Object-Oriented Programming (OOPS)

Main Concepts of Object-Oriented Programming (OOPs)

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

⬆ Back to Top

Ques. What is Class?

- The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods.
- To define a class in Python, you can use the class keyword, followed by the class name and a colon. Inside the class, an **init** method has to be defined with def. This is the initializer that you can later use to instantiate objects. It's similar to a constructor in Java. **init** must always be present! It takes one argument: self, which refers to the object itself. Inside the method, the pass keyword is used as of now, because Python expects you to type something there.

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

- **Instantiating object**

```
mohit = ClassName()
print(mohit)
```

Example

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Example

```
class Dog:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("bark bark!")

    def doginfo(self):
        print(self.name + " is " + str(self.age) + " year(s) old.")

ozzy = Dog("Ozzy", 2)
skippy = Dog("Skippy", 12)
filou = Dog("Filou", 8)

ozzy.bark()
skippy.doginfo()
filou.doginfo()

Output:-
bark bark!
Skippy is 12 year(s) old.
Filou is 8 year(s) old.
```

- we have shown two ways of accessing the values of those attributes. One, by directly using the class name and the other by using an object(class instance). Assigning a class to a variable is known as object instantiation.

```
class Scaler:
    Course1 = 'Python'
    Course2 = 'C++'
    Course3 = 'Java'
# Accessing the values of the attributes
print(Scaler.Course1)
print(Scaler.Course3)
# Accessing through object instantiation.
obj= Scaler()
print(obj.Course2)

Output:-
Python
Java
C++
```

- If we change the value of the attribute using the class name, then it would change across all the instances of that class. While if we change the value of an attribute using class instance(object instantiation), it would only change the value of the attribute in that instance only.

```
class Scaler:
    Course = 'Python'
# Changing value using Class Name
Scaler.Course = 'Machine Learning'
obj= Scaler()
print(obj.Course)
# Changing value using Class Instance
obj.Course = 'AI'
print(obj.Course)   # Value will change in this instance only
print('Using class instance would not reflect the changes to other instances')
print(Scaler.Course) # Value haven't changed
obj2= Scaler()
print(obj2.Course)   # Value haven't changed

Output:-
Machine Learning
AI
Using class instance would not reflect the changes to other instances
Machine Learning
Machine Learning
```

⬆ Back to Top

Ques. What is 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.
- **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.

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

c1 = car("Toyota", 2016)
print(c1.a)             # Accessing Object's variables
c1.display()            # Accessing Object's functions

output:-
mohit
Toyota 2016
```

- Modifying Object's properties

```
class Scaler:
  a = 10

# Declaring an object
obj1 = Scaler()
print(obj1.a)

#Modifying value
obj1.a = 200
print("After modifying the object properties")
print(obj1.a)

Output:-
10
After modifying the object properties
200
```

- 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** 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

Ques. Delete the Object?

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

Ques. What is the use of self in Python?

Self is used to represent the instance of the class. With this keyword, you can access the attributes and methods of the class in python. It binds the attributes with the given arguments. self is used in different places and often thought to be a keyword. But unlike in C++, self is not a keyword in Python.

Ques. What is init?

init is a contructor method in Python and is automatically called to allocate memory when a new object/instance is created. All classes have a **init** method associated with them. It helps in distinguishing methods and attributes of a class from local variables.

```
class Student:
   def __init__(self, fname, lname, age, section):
       self.firstname = fname
       self.lastname = lname
       self.age = age
       self.section = section
# creating a new object
stu1 = Student("Sara", "Ansh", 22, "A2")
```

Ques. What is pass in Python?

- The pass keyword represents a null operation in Python.
  Without the pass statement in the following code, we may run into some errors during code execution.

```
def myEmptyFunc():
   # do nothing
   pass
myEmptyFunc()    # nothing happens
## Without the pass keyword
# File "<stdin>", line 3
# IndentationError: expected an indented block
```

Ques. What is break, continue and pass in Python?

- **Break:-** The break statement terminates the loop immediately and the control flows to the statement after the body of the loop.
- **Continue:-** The continue statement terminates the current iteration of the statement, skips the rest of the code in the current iteration and the control flows to the next iteration of the loop.
- **Pass:-** As explained above, the pass keyword in Python is generally used to fill up empty blocks and is similar to an empty statement represented by a semi-colon in languages such as Java, C++, Javascript, etc.

Python Access Modifiers

- **Public Member:** Accessible anywhere from outside the class.
- **Private Member:** Accessible only within the class.
- **Protected Member:** Accessible within the class and it's sub-classes.

```
#defining class Student
class Student:
    #constructor is defined
    def __init__(self, name, age, salary):
        self.age = age              # public Attribute
        self._name = name           # protected Attribute
        self.__salary = salary      # private Attribute

    def _funName(self):             # protected method
        pass

    def __funName(self):            # private method
        pass

# object creation
obj = Student('Mohit',53434)
```

- **Public Access Modifier**
- By default, all the variables and member functions of a class are public in a python program.

```
class Employee:
    # constructor
    def __init__(self, name, sal):
        self.name = name;
        self.sal = sal;
obj = Employee('mohit',555)
print(obj.name)
print(obj.sal)

Output:-
Mohit
555
```

- **protected Access Modifier** adding a prefix _(single underscore) to a variable name makes it protected.

```
class Employee:
    # constructor
    def __init__(self, name, sal):
        self._name = name;    # protected attribute
        self._sal = sal;      # protected attribute

obj = Employee('mohit',15)
print(obj._name)

Output:-
Mohit

# Example2:-
```

```
class Employee:
    # constructor
    def __init__(self, name, sal):
        self._name = name;   # protected attribute
        self._sal = sal;     # protected attribute

class HR(Employee):
    def task(self):
        print ("We manage Employees")


hrEmp = HR("Captain", 10000);
print(hrEmp._sal)
print(hrEmp.task())

Output:-
10000
We manage Employees
```

- **private Access Modifier** While the addition of prefix __(double underscore) results in a member variable or function becoming private.

```
class Person:
    def __init__(self, name, age, height):
        self.name     = name   # public
        self._age     = age    # protected
        self.__height = height # private

p1 = Person("John", 20, 170)

print(p1.name)         # public: can be accessed
print(p1._age)         # protected: can be accessed but not advised
# print(p1.__height)   # private: will give AttributeError
```

Ques What is Inheritance?

- Inheritance is the ability to ‘inherit’ features or attributes from already written classes into newer classes we make.
- Inheritance is the capability of one class to derive or inherit the properties from another class.
- **Benefits:-** It provides the reusability of a code. We don’t have to write the same code again and again.

```
class house:
    height = “572ft”                 # predefined attributes of a class
    architect = “John Doe”

    def display_height(self):
        print("This house is " + self.height)     # methods for displaying height
    def display_architect(self):
        print (“The architect is “ + self.architect)  # methods for displaying
```

```
architect

# Driver Code
Bungalow = house()                     # object creation
print(Bungalow.height)                     # accessing class attributes
Bungalow.display_height()                  # calling method through object
Bungalow.display_architect()               # calling method through object

Output:-
572ft
This building is 572ft
The architect is John Doe
```

Types Of Inheritance?

- **Single Inheritance:-** Single Inheritance is the simplest form of inheritance where a single child class is derived from a single parent class.

```
class parent:                  # parent class
    def func1:
        print("Hello Parent")

class child(parent):           # child class
    def func2:                 # we include the parent class
        print("Hello Child")   # as an argument in the child
                               # class

# Driver Code
test = child()                 # object created
test.func1()                   # parent method called via child object
test.func2()                   # child method called

Output:-
Hello Parent
Hello Child
```

- **Multiple Inheritance:-** In multiple inheritance, a single child class is inherited from two or more parent classes. This means the child class has access to all the methods and attributes of all the parent classes.

```
class parent1:                     # first parent class
    def func1:
        print("Hello Parent1")

class parent2:                     # second parent class
    def func2:
        print("Hello Parent2")

class parent3:                     # third parent class
    def func2:                     # the function name is same as parent2
```

```
        print(“Hello Parent3”)

class child(parent1, parent2, parent3):     # child class
    def func3:                          # we include the parent classes
        print(“Hello Child”)            # as an argument comma separated

# Driver Code
test = child()          # object created
test.func1()            # parent1 method called via child
test.func2()            # parent2 method called via child instead of parent3
test.func3()            # child method called

Output:-
Hello Parent1
Hello Parent2
Hello Child
```

- **Multi-Level Inheritance:-** In multilevel inheritance, we go beyond just a parent-child relation. We introduce grandchildren, great-grandchildren, grandparents.

```
class grandparent:                      # first level
    def func1:
        print(“Hello Grandparent”)

class parent(grandparent):              # second level
    def func2:
        print(“Hello Parent”)

class child(parent):                    # third level
    def func3:
        print(“Hello Child”)


# Driver Code
test = child()                          # object created
test.func1()                            # 3rd level calls 1st level
test.func2()                            # 3rd level calls 2nd level
test.func3()                            # 3rd level calls 3rd level

Output:-
Hello Grandparent
Hello Parent
Hello Child
```

- **Hierarchical Inheritance:-** there are multiple derived child classes from a single parent class.

```
class parent:                            # parent class
    def func1:
        print(“Hello Parent”)
```

```
class child1(parent):                # first child class
    def func2:
        print(“Hello Child1”)


class child2(parent):                # second child class
    def func3:
        print(“Hello Child2”)


# Driver Code
test1 = child1()                      # objects created
test2 = child2()

test1.func1()                        # child1 calling parent method
test1.func2()                        # child1 calling its own method

test2.func1()                        # child2 calling parent method
test2.func3()                        # child2 calling its own method

Output:-
Hello Parent
Hello Child1
Hello Parent
Hello Child2
```

- **Hybrid Inheritance:-** Hybrid Inheritance is the mixture of two or more different types of inheritance. Here we can have many to many relations between parent classes and child classes with multiple levels.

```
class parent1:                             # first parent class
    def func1:
        print(“Hello Parent”)


class parent2:                             # second parent class
    def func2:
        print(“Hello Parent”)

class child1(parent1):                     # first child class
    def func3:
        print(“Hello Child1”)


class child2(child1, parent2):             # second child class
    def func4:
        print(“Hello Child2”)


# Driver Code
test1 = child1()                           # object created
```

```
test2 = child2()

test1.func1()                       # child1 calling parent1 method
test1.func2()                       # child1 calling its own method

test2.func1()                       # child2 calling parent1 method
test2.func2()                       # child2 calling parent2 method
test2.func3()                       # child2 calling child1 method
test2.func4()                       # child2 calling its own method

Output:-
Hello Parent1
Hello Child1

Hello Parent1
Hello Parent2
Hello Child1
Hello Child2
```

Ques. What is str and repr?

- The repr() method returns a string containing a printable representation of an object. The repr() function calls the underlying **repr**() function of the object.
- The **str** method returns a string representation of an object that is human-readable while the
- **repr** method returns a string representation of an object that is machine-readable.

Ques. What is MRO(Method Resolution Order) / Diamond Problam?

- MRO is a concept used in inheritance.
- If a class is derived from more then one class, then it is called multiple inheritance.
- In Python, the MRO is from bottom to top and left to right. This means that, first, the method is searched in the class of the object. If it's not found, it is searched in the immediate super class. In the case of multiple super classes, it is searched left to right, in the order by which was declared by the developer.

```
class father():
    def showF(self):
        print("fathe class method")

class mother():
    def showM(self):
        print("mother class method")
class son(father, mother):
    def showS(self):
        print("son class method")

obj = son()
obj.showS()
obj.showF()
obj.showM()
```

```
Output:-
son class method
fathe class method
mother class method
-------------------------------------------------------------------

# Example:-
class father():
    def display(self):
        print("father class method")

class mother():
    def display(self):
        print("mother class method")


class son(mother,father):                  # left to right
    def showS(self):
        print("son class method")

obj = son()
obj.display()

Output:-
mother class method

++++++++++++++++++++
class son(father, mother):                  # left to right
    def showS(self):
        print("son class method")

obj = son()
obj.display()

Output:- father class method
--------------------------------------------------------------------------
# Using Constructor
class father():
    def __init__(self):
        super().__init__()      # Calling Parent Class Constructor
        print("father class Constructor")
    def showF(self):
        print("father class method")

class mother():
    def __init__(self):
        super().__init__()      # Calling Parent Class Constructor
        print("mother class Constructor")
    def showM(self):
        print("mother class method")
class son(father, mother):                  # left to right
    def __init__(self):
        super().__init__()      # Calling Parent Class Constructor 1st Wala
        print("son class Constructor")
    def showS(self):
        print("son class method")

obj = son()

Output:-
mother class Constructor
father class Constructor
son class Constructor
```

Ques. What is 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.

Ques. Types Of Methods In Python?

There are three types of methods in Python.

1. Instance Methods.
2. Class Methods.
3. Static Methods.

- **Instance Method**
- when we create classes in python. If we want to print an instance variable or instance method we must create an object of that required class.
- If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.
- As we are working with instance variables we use self keyword.
- **Note:-** Instance variables are used with instance methods.

```
class Student:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def avg(self):
        return (self.a + self.b) / 2

s1 = Student(10, 20)
print( s1.avg() )

Output:- 15.0
```

- **Class Method**
- classmethod() function returns a class method as output for the given function.

- If we want to create a class method we must use **@classmethod** decorator and **cls** as a parameter for that function.

```
class Student:
    name = 'Student'
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @classmethod
    def info(cls):
        return cls.name

print(Student.info())

Output:- Student
```

- **Static Method**
- A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.
- A static method in python must be created by decorating it with **@staticmethod**

```
class Student:
    name = 'Student'
    def __init__(self, a, b):
        self.a = a
        self.b = b

    @staticmethod
    def info():
        return "This is a student class"

print(Student.info())

Output:- This a student class
```

```
class MethodTypes:

    name = "Ragnar"

    def instanceMethod(self):
        # Creates an instance atribute through keyword self
        self.lastname = "Lothbrock"
        print(self.name)
        print(self.lastname)

    @classmethod
    def classMethod(cls):
        # Access a class atribute through keyword cls
        cls.name = "Lagertha"
        print(cls.name)

    @staticmethod
    def staticMethod():
        print("This is a static method")

# Creates an instance of the class
m = MethodTypes()
# Calls instance method
m.instanceMethod()


MethodTypes.classMethod()
MethodTypes.staticMethod()

Output:-
Ragnar
Lothbrock
Lagertha
This is a static method
```

Ques. What is a constructor in Python?

- In Python the **init**() method is called the constructor and is always called when an object is created.

Types of constructors:

1. **default constructor/Non-Parameterized 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.

```

class Shot:
    # Creating default constructor
    def __init__(self):
        print("This is a default constructor")
        self.title = "Constructor in Python"

    # a method to display the data member
    def display_message(self):
        print(self.title)

# creating object of the class
s1_obj = Shot()
s1_obj.display_message()

Output:-
This is a default constructor
Constructor in Python
```

2. **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.

```
class Triangle:
    # parameterized constructor
    def __init__(self, b, h):
        self.base = b
        self.height = h


    def areaOfTriangle(self):
        self.area = self.base * self.height * 0.5
        print("Area of triangle: " + str(self.area))

# creating object of the class
# this will invoke parameterized constructor
obj = Triangle(5, 14)
obj.areaOfTriangle()

Output:-
Area of triangle: 35.0
```

d. Why used constructor? e. Advantage of constructor

Ques. What is Polymorphism?

- When one task is performed by different ways i.e known as polymorphism.
- A child class inherits all the methods from the parent class.
- we understand that one task can be performed in different ways.
- Polymorphism is ability to use function & method in different ways.

Types of Polymorphism?

- Polymorphism could be static and dynamic both. Overloading is static polymorphism while, overriding is dynamic polymorphism.

1. **Duck Typing**
2. **Compile time polymorphism (Static) - Method Overloading:-** Overloading is defining functions/methods that have same signatures with different parameters in the same class.

```
class VIP:
    def vsp(self, x=None, y=None):
        if x==None and y==None:
            print("this is python polymorphism")
        elif x!=None and y==None:
            f=1
            for i in range(1, x+1):
                f= f*i
            print(f)
        else:
            print("add:",x+y)

obj = VIP()

obj.vsp()
obj.vsp(5)
obj.vsp(5,5)

output:-
this is python polymorphism
120
add: 10
```

3. **Runtime time polymorphism (Dynamic) - Method Overriding:-**
    1. Overriding is redefining parent class functions/methods in child class with same signature. So, basically the purpose of overriding is to change the behavior of your parent class method.
    2. function wo wala call hota jiska hamne obj banya hota hai.

```
class A:
    def vsp(self):
        print("Hiiiiii")
class B(A):
    def vsp(self):
        print("Hello")

obj = A()
obj.vsp()

Output:-
Hiiiiii
```

4. **operator Overloading**

```
class A:
    def vsp(self,x):
        self.x = x
    def __add__(self,o):
        return self.x+o.x

obj1 = A()
obj1.vsp(10)

obj2 = A()
obj2.vsp(20)

print(obj1+obj2)
```

```
Output:- 30
```

Polymorphism with Class Methods

```
class Monkey:
    def color(self):
        print("The monkey is yellow coloured!")

    def eats(self):
        print("The monkey eats bananas!")


class Rabbit:
    def color(self):
        print("The rabbit is white coloured!")

    def eats(self):
        print("The rabbit eats carrots!")


mon = Monkey()
rab = Rabbit()
for animal in (mon, rab):
    animal.color()
    animal.eats()

Output:-
The monkey is yellow coloured!
The monkey eats bananas!
The rabbit is white coloured!
The rabbit eats carrots!
```

Polymorphism with Inheritance

```
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.
```

Ques. What is Encapsulation?

- Binding(or wrapping) code and data together into a single unit is known as encapsulation. One object is encapsulated from another object.

Code of getter & setter in Encapsulation

```
class Library:
    def __init__(self, id, name):
        self.bookId = id
        self.bookName = name

    def setBookName(self, newBookName): #setters method to set the book name
        self.bookName = newBookName

    def getBookName(self): #getters method to get the book name
        print(f"The name of book is {self.bookName}")


book = Library(101,"The Witchers")
book.getBookName()
book.setBookName("The Witchers Returns")
book.getBookName()

Output:-
The name of book is The Witchers
The name of book is The Witchers Returns
```

Ques. What is Abstract Class?

- We cannot create an abstract class in Python directly. However, Python does provide a module that allows us to define abstract classes. The module we can use to create an abstract class in Python is abc(abstract base class) module. **Rule**
- PVM can not create objects of an abstract class (abstract class ka hum object nahi bna sakte hai).
- It is not neccessary to declare all methods abstract in a abstract class.
- Abstract class can have abstract method and concreate method.
- If there is any abstract method in a class, that class must be abstract.
- The abstract methods of an abstract class must be defined in its child class/subclass.

Ques. When use abstratc class?

- We use abstract class when there are some common feature shered by all the objects as they are. **Example:-**

```
# gun is common features
# Defence Forse
#   Gun:- Ak 47
#   Area:- --
--------------------------
#   army:- Gun Ak 47,    Area:- Land
#   Air Force:- Gun Ak 47,    Area:- Sky
#   Navy:-      Gun Ak 47,    Area:- Sea
```

```
from abc import ABC
class <Abstract_Class_Name>(ABC):
```

```
from abc import ABC, abstractmethod
class Father(ABC):
    @abstractmethod
    def disp(self):
        pass

    def show(self):
        print("concreate class")

class child(Father):
    def disp(self):
        print("child class")
        print("defining abstrat class")
c = child()
c.disp()
c.show()
```

Ques. What is Abstract Method?

- To define an abstract **method** we use the **@abstractmethod** decorator of the abc module.

```
from abc import ABC, abstractmethod
class DemoAbstractClass(ABC):
    @abstractmethod
    def abstract_method_name(self):
        Pass
```

Ques. What is Concrete Method?

- A concreate method is a method whose action is defined in the abstract class itself.

```
from abc import ABC, abstractmethod
class Father(ABC):
    @abstractmethod
    def disp(self):
        pass                            # method without body
    def show(self):
        print("concrete method")        # concrete Method / method with body
```