sqArea(3))
```

Output:

```
Area of Rectangle =  8
Area of Square =  9
```

Static methods

Any functionality that belongs to a class, but that does not require the object is placed in the static method. Static methods are similar to class methods. The only difference is that a static method does not receive any additional arguments. They are just like normal functions that belong to a class.

A static method does not use the self variable and is defined using a built-in function named staticmethod. Python has a handy syntax, called a decorator, to make it easier to apply the staticmethod function to the method function definition. The syntax for using the staticmethod decorator.

Syntax:

```
class  <class_name>:
     @staticmethod(arg1, arg2,…):
          Statement-1
          Statement-2
                    …
```

Example:

```
class Shape:
    def rectArea(self,l,b):
        return(l*b)
    @classmethod
    def sqArea(cls,s):
        return(s*s)
    @staticmethod
    def CircleArea(r):
        return(3.14*r*r)
s=Shape()
print("Area of Rectangle = ",s.rectArea(2,4))
#calling a class method
print("Area of Square = ",Shape.sqArea(3))
#calling a static method
print("Area of Circle = ",Shape.CircleArea(2))
```

Output:

```
Area of Rectangle =  8
Area of Square =  9
Area of Circle =  12.56
```