# Python's Object Oriented Programming (OOPs)

## What is Class:

- In Python every thing is an object. To create objects we required some Model or Plan or Blue print, which is nothing but class.
- We can write a class to represent properties (attributes) and actions (behaviour) of object.

- Properties can be represented by variables
- Actions can be represented by Methods.

- Hence class contains both variables and methods.

## How to Define a class?

We can define a class by using class keyword.

**Syntax:**

```
class className:
  ''' documenttation string '''
   variables:instance variables,static and local variables
   methods: instance methods,static methods,class methods
```

Documentation string represents description of the class. Within the class doc string is always optional. We can get doc string by using the following 2 ways.

1. print(classname.__doc__)
2. help(classname)

**Example:**

```
1)  class Student:
2)      ''' This is student class with required data'''
3)  print(Student.__doc__)
4)  help(Student)
```

Within the Python class we can represent data by using variables.
There are 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

Within the Python class, we can represent operations by using methods. The following are various types of allowed methods

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

**Example for class:**

```
1)  class Student:
2)      '''Developed by durga for python demo'''
3)      def __init__(self):
4)          self.name='durga'
5)          self.age=40
6)          self.marks=80
7)
8)      def talk(self):
9)          print("Hello I am :",self.name)
10)         print("My Age is:",self.age)
11)         print("My Marks are:",self.marks)
```

# What is Object:

Pysical existence of a class is nothing but object. We can create any number of objects for a class.

**Syntax to create object:** referencevariable = classname()

**Example:** s = Student()

# What is Reference Variable:

The variable which can be used to refer object is called reference variable.
By using reference variable, we can access properties and methods of object.

**Program:** Write a Python program to create a Student class and Creates an object to it. Call the method talk() to display student details

```
1)  class Student:
2)
3)      def __init__(self,name,rollno,marks):
4)          self.name=name
5)          self.rollno=rollno
6)          self.marks=marks
7)
8)      def talk(self):
9)          print("Hello My Name is:",self.name)
10)         print("My Rollno is:",self.rollno)
11)         print("My Marks are:",self.marks)
12)
```

```
13) s1=Student("Durga",101,80)
14) s1.talk()
```

**Output:**

```
D:\durgaclasses>py test.py
Hello My Name is: Durga
My Rollno is: 101
My Marks are: 80
```

# Self variable:

self is the default variable which is always pointing to current object (like this keyword in Java)

By using self we can access instance variables and instance methods of object.

**Note:**

1. self should be first parameter inside constructor
   def __init__(self):

2. self should be first parameter inside instance methods
   def talk(self):

# Constructor Concept:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be exeucted only once.
- Constructor can take atleast one argument(atleast self)

- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

**Example:**

```
1) def __init__(self,name,rollno,marks):
2)         self.name=name
3)         self.rollno=rollno
4)         self.marks=marks
```

**Program to demonistrate constructor will execute only once per object:**

```
1) class Test:
2)
3)     def __init__(self):
4)         print("Constructor exeuction...")
5)
```

```
    def m1(self):
        print("Method execution...")

t1=Test()
t2=Test()
t3=Test()
t1.m1()
```

**Output**

```
Constructor exeuction...
Constructor exeuction...
Constructor exeuction...
Method execution...
```

**Program:**

```
class Student:

    ''' This is student class with required data'''
    def __init__(self,x,y,z):
        self.name=x
        self.rollno=y
        self.marks=z

    def display(self):
        print("Student Name:{}\nRollno:{} \nMarks:{}".format(self.name,self.rollno,self.marks)
    )

s1=Student("Durga",101,80)
s1.display()
s2=Student("Sunny",102,100)
s2.display()
```

**Output**

```
Student Name:Durga
Rollno:101
Marks:80
Student Name:Sunny
Rollno:102
Marks:100
```

# Differences between Methods and Constructors:

| Method | Constructor |
|---|---|
| 1. Name of method can be any name | 1. Constructor name should be always __init__ |
| 2. Method will be executed if we call that method | 2. Constructor will be executed automatically at the time of object creation. |
| 3. Per object, method can be called any number of times. | 3. Per object, Constructor will be executed only once |
| 4. Inside method we can write business logic | 4. Inside Constructor we have to declare and initialize instance variables |

# Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

# 1. Instance Variables:

If the value of a variable is varied from object to object, then such type of variables are called instance variables.

For every object a separate copy of instance variables will be created.

## Where we can declare Instance variables:

1. Inside Constructor by using self variable
2. Inside Instance Method by using self variable
3. Outside of the class by using object reference variable

# 1. Inside Constructor by using self variable:

We can declare instance variables inside a constructor by using self keyword. Once we creates object, automatically these variables will be added to the object.

**Example:**

```
1)  class Employee:
2)
3)    def __init__(self):
4)       self.eno=100
5)       self.ename='Durga'
6)       self.esal=10000
7)
8)  e=Employee()
```

```
9)  print(e.__dict__)
```

**Output:** {'eno': 100, 'ename': 'Durga', 'esal': 10000}

## 2. Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable. If any instance variable declared inside instance method, that instance variable will be added once we call taht method.

**Example:**

```
1)  class Test:
2)
3)      def __init__(self):
4)          self.a=10
5)          self.b=20
6)
7)      def m1(self):
8)          self.c=30
9)
10) t=Test()
11) t.m1()
12) print(t.__dict__)
```

**Output**
{'a': 10, 'b': 20, 'c': 30}

## 3. Outside of the class by using object reference variable:

We can also add instance variables outside of a class to a particular object.

```
1)  class Test:
2)
3)      def __init__(self):
4)          self.a=10
5)          self.b=20
6)
7)      def m1(self):
8)          self.c=30
9)
10) t=Test()
11) t.m1()
12) t.d=40
13) print(t.__dict__)
```

**Output** {'a': 10, 'b': 20, 'c': 30, 'd': 40}

# How to access Instance variables:

**We can access instance variables with in the class by using self variable and outside of the class by using object reference.**

```
1)  class Test:
2)
3)      def __init__(self):
4)          self.a=10
5)          self.b=20
6)
7)      def display(self):
8)          print(self.a)
9)          print(self.b)
10)
11) t=Test()
12) t.display()
13) print(t.a,t.b)
```

**Output**

```
10
20
10 20
```

# How to delete instance variable from the object:

**1. Within a class we can delete instance variable as follows**

```
del self.variableName
```

**2. From outside of class we can delete instance variables as follows**

```
del objectreference.variableName
```

**Example:**

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)          self.c=30
6)          self.d=40
7)      def m1(self):
8)          del self.d
9)
10) t=Test()
11) print(t.__dict__)
```

```
12) t.m1()
13) print(t.__dict__)
14) del t.c
15) print(t.__dict__)
```

**Output**

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

**Note:** The instance variables which are deleted from one object,will not be deleted from other objects.

**Example:**

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)          self.c=30
6)          self.d=40
7)
8)
9)  t1=Test()
10) t2=Test()
11) del t1.a
12) print(t1.__dict__)
13) print(t2.__dict__)
```

**Output**

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

**Example:**

```
1)  class Test:
2)      def __init__(self):
3)          self.a=10
4)          self.b=20
5)
6)  t1=Test()
7)  t1.a=888
8)  t1.b=999
9)  t2=Test()
10) print('t1:',t1.a,t1.b)
```

```
11) print('t2:',t2.a,t2.b)
```

**Output**

**t1: 888 999**
**t2: 10 20**

# 1. Static variables:

**If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.**

**For total class only one copy of static variable will be created and shared by all objects of that class.**

**We can access static variables either by class name or by object reference. But recommended to use class name.**

# Instance Variable vs Static Variable:

**Note: In the case of instance variables for every object a seperate copy will be created,but in the case of static variables for total class only one copy will be created and shared by every object of that class.**

```
1)  class Test:
2)      x=10
3)      def __init__(self):
4)          self.y=20
5)
6)  t1=Test()
7)  t2=Test()
8)  print('t1:',t1.x,t1.y)
9)  print('t2:',t2.x,t2.y)
10) Test.x=888
11) t1.y=999
12) print('t1:',t1.x,t1.y)
13) print('t2:',t2.x,t2.y)
```

**Output**

**t1: 10 20**
**t2: 10 20**
**t1: 888 999**
**t2: 888 20**

# Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside classmethod by using either class name or cls variable
5. Inside static method by using class name

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          Test.b=20
5)      def m1(self):
6)          Test.c=30
7)      @classmethod
8)      def m2(cls):
9)          cls.d1=40
10)         Test.d2=400
11)     @staticmethod
12)     def m3():
13)         Test.e=50
14) print(Test.__dict__)
15) t=Test()
16) print(Test.__dict__)
17) t.m1()
18) print(Test.__dict__)
19) Test.m2()
20) print(Test.__dict__)
21) Test.m3()
22) print(Test.__dict__)
23) Test.f=60
24) print(Test.__dict__)
```

# How to access static variables:

1. inside constructor: by using either self or classname
2. inside instance method:  by using either self or classname
3. inside class method: by using either cls variable or classname
4. inside static method: by using classname
5. From outside of class: by using either object reference or classnmae

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          print(self.a)
5)          print(Test.a)
6)      def m1(self):
7)          print(self.a)
```

```
        print(Test.a)
    @classmethod
    def m2(cls):
        print(cls.a)
        print(Test.a)
    @staticmethod
    def m3():
        print(Test.a)
t=Test()
print(Test.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

## Where we can modify the value of static variable:

**Anywhere either with in the class or outside of class we can modify by using classname.**
**But inside class method, by using cls variable.**

**Example:**

```
class Test:
    a=777
    @classmethod
    def m1(cls):
        cls.a=888
    @staticmethod
    def m2():
        Test.a=999
print(Test.a)
Test.m1()
print(Test.a)
Test.m2()
print(Test.a)
```

**Output**

```
777
888
999
```

*****

# If we change the value of static variable by using either self or object reference variable:

**If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed,just a new instance variable with that name will be added to that particular object.**

**Example 1:**

```
1)  class Test:
2)      a=10
3)      def  m1(self):
4)          self.a=888
5)  t1=Test()
6)  t1.m1()
7)  print(Test.a)
8)  print(t1.a)
```

**Output**

```
10
888
```

**Example:**

```
1)  class Test:
2)      x=10
3)      def __init__(self):
4)          self.y=20
5)
6)  t1=Test()
7)  t2=Test()
8)  print('t1:',t1.x,t1.y)
9)  print('t2:',t2.x,t2.y)
10) t1.x=888
11) t1.y=999
12) print('t1:',t1.x,t1.y)
13) print('t2:',t2.x,t2.y)
```

**Output**

```
t1: 10 20
t2: 10 20
t1: 888 999
t2: 10 20
```

**Example:**

```
1)  class Test:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)  t1=Test()
6)  t2=Test()
7)  Test.a=888
8)  t1.b=999
9)  print(t1.a,t1.b)
10) print(t2.a,t2.b)
```

**Output**

888 999
888 20

---

```
1)   class Test:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      def m1(self):
6)          self.a=888
7)          self.b=999
8)
9)  t1=Test()
10) t2=Test()
11) t1.m1()
12) print(t1.a,t1.b)
13)  print(t2.a,t2.b)
```

**Output**

888 999
10 20

**Example:**

```
1)   class Test:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)      @classmethod
6)      def m1(cls):
7)          cls.a=888
8)          cls.b=999
9)
10) t1=Test()
11) t2=Test()
```

```
t1.m1()
print(t1.a,t1.b)
print(t2.a,t2.b)
print(Test.a,Test.b)
```

**Output**

```
888 20
888 20
888 999
```

# How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

```
del   classname.variablename
```

But inside classmethod we can also use cls variable

```
del   cls.variablename
```

```
class Test:
    a=10
    @classmethod
    def m1(cls):
        del cls.a
Test.m1()
print(Test.__dict__)
```

**Example:**

```
class Test:
    a=10
    def __init__(self):
        Test.b=20
        del Test.a
    def m1(self):
        Test.c=30
        del Test.b
    @classmethod
    def m2(cls):
        cls.d=40
        del Test.c
    @staticmethod
    def m3():
        Test.e=50
        del Test.d
print(Test.__dict__)
t=Test()
```

```
print(Test.__dict__)
t.m1()
print(Test.__dict__)
Test.m2()
print(Test.__dict__)
Test.m3()
print(Test.__dict__)
Test.f=60
print(Test.__dict__)
del Test.e
print(Test.__dict__)
```

****

**Note:** By using object reference variable/self we can read static variables, but we cannot modify or delete.

If we are trying to modify, then a new instance variable will be added to that particular object.
t1.a = 70

If we are trying to delete then we will get error.

**Example:**

```
class Test:
    a=10

t1=Test()
del t1.a     ===>AttributeError: a
```

We can modify or delete static variables only by using classname or cls variable.

```
import sys
class Customer:
    ''' Customer class with bank operations.. '''
    bankname='DURGABANK'
    def __init__(self,name,balance=0.0):
        self.name=name
        self.balance=balance
    def deposit(self,amt):
        self.balance=self.balance+amt
        print('Balance after deposit:',self.balance)
    def withdraw(self,amt):
        if amt>self.balance:
            print('Insufficient Funds..cannot perform this operation')
            sys.exit()
        self.balance=self.balance-amt
        print('Balance after withdraw:',self.balance)

print('Welcome to',Customer.bankname)
```

```
name=input('Enter Your Name:')
c=Customer(name)
while True:
    print('d-Deposit \nw-Withdraw \ne-exit')
    option=input('Choose your option:')
    if option=='d' or option=='D':
        amt=float(input('Enter amount:'))
        c.deposit(amt)
    elif option=='w' or option=='W':
        amt=float(input('Enter amount:'))
        c.withdraw(amt)
    elif option=='e' or option=='E':
        print('Thanks for Banking')
        sys.exit()
    else:
        print('Invalid option..Plz choose valid option')
```

**output:**

```
D:\durga_classes>py test.py
Welcome to DURGABANK
Enter Your Name:Durga
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:10000
Balance after deposit: 10000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:20000
Balance after deposit: 30000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:w
Enter amount:2000
Balance after withdraw: 28000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:r
Invalid option..Plz choose valid option
d-Deposit
w-Withdraw
e-exit
Choose your option:e
Thanks for Banking
```

# Local variables:

**Sometimes to meet temporary requirements of programmer,we can declare variables inside a method directly,such type of variables are called local variable or temporary variables.**

**Local variables will be created at the time of method execution and destroyed once method completes.**

**Local variables of a method cannot be accessed from outside of method.**

**Example:**

```
1)  class Test:
2)      def m1(self):
3)          a=1000
4)          print(a)
5)      def m2(self):
6)          b=2000
7)          print(b)
8)  t=Test()
9)  t.m1()
10) t.m2()
```

**Output**

```
1000
2000
```

**Example 2:**

```
1)  class Test:
2)      def m1(self):
3)          a=1000
4)          print(a)
5)      def m2(self):
6)          b=2000
7)          print(a)    #NameError: name 'a' is not defined
8)          print(b)
9)  t=Test()
10) t.m1()
11) t.m2()
```

# Types of Methods:

**Inside Python class 3 types of methods are allowed**

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

# 1. Instance Methods:

**Inside method implementation if we are using instance variables then such type of methods are called instance methods.**
**Inside instance method declaration,we have to pass self variable.**

```
def m1(self):
```

**By using self variable inside method we can able to access instance variables.**

**Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.**

```
1)  class Student:
2)      def __init__(self,name,marks):
3)          self.name=name
4)          self.marks=marks
5)      def display(self):
6)          print('Hi',self.name)
7)          print('Your Marks are:',self.marks)
8)      def grade(self):
9)          if self.marks>=60:
10)             print('You got First Grade')
11)         elif self.marks>=50:
12)             print('Yout got Second Grade')
13)         elif self.marks>=35:
14)             print('You got Third Grade')
15)         else:
16)             print('You are Failed')
17) n=int(input('Enter number of students:'))
18) for i in range(n):
19)     name=input('Enter Name:')
20)     marks=int(input('Enter Marks:'))
21)     s= Student(name,marks)
22)     s.display()
23)     s.grade()
24)     print()
```

**ouput:**

```
D:\durga_classes>py test.py
Enter number of students:2
Enter Name:Durga
Enter Marks:90
Hi Durga
Your Marks are: 90
You got First Grade

Enter Name:Ravi
Enter Marks:12
Hi Ravi
Your Marks are: 12
You are Failed
```

## Setter and Getter Methods:

**We can set and get the values of instance variables by using getter and setter methods.**

## Setter Method:

**setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.**

**syntax:**

```
def  setVariable(self,variable):
      self.variable=variable
```

**Example:**

```
def  setName(self,name):
      self.name=name
```

## Getter Method:

**Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.**

**syntax:**

```
def  getVariable(self):
     return self.variable
```

**Example:**

```
def  getName(self):
      return self.name
```

**Demo Program:**

```
1)  class Student:
2)      def setName(self,name):
3)          self.name=name
4)
5)      def getName(self):
6)          return self.name
7)
8)      def setMarks(self,marks):
9)          self.marks=marks
10)
11)     def getMarks(self):
12)         return self.marks
13)
14) n=int(input('Enter number of students:'))
15) for i in range(n):
16)     s=Student()
17)     name=input('Enter Name:')
18)     s.setName(name)
19)     marks=int(input('Enter Marks:'))
20)     s.setMarks(marks)
21)
22)     print('Hi',s.getName())
23)     print('Your Marks are:',s.getMarks())
24)     print()
```

**output:**

```
D:\python_classes>py test.py
Enter number of students:2
Enter Name:Durga
Enter Marks:100
Hi Durga
Your Marks are: 100

Enter Name:Ravi
Enter Marks:80
Hi Ravi
Your Marks are: 80
```

# 2. Class Methods:

**Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.**

**We can declare class method explicitly by using @classmethod decorator.**
**For class method we should provide cls variable at the time of declaration**

We can call classmethod by using classname or object reference variable.

**Demo Program:**

```
1)  class Animal:
2)      legs=4
3)      @classmethod
4)      def walk(cls,name):
5)          print('{} walks with {} legs...'.format(name,cls.legs))
6)  Animal.walk('Dog')
7)  Animal.walk('Cat')
```

**Output**

```
D:\python_classes>py test.py
Dog walks with 4 legs...
Cat walks with 4 legs...
```

**Program to track the number of objects created for a class:**

```
1)  class Test:
2)      count=0
3)      def __init__(self):
4)          Test.count =Test.count+1
5)      @classmethod
6)      def noOfObjects(cls):
7)          print('The number of objects created for test class:',cls.count)
8)
9)  t1=Test()
10) t2=Test()
11) Test.noOfObjects()
12) t3=Test()
13) t4=Test()
14) t5=Test()
15) Test.noOfObjects()
```

# 3. Static Methods:

In general these methods are general utility methods.
Inside these methods we won't use any instance or class variables.
Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator
We can access static methods by using classname or object reference

```
1)  class DurgaMath:
2)
3)      @staticmethod
4)      def add(x,y):
```

```
        print('The Sum:',x+y)

    @staticmethod
    def product(x,y):
        print('The Product:',x*y)

    @staticmethod
    def average(x,y):
        print('The average:',(x+y)/2)

DurgaMath.add(10,20)
DurgaMath.product(10,20)
DurgaMath.average(10,20)
```

**Output**

**The Sum: 30**
**The Product: 200**
**The average: 15.0**

**Note: In general we can use only instance and static methods.Inside static method we can access class level variables by using class name.**

**class methods are most rarely used methods in python.**

# Passing members of one class to another class:

**We can access members of one class inside another class.**

```
class Employee:
    def __init__(self,eno,ename,esal):
        self.eno=eno
        self.ename=ename
        self.esal=esal
    def display(self):
        print('Employee Number:',self.eno)
        print('Employee Name:',self.ename)
        print('Employee Salary:',self.esal)
class Test:
    def modify(emp):
        emp.esal=emp.esal+10000
        emp.display()
e=Employee(100,'Durga',10000)
Test.modify(e)
```

**Output**

```
D:\python_classes>py test.py
Employee Number: 100
Employee Name: Durga
```

Employee Salary: 20000

In the above application, Employee class members are available to Test class.

# Inner classes:

Sometimes we can declare a class inside another class,such type of classes are called inner classes.

Without existing one type of object if there is no chance of existing another type of object,then we should go for inner classes.

**Example:** Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

```
class Car:
    .....
    class Engine:
    ......
```

**Example:** Without existing university object there is no chance of existing Department object

```
class University:
    .....
    class Department:
    ......
```

**eg3:**

Without existing Human there is no chance of existin Head. Hence Head should be part of Human.

```
class Human:
   class Head:
```

**Note:** Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

**Demo Program-1:**

```
1)  class Outer:
2)      def __init__(self):
3)          print("outer class object creation")
4)      class Inner:
5)          def __init__(self):
6)              print("inner class object creation")
7)          def m1(self):
8)              print("inner class method")
9)  o=Outer()
10) i=o.Inner()
11) i.m1()
```

**Output**

```
outer class object creation
inner class object creation
inner class method
```

**Note:** The following are various possible syntaxes for calling inner class method

```
1.
o=Outer()
i=o.Inner()
i.m1()

2.
i=Outer().Inner()
i.m1()

3. Outer().Inner().m1()
```

**Demo Program-2:**

```
1)  class Person:
2)      def __init__(self):
3)          self.name='durga'
4)          self.db=self.Dob()
5)      def display(self):
6)          print('Name:',self.name)
7)      class Dob:
8)          def __init__(self):
9)              self.dd=10
10)             self.mm=5
11)             self.yy=1947
12)         def display(self):
13)             print('Dob={}/{}/{}'.format(self.dd,self.mm,self.yy))
14) p=Person()
15) p.display()
16) x=p.db
17) x.display()
```

**Output**

```
Name: durga
Dob=10/5/1947
```

**Demo Program-3:**

Inside a class we can declare any number of inner classes.

```
1)  class Human:
2)
3)      def __init__(self):
```

```
4)          self.name = 'Sunny'
5)          self.head = self.Head()
6)          self.brain = self.Brain()
7)      def display(self):
8)          print("Hello..",self.name)
9)
10)     class Head:
11)         def talk(self):
12)             print('Talking...')
13)
14)     class Brain:
15)         def think(self):
16)             print('Thinking...')
17)
18) h=Human()
19) h.display()
20) h.head.talk()
21) h.brain.think()
```

**Output**

**Hello.. Sunny**
**Talking...**
**Thinking...**

# Garbage Collection:

**In old languages like C++, programmer is responsible for both creation and destruction of objects.Usually programmer taking very much care while creating object, but neglecting destruction of useless objects. Because of his neglectance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.**

**But in Python, We have some assistant which is always running in the background to destroy useless objects.Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.**

**Hence the main objective of Garbage Collector is to destroy useless objects.**

**If an object does not have any reference variable then that object eligible for Garbage Collection.**

## How to enable and disable Garbage Collector in our program:

**By default Gargbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.**

### 1. gc.isenabled()

**Returns True if GC enabled**

**2. gc.disable()**

To disable GC explicitly

**3. gc.enable()**

To enable GC explicitly

**Example:**

```
1) import gc
2) print(gc.isenabled())
3) gc.disable()
4) print(gc.isenabled())
5) gc.enable()
6) print(gc.isenabled())
```

**Output**

True
False
True

# Destructors:

Destructor is a special method and the name should be __del__
Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).
Once destructor execution completed then Garbage Collector automatically destroys that object.

**Note:** The job of destructor is not to destroy object and it is just to perform clean up activities.

**Example:**

```
1) import time
2) class Test:
3)     def __init__(self):
4)         print("Object Initialization...")
5)     def __del__(self):
6)         print("Fulfilling Last Wish and performing clean up activities...")
7)
8) t1=Test()
9) t1=None
10) time.sleep(5)
11) print("End of application")
```

**Output**

Object Initialization...
Fulfilling Last Wish and performing clean up activities...
End of application

**Note:**
**If the object does not contain any reference variable then only it is eligible fo GC. ie if the reference count is zero then only object eligible for GC**

**Example:**

```
1)  import time
2)  class Test:
3)      def __init__(self):
4)          print("Constructor Execution...")
5)      def __del__(self):
6)          print("Destructor Execution...")
7)
8)  t1=Test()
9)  t2=t1
10) t3=t2
11) del t1
12) time.sleep(5)
13) print("object not yet destroyed after deleting t1")
14) del t2
15) time.sleep(5)
16) print("object not yet destroyed even after deleting t2")
17) print("I am trying to delete last reference variable...")
18) del t3
```

**Example:**

```
1)  import time
2)  class Test:
3)      def __init__(self):
4)          print("Constructor Execution...")
5)      def __del__(self):
6)          print("Destructor Execution...")
7)
8)  list=[Test(),Test(),Test()]
9)  del list
10) time.sleep(5)
11) print("End of application")
```

**Output**

```
Constructor Execution...
Constructor Execution...
Constructor Execution...
Destructor Execution...
Destructor Execution...
Destructor Execution...
End of application
```

# How to find the number of references of an object:

**sys module contains getrefcount() function for this purpose.**

**sys.getrefcount(objectreference)**

**Example:**

```
1)  import sys
2)  class Test:
3)      pass
4)  t1=Test()
5)  t2=t1
6)  t3=t1
7)  t4=t1
8)  print(sys.getrefcount(t1))
```

**Output** 5

**Note:** **For every object, Python internally maintains one default reference variable self.**