# OOP's Part - 1

## 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:**

```
class Student:
    '''' This is student class with required data'''
print(Student.__doc__)
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:**

```
class Student:
  '''Developed by durga for python demo'''
  def __init__(self):
    self.name='durga'
    self.age=40
    self.marks=80

  def talk(self):
    print("Hello I am :",self.name)
    print("My Age is:",self.age)
    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

```
class Student:

  def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks

```

```
    def talk(self):
        print("Hello My Name is:",self.name)
        print("My Rollno is:",self.rollno)
        print("My Marks are:",self.marks)

s1=Student("Durga",101,80)
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:**

```
def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks
```

**Program to demonistrate Constructor will execute only once per Object:**

```
class Test:

    def __init__(self):
        print("Constructor exeuction...")

    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.

```
class Employee:

  def __init__(self):
    self.eno=100
    self.ename='Durga'
```

```
        self.esal=10000

    e=Employee()
    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.

```
class Test:

    def __init__(self):
        self.a=10
        self.b=20

    def m1(self):
        self.c=30

t=Test()
t.m1()
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.

```
class Test:

    def __init__(self):
        self.a=10
        self.b=20
    def m1(self):
        self.c=30

t=Test()
t.m1()
t.d=40
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.

```
class Test:

  def __init__(self):
    self.a=10
    self.b=20
 def display(self):
 print(self.a)
 print(self.b)

t=Test()
t.display()
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

```
class Test:
  def __init__(self):
    self.a=10
    self.b=20
    self.c=30
    self.d=40
  def m1(self):
    del self.d

t=Test()
print(t.__dict__)
t.m1()
print(t.__dict__)
del t.c
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.

```python
class Test:
   def __init__(self):
      self.a=10
      self.b=20
      self.c=30
      self.d=40

t1=Test()
t2=Test()
del t1.a
print(t1.__dict__)
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.

```python
class Test:
   def __init__(self):
      self.a=10
      self.b=20

t1=Test()
t1.a=888
t1.b=999
t2=Test()
print('t1:',t1.a,t1.b)
print('t2:',t2.a,t2.b)
```

**Output**

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

## 2) 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 types 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.

```
class Test:
    x=10
    def __init__(self):
        self.y=20

t1=Test()
t2=Test()
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
Test.x=888
t1.y=999
print('t1:',t1.x,t1.y)
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

```
class Test:
    a=10
    def __init__(self):
        Test.b=20
    def m1(self):
        Test.c=30
    @classmethod
    def m2(cls):
        cls.d1=40
        Test.d2=400
    @staticmethod
    def m3():
        Test.e=50
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__)
```

## 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 classname

```
class Test:
    a=10
    def __init__(self):
        print(self.a)
        print(Test.a)
    def m1(self):
        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.

```
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.

```
class Test:
    a=10
    def  m1(self):
        self.a=888
```

```
t1=Test()
t1.m1()
print(Test.a)
print(t1.a)
```

**Output**

```
10
888
```

```
class Test:
    x=10
    def __init__(self):
        self.y=20

t1=Test()
t2=Test()
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
t1.x=888
t1.y=999
print('t1:',t1.x,t1.y)
print('t2:',t2.x,t2.y)
```

**Output**

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

```
class Test:
    a=10
    def __init__(self):
        self.b=20
t1=Test()
t2=Test()
Test.a=888
t1.b=999
print(t1.a,t1.b)
print(t2.a,t2.b)
```

**Output**

```
888 999
888 20
```

```
class Test:
    a=10
    def __init__(self):
        self.b=20
    def m1(self):
        self.a=888
        self.b=999

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

<u>**Output**</u>

```
888 999
10 20
```

```
class Test:
    a=10
    def __init__(self):
        self.b=20
    @classmethod
    def m1(cls):
        cls.a=888
        cls.b=999

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

<u>**Output**</u>

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

## How to Delete Static Variables of a Class:

1) We can delete static variables from anywhere by using the following syntax
   **del classname.variablename**

2) 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__)
```

****<u>Note:</u>

- 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.

<u>**Example:**</u>

```
 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
```

## 3) 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.

```
class Test:
  def m1(self):
    a=1000
    print(a)
  def m2(self):
    b=2000
    print(b)
t=Test()
t.m1()
t.m2()
```

**Output**

```
1000
2000
```

```
class Test:
  def m1(self):
    a=1000
    print(a)
  def m2(self):
    b=2000
    print(a)   #NameError: name 'a' is not defined
    print(b)
t=Test()
t.m1()
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.

```
class Student:
  def __init__(self,name,marks):
    self.name=name
    self.marks=marks
  def display(self):
    print('Hi',self.name)
    print('Your Marks are:',self.marks)
  def grade(self):
    if self.marks>=60:
      print('You got First Grade')
    elif self.marks>=50:
      print('Yout got Second Grade')
    elif self.marks>=35:
      print('You got Third Grade')
    else:
      print('You are Failed')
n=int(input('Enter number of students:'))
for i in range(n):
  name=input('Enter Name:')
  marks=int(input('Enter Marks:'))
  s= Student(name,marks)
  s.display()
  s.grade()
  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
```

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

    def getName(self):
        return self.name

    def setMarks(self,marks):
        self.marks=marks

    def getMarks(self):
        return self.marks

n=int(input('Enter number of students:'))
for i in range(n):
    s=Student()
    name=input('Enter Name:')
    s.setName(name)
    marks=int(input('Enter Marks:'))
    s.setMarks(marks)

    print('Hi',s.getName())
    print('Your Marks are:',s.getMarks())
    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.

```
class Animal:
    lEgs=4
    @classmethod
    def walk(cls,name):
        print('{} walks with {} lEgs...'.format(name,cls.lEgs))
Animal.walk('Dog')
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:

```
class Test:
    count=0
    def __init__(self):
        Test.count =Test.count+1
    @classmethod
    def noOfObjects(cls):
        print('The number of objects created for test class:',cls.count)

t1=Test()
t2=Test()
Test.noOfObjects()
t3=Test()
t4=Test()
t5=Test()
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

```
class DurgaMath:

    @staticmethod
    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:
    ......
```

**Example:** 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:**

```
class Outer:
    def __init__(self):
        print("outer class object creation")
    class Inner:
        def __init__(self):
            print("inner class object creation")
        def m1(self):
            print("inner class method")
o=Outer()
i=o.Inner()
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:**

```
class Person:
    def __init__(self):
        self.name='durga'
        self.db=self.Dob()
    def display(self):
        print('Name:',self.name)
    class Dob:
        def __init__(self):
            self.dd=10
            self.mm=5
            self.yy=1947
        def display(self):
            print('Dob={}/{}/{}'.format(self.dd,self.mm,self.yy))
```

```
p=Person()
p.display()
x=p.db
x.display()
```

**Output**

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

**Demo Program-3:**

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

```
class Human:

  def __init__(self):
    self.name = 'Sunny'
    self.head = self.Head()
    self.brain = self.Brain()
  def display(self):
    print("Hello..",self.name)

  class Head:
    def talk(self):
      print('Talking...')

  class Brain:
    def think(self):
      print('Thinking...')

h=Human()
h.display()
h.head.talk()
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

```
import gc
print(gc.isenabled())
gc.disable()
print(gc.isenabled())
gc.enable()
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.

```
import time
class Test:
    def __init__(self):
        print("Object Initialization...")
    def __del__(self):
        print("Fulfilling Last Wish and performing clean up activities...")

t1=Test()
t1=None
time.sleep(5)
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.

```
import time
class Test:
    def __init__(self):
        print("Constructor Execution...")
    def __del__(self):
        print("Destructor Execution...")

t1=Test()
t2=t1
t3=t2
del t1
time.sleep(5)
print("object not yet destroyed after deleting t1")
del t2
```

```
time.sleep(5)
print("object not yet destroyed even after deleting t2")
print("I am trying to delete last reference variable...")
        del t3
```

**Example:**

```
import time
class Test:
    def __init__(self):
        print("Constructor Execution...")
    def __del__(self):
        print("Destructor Execution...")

list=[Test(),Test(),Test()]
del list
time.sleep(5)
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)

```
import sys
class Test:
    pass
t1=Test()
t2=t1
t3=t1
t4=t1
print(sys.getrefcount(t1))
```

**Output** 5

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

# OOP's Part - 2

## Agenda

- ❖ Inheritance
- ❖ Has-A Relationship
- ❖ IS-A Relationship
- ❖ IS-A vs HAS-A Relationship
- ❖ Composition vs Aggregation

- ❖ Types of Inheritance
    - Single Inheritance
    - Multi Level Inheritance
    - Hierarchical Inheritance
    - Multiple Inheritance
    - Hybrid Inheritance
    - Cyclic Inheritance

- ❖ Method Resolution Order (MRO)
- ❖ super() Method

## Using Members of One Class inside Another Class:

**We can use members of one class inside another class by using the following ways**

1) **By Composition (Has-A Relationship)**
2) **By Inheritance (IS-A Relationship)**

## 1) By Composition (Has-A Relationship):

- **By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).**
- **The main advantage of Has-A Relationship is Code Reusability.**

**Demo Program-1:**

```
class Engine:
    a=10
    def __init__(self):
        self.b=20
    def m1(self):
            print('Engine Specific Functionality')
    class Car:
        def __init__(self):
            self.engine=Engine()
        def m2(self):
            print('Car using Engine Class Functionality')
            print(self.engine.a)
            print(self.engine.b)
            self.engine.m1()
    c=Car()
    c.m2()
```

**Output:**

```
Car using Engine Class Functionality
10
20
Engine Specific Functionality
```

**Demo Program-2:**

```
class Car:
    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color
    def getinfo(self):
        print("Car Name:{} , Model:{} and Color:{}".format(self.name,self.model,self.c
olor))

class Employee:
    def __init__(self,ename,eno,car):
        self.ename=ename
        self.eno=eno
        self.car=car
    def empinfo(self):
        print("Employee Name:",self.ename)
        print("Employee Number:",self.eno)
        print("Employee Car Info:")
        self.car.getinfo()
c=Car("Innova","2.5V","Grey")
e=Employee('Durga',10000,c)
e.empinfo()
```

**Output:**
```
Employee Name: Durga
Employee Number: 10000
Employee Car Info:
Car Name: Innova, Model: 2.5V and Color:Grey
```

In the above program Employee class Has-A Car reference and hence Employee class can access all members of Car class.

**Demo Program-3:**

```python
class X:
  a=10
  def __init__(self):
    self.b=20
  def m1(self):
      print("m1 method of X class")

class Y:
  c=30
  def __init__(self):
    self.d=40
  def m2(self):
    print("m2 method of Y class")

  def m3(self):
    x1=X()
    print(x1.a)
    print(x1.b)
    x1.m1()
    print(Y.c)
    print(self.d)
    self.m2()
    print("m3 method of Y class")
y1=Y()
     y1.m3()
```

**Output:**
```
10
20
m1 method of X class
30
40
m2 method of Y class
m3 method of Y class
```

## 2) By Inheritance (IS-A Relationship):

What ever variables, methods and constructors available in the parent class by default available to the child classes and we are not required to rewrite. Hence the main advantage of inheritance is Code Reusability and we can extend existing functionality with some more extra functionality.

**Syntax:** class childclass(parentclass)

```
class P:
    a=10
    def __init__(self):
        self.b=10
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')

class C(P):
    pass

c=C()
print(c.a)
print(c.b)
c.m1()
c.m2()
c.m3()
```

**Output:**

```
10
10
Parent instance method
Parent class method
Parent static method
```

```
class P:
     10 methods
class C(P):
    5 methods
```

In the above example Parent class contains 10 methods and these methods automatically available to the child class and we are not required to rewrite those methods(Code Reusability)
Hence child class contains 15 methods.

<u>Note:</u> What ever members present in Parent class are by default available to the child class through inheritance.

```
class P:
   def m1(self):
      print("Parent class method")
class C(P):
   def m2(self):
      print("Child class method")

c=C();
c.m1()
c.m2()
```

**<u>Output:</u>**

```
Parent class method
Child class method
```

What ever methods present in Parent class are automatically available to the child class and hence on the child class reference we can call both parent class methods and child class methods.

Similarly variables also

```
class P:
   a=10
   def __init__(self):
      self.b=20
class C(P):
   c=30
   def __init__(self):
      super().__init__()===>Line-1
      self.d=30

c1=C()
print(c1.a,c1.b,c1.c,c1.d)
```

If we comment Line-1 then variable b is not available to the child class.

## Demo program for Inheritance:

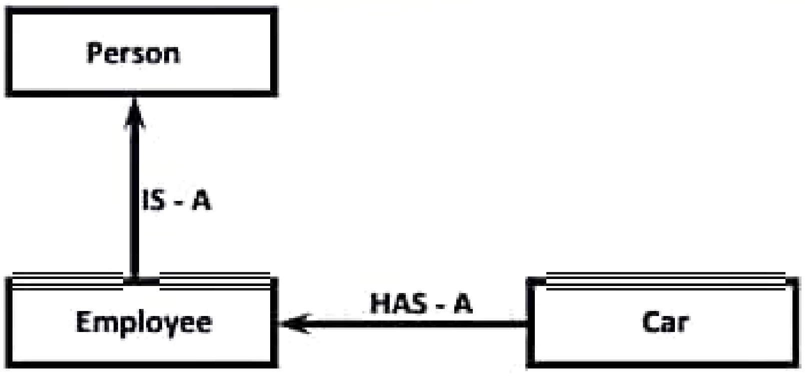
```
class Person:
  def __init__(self,name,age):
    self.name=name
    self.age=age
  def eatndrink(self):
    print('Eat Biryani and Drink Beer')

class Employee(Person):
  def __init__(self,name,age,eno,esal):
    super().__init__(name,age)
    self.eno=eno
    self.esal=esal

  def work(self):
    print("Coding Python is very easy just like drinking Chilled Beer")
  def empinfo(self):
    print("Employee Name:",self.name)
    print("Employee Age:",self.age)
    print("Employee Number:",self.eno)
    print("Employee Salary:",self.esal)

e=Employee('Durga', 48, 100, 10000)
e.eatndrink()
e.work()
e.empinfo()
```

**Output:**

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
```

## IS-A vs HAS-A Relationship:

- If we want to extend existing functionality with some more extra functionality then we should go for IS-A Relationship.
- If we dont want to extend and just we have to use existing functionality then we should go for HAS-A Relationship.
- Eg: Employee class extends Person class Functionality But Employee class just uses Car functionality but not extending



```
class Car:
    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color
    def getinfo(self):
        print("\tCar Name:{} \n\t Model:{} \n\t Color:{}".format(self.name,self.model,
self.color))

class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def  eatndrink(self):
        print('Eat Biryani and Drink Beer')

class Employee(Person):
    def __init__(self,name,age,eno,esal,car):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal
        self.car=car
    def work(self):
        print("Coding Python is very easy just like drinking Chilled Beer")
    def empinfo(self):
        print("Employee Name:",self.name)
        print("Employee Age:",self.age)
        print("Employee Number:",self.eno)
        print("Employee Salary:",self.esal)
        print("Employee Car Info:")
        self.car.getinfo()

c=Car("Innova","2.5V","Grey")
e=Employee('Durga',48,100,10000,c)
e.eatndrink()
```

```
35) e.work()
36) e.empinfo()
```

**Output:**

```
Eat Biryani and Drink Beer
Coding Python is very easy just like drinking Chilled Beer
Employee Name: Durga
Employee Age: 48
Employee Number: 100
Employee Salary: 10000
Employee Car Info:
      Car Name:Innova
       Model:2.5V
       Color:Grey
```

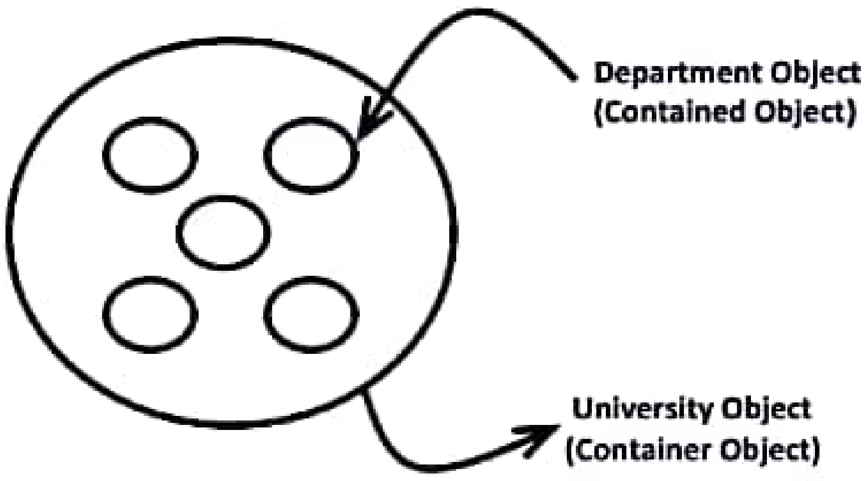
In the above example Employee class extends Person class functionality but just uses Car class functionality.

## Composition vs Aggregation:

### Composition:

Without existing container object if there is no chance of existing contained object then the container and contained objects are strongly associated and that strong association is nothing but Composition.

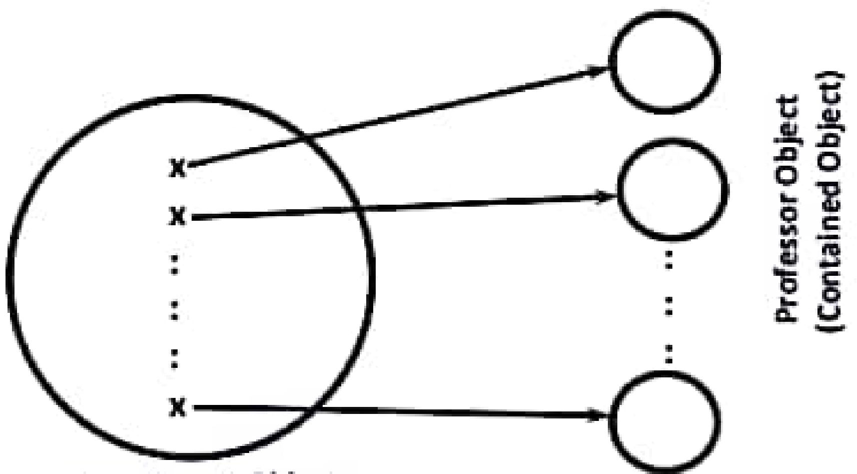
Eg: University contains several Departments and without existing university object there is no chance of existing Department object. Hence University and Department objects are strongly associated and this strong association is nothing but Composition.



## Aggregation:

Without existing container object if there is a chance of existing contained object then the container and contained objects are weakly associated and that weak association is nothing but Aggregation.

Eg: Department contains several Professors. Without existing Department still there may be a chance of existing Professor. Hence Department and Professor Objects are weakly associated, which is nothing but Aggregation.



Department Object
(Container Object)

### Coding Example:

```
class Student:
    collegeName='DURGASOFT'
    def __init__(self,name):
        self.name=name
print(Student.collegeName)
s=Student('Durga')
print(s.name)
```

### Output:

```
DURGASOFT
Durga
```

In the above example without existing Student object there is no chance of existing his name. Hence Student Object and his name are strongly associated which is nothing but Composition.

But without existing Student object there may be a chance of existing collegeName. Hence Student object and collegeName are weakly associated which is nothing but Aggregation.

## Conclusion:

The relation between object and its instance variables is always Composition where as the relation between object and static variables is Aggregation.

**Note:** Whenever we are creating child class object then child class constructor will be executed. If the child class does not contain constructor then parent class constructor will be executed, but parent object won't be created.

```python
class P:
    def __init__(self):
        print(id(self))
class C(P):
    pass
c=C()
print(id(c))
```

**Output:**

```
6207088
6207088
```

```python
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
class Student(Person):
    def __init__(self,name,age,rollno,marks):
        super().__init__(name,age)
        self.rollno=rollno
        self.marks=marks
    def __str__(self):
        return 'Name={}\nAge={}\nRollno={}\nMarks={}'.format(self.name,self.age,self.rollno,self.marks)
s1=Student('durga',48,101,90)
print(s1)
```

**Output:**

```
Name=durga
Age=48
Rollno=101
Marks=90
```

**Note:** In the above example when ever we are creating child class object both parent and child class constructors got executed to perform initialization of child object.

## Types of Inheritance:

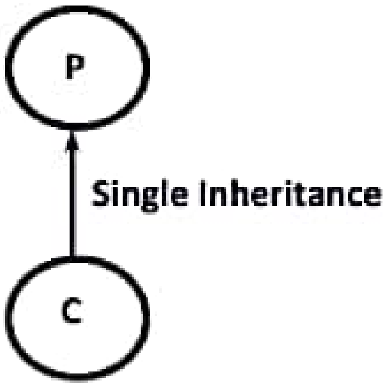
### 1) Single Inheritance:

The concept of inheriting the properties from one class to another class is known as single inheritance.

```
class P:
   def m1(self):
       print("Parent Method")
class C(P):
   def m2(self):
       print("Child Method")
c=C()
c.m1()
c.m2()
```

**Output:**

```
Parent Method
Child Method
```



### 2) Multi Level Inheritance:

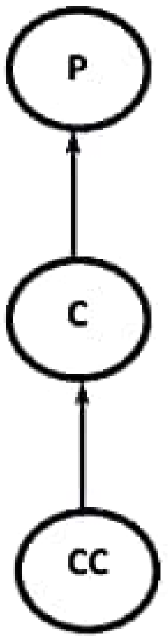
The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance.

```
class P:
   def m1(self):
       print("Parent Method")
class C(P):
   def m2(self):
       print("Child Method")
class CC(C):
   def m3(self):
       print("Sub Child Method")
c=CC()
```

```
11) c.m1()
12) c.m2()
13) c.m3()
```

**Output:**

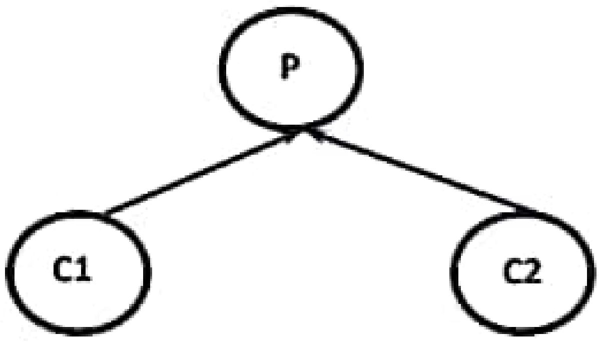
```
Parent Method
Child Method
Sub Child Method
```



Multi – Level Inheritance

## 3) Hierarchical Inheritance:

The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance



Hierarchical Inheritance

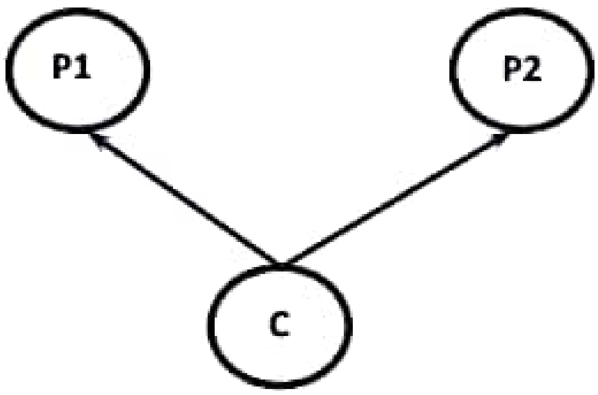
```
1) class P:
2)     def m1(self):
3)         print("Parent Method")
4) class C1(P):
    def m2(self):
        print("Child1 Method")
class C2(P):
    def m3(self):
        print("Child2 Method")
c1=C1()
c1.m1()
c1.m2()
c2=C2()
c2.m1()
c2.m3()
```

**Output:**

```
Parent Method
Child1 Method
Parent Method
Child2 Method
```

## 4) Multiple Inheritance:

The concept of inheriting the properties from multiple classes into a single class at a time, is known as multiple inheritance.



Multiple Inheritance

```
class P1:
    def m1(self):
        print("Parent1 Method")
class P2:
    def m2(self):
        print("Parent2 Method")
class C(P1,P2):
    def m3(self):
        print("Child2 Method")
```

```
c=C()
c.m1()
c.m2()
c.m3()
```

**Output:**

```
Parent1 Method
Parent2 Method
Child2 Method
```

If the same method is inherited from both parent classes, then Python will always consider the order of Parent classes in the declaration of the child class.

class C(P1, P2): → P1 method will be considered
class C(P2, P1): → P2 method will be considered

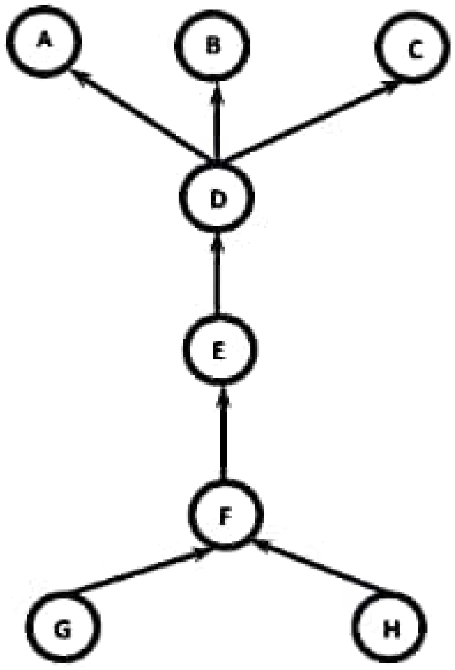
```
class P1:
    def m1(self):
        print("Parent1 Method")
class P2:
    def m1(self):
        print("Parent2 Method")
class C(P1,P2):
    def m2(self):
        print("Child Method")
c=C()
c.m1()
c.m2()
```

**Output:**

```
Parent1 Method
Child Method
```

## 5) Hybrid Inheritance:

Combination of Single, Multi level, multiple and Hierarchical Inheritance is known as Hybrid Inheritance.

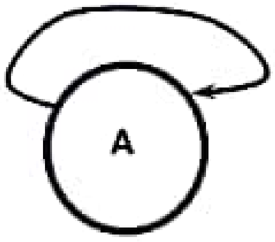


## 6) Cyclic Inheritance:

The concept of inheriting properties from one class to another class in cyclic way, is called Cyclic inheritance.Python won't support for Cyclic Inheritance of course it is really not required.

**Eg - 1:** class A(A):pass

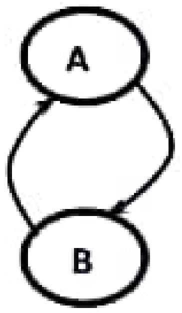
NameError: name 'A' is not defined



**Eg - 2:**

```
class A(B):
    pass
class B(A):
    pass
```

NameError: name 'B' is not defined



## Method Resolution Order (MRO):

- In Hybrid Inheritance the method resolution order is decided based on MRO algorithm.
- This algorithm is also known as C3 algorithm.
- Samuele Pedroni proposed this algorithm.
- It follows DLR (Depth First Left to Right) i.e Child will get more priority than Parent.
- Left Parent will get more priority than Right Parent.
- MRO(X) = X+Merge(MRO(P1),MRO(P2),...,ParentList)

## Head Element vs Tail Terminology:

- Assume C1,C2,C3,...are classes.
- In the list: C1C2C3C4C5....
- C1 is considered as Head Element and remaining is considered as Tail.

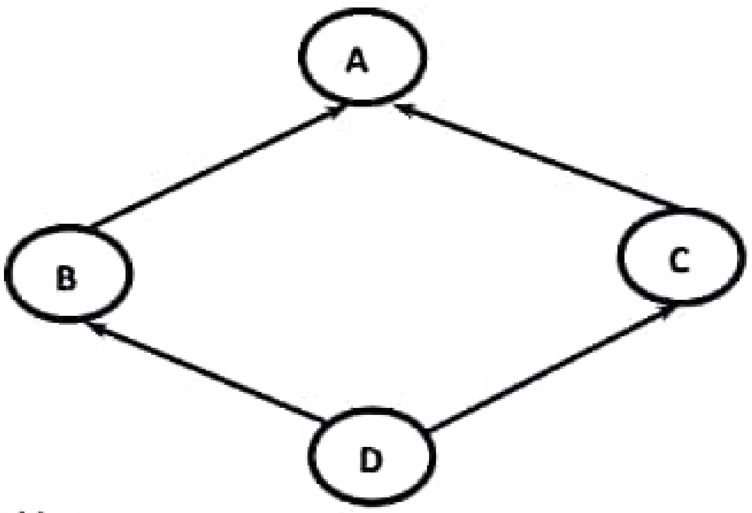
## How to find Merge:

- Take the head of first list
- If the head is not in the tail part of any other list, then add this head to the result and remove it from the lists in the merge.
- If the head is present in the tail part of any other list, then consider the head element of the next list and continue the same process.

**Note:** We can find MRO of any class by using mro() function.

```
print(ClassName.mro())
```

## Demo Program-1 for Method Resolution Order:


$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

mro(A) = A, object
mro(B) = B, A, object
mro(C) = C, A, object
mro(D) = D, B, C, A, object

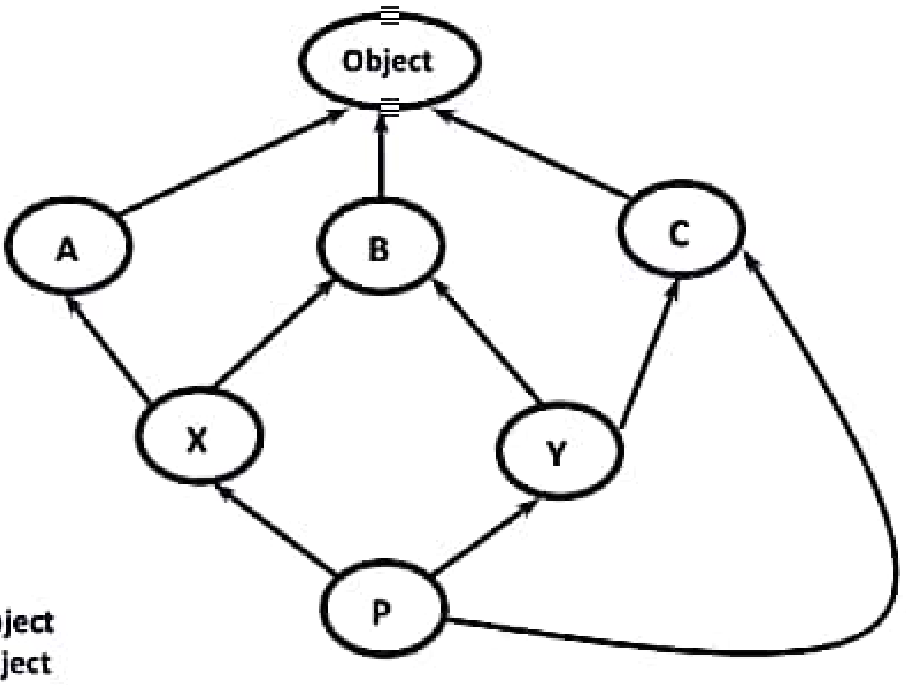
**test.py**

```
class A:pass
class B(A):pass
class C(A):pass
class D(B,C):pass
print(A.mro())
print(B.mro())
print(C.mro())
print(D.mro())
```

**Output:**

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

## Demo Program-2 for Method Resolution Order:



mro(A)=A,object
mro(B)=B,object
mro(C)=C,object
mro(X)=X,A,B,object
mro(Y)=Y,B,C,object
mro(P)=P,X,A,Y,B,C,object

## Finding mro(P) by using C3 Algorithm:

**Formula:** MRO(X) = X+Merge(MRO(P1),MRO(P2),...,ParentList)

```
mro(p) = P+Merge(mro(X),mro(Y),mro(C),XYC)
       = P+Merge(XABO,YBCO,CO,XYC)
       = P+X+Merge(ABO,YBCO,CO,YC)
       = P+X+A+Merge(BO,YBCO,CO,YC)
       = P+X+A+Y+Merge(BO,BCO,CO,C)
       = P+X+A+Y+B+Merge(O,CO,CO,C)
       = P+X+A+Y+B+C+Merge(O,O,O)
       = P+X+A+Y+B+C+O
```

**test.py**

```
class A:pass
class B:pass
class C:pass
class X(A,B):pass
class Y(B,C):pass
class P(X,Y,C):pass
```

```
print(A.mro())#AO
print(X.mro())#XABO
print(Y.mro())#YBCO
print(P.mro())#PXAYBCO
```

**Output:**

```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.X'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
[<class '__main__.Y'>, <class '__main__.B'>, <class '__main__.C'>, <class 'object'>]
[<class '__main__.P'>, <class '__main__.X'>, <class '__main__.A'>, <class '__main__.Y'>,
<class '__main__.B'>,
 <class '__main__.C'>, <class 'object'>]
```

**test.py**

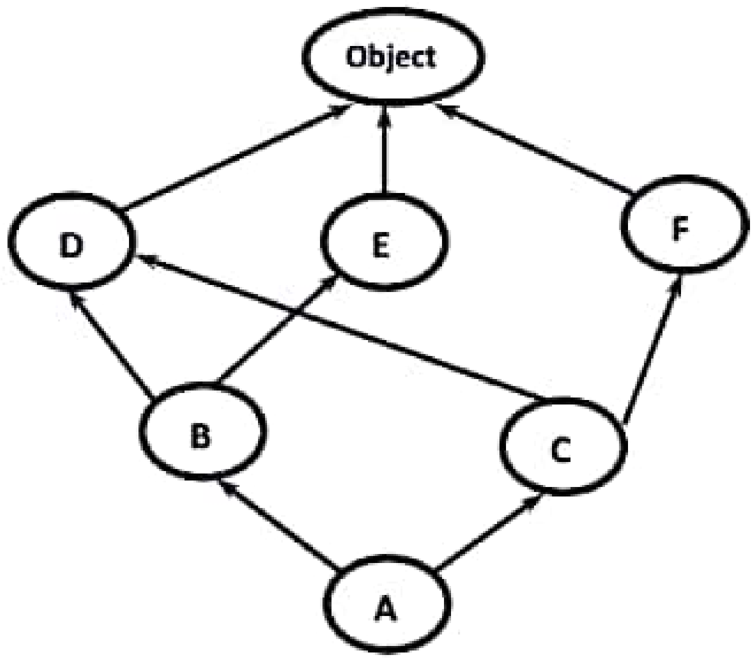
```
class A:
    def m1(self):
        print('A class Method')
class B:
    def m1(self):
        print('B class Method')
class C:
    def m1(self):
        print('C class Method')
class X(A,B):
    def m1(self):
        print('X class Method')
class Y(B,C):
    def m1(self):
        print('Y class Method')
class P(X,Y,C):
    def m1(self):
        print('P class Method')
p=P()
p.m1()
```

**Output:** P class Method

In the above example P class m1() method will be considered.If P class does not contain m1() method then as per MRO, X class method will be considered. If X class does not contain then A class method will be considered and this process will be continued.

The method resolution in the following order: PXAYBCO

## Demo Program-3 for Method Resolution Order:



```
mro(o) = object
mro(D) = D,object
mro(E) = E,object
mro(F) = F,object
mro(B) = B,D,E,object
mro(C) = C,D,F,object
mro(A) = A+Merge(mro(B),mro(C),BC)
         = A+Merge(BDEO,CDFO,BC)
         = A+B+Merge(DEO,CDFO,C)
         = A+B+C+Merge(DEO,DFO)
         = A+B+C+D+Merge(EO,FO)
         = A+B+C+D+E+Merge(O,FO)
         = A+B+C+D+E+F+Merge(O,O)
         = A+B+C+D+E+F+O
```

**test.py**

```python
class D:pass
class E:pass
class F:pass
class B(D,E):pass
class C(D,F):pass
class A(B,C):pass
print(D.mro())
print(B.mro())
print(C.mro())
print(A.mro())
```

**Output:**

```
[<class '__main__.D'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.D'>, <class '__main__.E'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.D'>, <class '__main__.F'>, <class 'object'>]
[<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.D'>,
<class '__main__.E'>,
 <class '__main__.F'>, <class 'object'>]
```

## super() Method:

super() is a built-in method which is useful to call the super class constructors,variables and methods from the child class.

## Demo Program-1 for super():

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def display(self):
        print('Name:',self.name)
        print('Age:',self.age)

class Student(Person):
    def __init__(self,name,age,rollno,marks):
        super().__init__(name,age)
        self.rollno=rollno
        self.marks=marks

    def display(self):
        super().display()
        print('Roll No:',self.rollno)
        print('Marks:',self.marks)

s1=Student('Durga',22,101,90)
s1.display()
```

**Output:**

```
Name: Durga
Age: 22
Roll No: 101
Marks: 90
```

In the above program we are using super() method to call parent class constructor and display() method

## Demo Program-2 for super():

```
class P:
  a=10
  def __init__(self):
    self.b=10
  def m1(self):
    print('Parent instance method')
  @classmethod
  def m2(cls):
    print('Parent class method')
  @staticmethod
  def m3():
    print('Parent static method')

class C(P):
  a=888
  def __init__(self):
    self.b=999
    super().__init__()
    print(super().a)
    super().m1()
    super().m2()
    super().m3()

c=C()
```

**Output:**

```
10
Parent instance method
Parent class method
Parent static method
```

In the above example we are using super() to call various members of Parent class.

## How to Call Method of a Particular Super Class:

We can use the following approaches

1) super(D, self).m1()
   It will call m1() method of super class of D.

2) A.m1(self)
   It will call A class m1() method

```
class A:
    def m1(self):
        print('A class Method')
class B(A):
    def m1(self):
        print('B class Method')
class C(B):
    def m1(self):
        print('C class Method')
class D(C):
    def m1(self):
        print('D class Method')
class E(D):
    def m1(self):
        A.m1(self)

e=E()
e.m1()
```

**Output:** A class Method

## Various Important Points about super():

**Case-1:** From child class we are not allowed to access parent class instance variables by using super(), Compulsory we should use self only.
But we can access parent class static variables by using super().

```
class P:
    a=10
    def __init__(self):
        self.b=20

class C(P):
    def m1(self):
        print(super().a)#valid
        print(self.b)#valid
        print(super().b)#invalid
c=C()
c.m1()
```

**Output:**
```
10
20
AttributeError: 'super' object has no attribute 'b'
```

**Case-2:** From child class constructor and instance method, we can access parent class instance method, static method and class method by using super()

```
class P:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')

class C(P):
    def __init__(self):
        super().__init__()
        super().m1()
        super().m2()
        super().m3()

    def m1(self):
        super().__init__()
        super().m1()
        super().m2()
        super().m3()

c=C()
c.m1()
```

**Output:**

```
Parent Constructor
Parent instance method
Parent class method
Parent static method
Parent Constructor
Parent instance method
Parent class method
Parent static method
```

**Case-3:** From child class, class method we cannot access parent class instance methods and constructors by using super() directly(but indirectly possible). But we can access parent class static and class methods.

```
class P:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')

class C(P):
    @classmethod
    def m1(cls):
        #super().__init__()--->invalid
        #super().m1()--->invalid
        super().m2()
        super().m3()

C.m1()
```

**Output:**

```
Parent class method
Parent static method
```

**From Class Method of Child Class, how to call Parent Class Instance Methods and Constructors:**

```
class A:
    def __init__(self):
        print('Parent constructor')

    def m1(self):
        print('Parent instance method')

class B(A):
    @classmethod
    def m2(cls):
        super(B,cls).__init__(cls)
        super(B,cls).m1(cls)

B.m2()
```

**Output:**

```
Parent constructor
Parent instance method
```

**Case-4:** In child class static method we are not allowed to use super() generally (But in special way we can use)

```
class P:
    def __init__(self):
        print('Parent Constructor')
    def m1(self):
        print('Parent instance method')
    @classmethod
    def m2(cls):
        print('Parent class method')
    @staticmethod
    def m3():
        print('Parent static method')

class C(P):
    @staticmethod
    def m1():
        super().m1()-->invalid
        super().m2()--->invalid
        super().m3()--->invalid

C.m1()
```

RuntimeError: super(): no arguments

## How to Call Parent Class Static Method from Child Class Static Method by using super():

```
class A:

    @staticmethod
    def m1():
        print('Parent static method')

class B(A):
    @staticmethod
    def m2():
        super(B,B).m1()

B.m2()
```

**Output:** Parent static method

# OOP's Part - 3

# POLYMORPHISM

poly means many. Morphs means forms.
Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes.(overriding)

Related to Polymorphism the following 4 topics are important

1) Duck Typing Philosophy of Python

2) Overloading
    1) Operator Overloading
    2) Method Overloading
    3) Constructor Overloading

3) Overriding
    1) Method Overriding
    2) Constructor Overriding

## 1) Duck Typing Philosophy of Python:

In Python we cannot specify the type explicitly. Based on provided value at runtime the type will be considered automatically. Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):
obj.talk()
```

**What is the Type of obj? We cannot decide at the Beginning. At Runtime we can Pass any Type. Then how we can decide the Type?**

At runtime if 'it walks like a duck and talks like a duck,it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Dog:
    def talk(self):
        print('Bow Bow..')

class Cat:
    def talk(self):
        print('Moew Moew ..')

class Goat:
    def talk(self):
        print('Myaah Myaah ..')

def  f1(obj):
    obj.talk()

l=[Duck(),Cat(),Dog(),Goat()]
for obj in l:
    f1(obj)
```

**Output:**

```
Quack.. Quack..
Moew Moew ..
Bow Bow..
Myaah Myaah ..
```

The problem in this approach is if obj does not contain talk() method then we will get AttributeError.

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Dog:
    def bark(self):
        print('Bow Bow..')
def  f1(obj):
    obj.talk()

d=Duck()
f1(d)

```

```
d=Dog()
f1(d)
```

**Output:**

```
D:\durga_classes>py test.py
Quack.. Quack..
Traceback (most recent call last):
  File "test.py", line 22, in <module>
    f1(d)
  File "test.py", line 13, in f1
    obj.talk()
AttributeError: 'Dog' object has no attribute 'talk'
```

But we can solve this problem by using hasattr() function.

`hasattr(obj,'attributename')` ➔ attributename can be Method Name OR Variable Name

## Demo Program with hasattr() Function:

```
class Duck:
    def talk(self):
        print('Quack.. Quack..')

class Human:
    def talk(self):
        print('Hello Hi...')

class Dog:
    def bark(self):
        print('Bow Bow..')

def  f1(obj):
    if hasattr(obj,'talk'):
        obj.talk()
    elif hasattr(obj,'bark'):
        obj.bark()

d=Duck()
f1(d)

h=Human()
f1(h)

d=Dog()
f1(d)
```

```
27) Myaah Myaah Myaah...
```

## 2) Overloading

We can use same operator or methods for different purposes.

Eg 1: + operator can be used for Arithmetic addition and String concatenation
print(10+20)#30
print('durga'+'soft')#durgasoft

Eg 2: * operator can be used for multiplication and string repetition purposes.
print(10*20)#200
print('durga'*3)#durgadurgadurga

Eg 3: We can use deposit() method to deposit cash or cheque or dd
deposit(cash)
deposit(cheque)
deposit(dd)

There are 3 types of Overloading
1) Operator Overloading
2) Method Overloading
3) Constructor Overloading

## 1) Operator Overloading:

- We can use the same operator for multiple purposes, which is nothing but operator overloading.
- Python supports operator overloading.

Eg 1: + operator can be used for Arithmetic addition and String concatenation
print(10+20)#30
print('durga'+'soft')#durgasoft

Eg 2: * operator can be used for multiplication and string repetition purposes.
print(10*20)#200
print('durga'*3)#durgadurgadurga

## Demo program to use + operator for our class objects:

```
class Book:
  def __init__(self,pages):
    self.pages=pages

b1=Book(100)
```

```
b2=Book(200)
print(b1+b2)
```

```
D:\durga_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

- We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.
- For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.
- Internally + operator is implemented by using __add__() method.This method is called magic method for + operator. We have to override this method in our class.

## Demo Program to Overload + Operator for Our Book Class Objects:

```python
class Book:
    def __init__(self,pages):
        self.pages=pages

    def __add__(self,other):
        return self.pages+other.pages

b1=Book(100)
b2=Book(200)
print('The Total Number of Pages:',b1+b2)
```

**Output:** The Total Number of Pages: 300

The following is the list of operators and corresponding magic methods.

| | | |
|---|---|---|
| 1) + | → | object.__add__(self,other) |
| 2) - | → | object.__sub__(self,other) |
| 3) * | → | object.__mul__(self,other) |
| 4) / | → | object.__div__(self,other) |
| 5) // | → | object.__floordiv__(self,other) |
| 6) % | → | object.__mod__(self,other) |
| 7) ** | → | object.__pow__(self,other) |
| 8) += | → | object.__iadd__(self,other) |
| 9) -= | → | object.__isub__(self,other) |
| 10) *= | → | object.__imul__(self,other) |
| 11) /= | → | object.__idiv__(self,other) |
| 12) //= | → | object.__ifloordiv__(self,other) |

| | | |
|---|---|---|
| 13) %= | → | object.__imod__(self,other) |
| 14) **= | → | object.__ipow__(self,other) |
| 15) < | → | object.__lt__(self,other) |
| 16) <= | → | object.__le__(self,other) |
| 17) > | → | object.__gt__(self,other) |
| 18) >= | → | object.__ge__(self,other) |
| 19) == | → | object.__eq__(self,other) |
| 20) != | → | object.__ne__(self,other) |

## Overloading > and <= Operators for Student Class Objects:

```
class Student:
  def __init__(self,name,marks):
    self.name=name
    self.marks=marks
  def __gt__(self,other):
    return self.marks>other.marks
  def __le__(self,other):
    return self.marks<=other.marks

print("10>20 =",10>20)
s1=Student("Durga",100)
s2=Student("Ravi",200)
print("s1>s2=",s1>s2)
print("s1<s2=",s1<s2)
print("s1<=s2=",s1<=s2)
print("s1>=s2=",s1>=s2)
```

**Output**

```
10>20 = False
s1>s2= False
s1<s2= True
s1<=s2= True
s1>=s2= False
```

## Program to Overload Multiplication Operator to Work on Employee Objects:

```
class Employee:
  def __init__(self,name,salary):
    self.name=name
    self.salary=salary
  def __mul__(self,other):
    return self.salary*other.days

```

```
class TimeSheet:
    def __init__(self,name,days):
        self.name=name
        self.days=days

e=Employee('Durga',500)
t=TimeSheet('Durga',25)
print('This Month Salary:',e*t)
```

**Output:** This Month Salary: 12500

## 2) Method Overloading:

- If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.
  Eg: m1(int a)
  m1(double d)

- But in Python Method overloading is not possible.
- If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

## Demo Program:

```
class Test:
    def m1(self):
        print('no-arg method')
    def m1(self,a):
        print('one-arg method')
    def m1(self,a,b):
        print('two-arg method')

t=Test()
#t.m1()
#t.m1(10)
t.m1(10,20)
```

**Output:** two-arg method

In the above program python will consider only last method.

## How we can handle Overloaded Method Requirements in Python:

Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

## Demo Program with Default Arguments:

```
class Test:
  def sum(self,a=None,b=None,c=None):
    if a!=None and b!= None and c!= None:
      print('The Sum of 3 Numbers:',a+b+c)
    elif a!=None and b!= None:
      print('The Sum of 2 Numbers:',a+b)
    else:
      print('Please provide 2 or 3 arguments')
t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
```

**Output**

```
The Sum of 2 Numbers: 30
The Sum of 3 Numbers: 60
Please provide 2 or 3 arguments
```

## Demo Program with Variable Number of Arguments:

```
class Test:
  def sum(self,*a):
    total=0
    for x in a:
      total=total+x
    print('The Sum:',total)

t=Test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
t.sum()
```

## 3) Constructor Overloading:

- Constructor overloading is not possible in Python.
- If we define multiple constructors then the last constructor will be considered.

```
class Test:
  def __init__(self):
    print('No-Arg Constructor')

    def __init__(self,a):
        print('One-Arg constructor')

    def __init__(self,a,b):
        print('Two-Arg constructor')
#t1=Test()
#t1=Test(10)
t1=Test(10,20)
```

**Output:** Two-Arg constructor

- In the above program only Two-Arg Constructor is available.
- But based on our requirement we can declare constructor with default arguments and variable number of arguments.

## Constructor with Default Arguments:

```
class Test:
    def __init__(self,a=None,b=None,c=None):
        print('Constructor with 0|1|2|3 number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
```

**Output**

```
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
```

## Constructor with Variable Number of Arguments:

```
class Test:
    def __init__(self,*a):
        print('Constructor with variable number of arguments')

t1=Test()
t2=Test(10)
t3=Test(10,20)
t4=Test(10,20,30)
t5=Test(10,20,30,40,50,60)
```

**Output:**

```
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
Constructor with variable number of arguments
```

## 3) Overriding

### Method Overriding

- What ever members available in the parent class are bydefault available to the child class through inheritance. If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.
- Overriding concept applicable for both methods and constructors.

### Demo Program for Method Overriding:

```python
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Appalamma')
class C(P):
    def marry(self):
        print('Katrina Kaif')

c=C()
c.property()
c.marry()
```

**Output**

```
Gold+Land+Cash+Power
Katrina Kaif
```

From Overriding method of child class,we can call parent class method also by using super() method.

```python
class P:
    def property(self):
        print('Gold+Land+Cash+Power')
    def marry(self):
        print('Appalamma')
class C(P):
    def marry(self):
        super().marry()
        print('Katrina Kaif')

c=C()
c.property()
c.marry()
```

**Output**

```
Gold+Land+Cash+Power
Appalamma
Katrina Kaif
```

## Demo Program for Constructor Overriding:

```
class P:
    def __init__(self):
        print('Parent Constructor')

class C(P):
    def __init__(self):
        print('Child Constructor')

c=C()
```

**Output:** Child Constructor

In the above example,if child class does not contain constructor then parent class constructor will be executed

From child class constuctor we can call parent class constructor by using super() method.

## Demo Program to call Parent Class Constructor by using super():

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

class Employee(Person):
    def __init__(self,name,age,eno,esal):
        super().__init__(name,age)
        self.eno=eno
        self.esal=esal

    def display(self):
        print('Employee Name:',self.name)
        print('Employee Age:',self.age)
        print('Employee Number:',self.eno)
        print('Employee Salary:',self.esal)

e1=Employee('Durga',48,872425,26000)
e1.display()
e2=Employee('Sunny',39,872426,36000)
e2.display()
```

**Output**

```
Employee Name: Durga
Employee Age: 48
Employee Number: 872425
Employee Salary: 26000

Employee Name: Sunny
Employee Age: 39
Employee Number: 872426
Employee Salary: 36000
```

# OOP's Part - 4

## Agenda

1) Abstract Method
2) Abstract class
3) Interface
4) Public,Private and Protected Members
5) __str__() Method
6) Difference between str() and repr() functions
7) Small Banking Application

## Abstract Method:

- Sometimes we don't know about implementation, still we can declare a method. Such types of methods are called abstract methods.i.e abstract method has only declaration but not implementation.
- In python we can declare abstract method by using @abstractmethod decorator as follows.

- @abstractmethod
- def m1(self): pass

- @abstractmethod decorator present in abc module. Hence compulsory we should import abc module,otherwise we will get error.
- abc → abstract base class module

```
class Test:
    @abstractmethod
    def m1(self):
        pass
```

NameError: name 'abstractmethod' is not defined

**Eg:**

```
from abc import *
class Test:
    @abstractmethod
    def m1(self):
        pass
```

**Eg:**

```
from abc import *
class Fruit:
  @abstractmethod
  def taste(self):
     pass
```

Child classes are responsible to provide implemention for parent class abstract methods.

## Abstract class:

Some times implementation of a class is not complete,such type of partially implementation classes are called abstract classes. Every abstract class in Python should be derived from ABC class which is present in abc module.

### Case-1:

```
from abc import *
class Test:
  pass

t=Test()
```

In the above code we can create object for Test class b'z it is concrete class and it does not conatin any abstract method.

### Case-2:

```
from abc import *
class Test(ABC):
  pass

t=Test()
```

In the above code we can create object, even it is derived from ABC class,b'z it does not contain any abstract method.

### Case-3:

```
from abc import *
class Test(ABC):
  @abstractmethod
  def m1(self):
     pass
```

```

t=Test()
```

TypeError: Can't instantiate abstract class Test with abstract methods m1

**Case-4:**

```
from abc import *
class Test:
    @abstractmethod
    def m1(self):
        pass

t=Test()
```

We can create object even class contains abstract method b'z we are not extending ABC class.

**Case-5:**

```
from abc import *
class Test:
    @abstractmethod
    def m1(self):
        print('Hello')

t=Test()
t.m1()
```

**Output:** Hello

**Conclusion:** If a class contains atleast one abstract method and if we are extending ABC class then instantiation is not possible.

"abstract class with abstract method instantiation is not possible"

Parent class abstract methods should be implemented in the child classes. Otherwise we cannot instantiate child class.If we are not creating child class object then we won't get any error.

**Case-1:**

```
from abc import *
class Vehicle(ABC):
    @abstractmethod
```

```
    def noofwheels(self):
        pass

class Bus(Vehicle): pass
```

It is valid because we are not creating Child class object.

Case-2:

```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle): pass
b=Bus()
```

TypeError: Can't instantiate abstract class Bus with abstract methods noofwheels

Note: If we are extending abstract class and does not override its abstract method then child class is also abstract and instantiation is not possible.

```
from abc import *
class Vehicle(ABC):
    @abstractmethod
    def noofwheels(self):
        pass

class Bus(Vehicle):
    def noofwheels(self):
        return 7

class Auto(Vehicle):
    def noofwheels(self):
        return 3
b=Bus()
print(b.noofwheels())#7

a=Auto()
print(a.noofwheels())#3
```

Note: Abstract class can contain both abstract and non-abstract methods also.

## Interfaces In Python:

In general if an abstract class contains only abstract methods such type of abstract class is considered as interface.

```
from abc import *
class DBInterface(ABC):
    @abstractmethod
    def connect(self):pass

    @abstractmethod
    def disconnect(self):pass

class Oracle(DBInterface):
    def connect(self):
        print('Connecting to Oracle Database...')
    def disconnect(self):
        print('Disconnecting to Oracle Database...')

class Sybase(DBInterface):
    def connect(self):
        print('Connecting to Sybase Database...')
    def disconnect(self):
        print('Disconnecting to Sybase Database...')

dbname=input('Enter Database Name:')
classname=globals()[dbname]
x=classname()
x.connect()
x.disconnect()
```

```
D:\durga_classes>py test.py
Enter Database Name:Oracle
Connecting to Oracle Database...
Disconnecting to Oracle Database...

D:\durga_classes>py test.py
Enter Database Name:Sybase
Connecting to Sybase Database...
Disconnecting to Sybase Database...
```

**Note:** The inbuilt function globals()[str] converts the string 'str' into a class name and returns the classname.

**Demo Program-2:** Reading class name from the file

**config.txt**

EPSON

**test.py**

```
from abc import *
class Printer(ABC):
    @abstractmethod
    def printit(self,text):pass

    @abstractmethod
    def disconnect(self):pass

class EPSON(Printer):
    def printit(self,text):
        print('Printing from EPSON Printer...')
        print(text)
    def disconnect(self):
        print('Printing completed on EPSON Printer...')

class HP(Printer):
    def printit(self,text):
        print('Printing from HP Printer...')
        print(text)
    def disconnect(self):
        print('Printing completed on HP Printer...')

with open('config.txt','r') as f:
    pname=f.readline()

classname=globals()[pname]
x=classname()
x.printit('This data has to print...')
x.disconnect()
```

**Output:**

```
Printing from EPSON Printer...
This data has to print...
Printing completed on EPSON Printer...
```

## Concreate class vs Abstract Class vs Inteface:

1) If we dont know anything about implementation just we have requirement specification then we should go for interface.
2) If we are talking about implementation but not completely then we should go for abstract class. (partially implemented class).
3) If we are talking about implementation completely and ready to provide service then we should go for concrete class.

```
from abc import *
class CollegeAutomation(ABC):
    @abstractmethod
    def m1(self): pass
    @abstractmethod
    def m2(self): pass
    @abstractmethod
    def m3(self): pass
class AbsCls(CollegeAutomation):
    def m1(self):
        print('m1 method implementation')
    def m2(self):
        print('m2 method implementation')

class ConcreteCls(AbsCls):
    def m3(self):
        print('m3 method implemnentation')

c=ConcreteCls()
c.m1()
c.m2()
c.m3()
```

## Public, Protected and Private Attributes:

By default every attribute is public. We can access from anywhere either within the class or from outside of the class.

Eg: name = 'durga'

Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefexing with _ symbol.

Syntax: _variablename = value

Eg: _name='durga'

But is is just convention and in reality does not exists protected attributes.

private attributes can be accessed only within the class.i.e from outside of the class we cannot access. We can declare a variable as private explicitly by prefexing with 2 underscore symbols.

**syntax:** __variablename=value

**Eg:** __name='durga'

```
class Test:
    x=10
    _y=20
    __z=30
    def m1(self):
        print(Test.x)
        print(Test._y)
        print(Test.__z)

t=Test()
t.m1()
print(Test.x)
print(Test._y)
print(Test.__z)
```

**Output:**

```
10
20
30
10
20
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    print(Test.__z)
AttributeError: type object 'Test' has no attribute '__z'
```

## How to Access Private Variables from Outside of the Class:

We cannot access private variables directly from outside of the class.
But we can access indirectly as follows    objectreference._classname__variablename

```
class Test:
    def __init__(self):
        self.__x=10

```

```
t=Test()
print(t._Test__x)#10
```

## __str__() method:

- Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format
  <__main__.classname object at 0x022144B0>

- To return meaningful string representation we have to override __str__() method.

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

    def __str__(self):
      return 'This is Student with Name:{} and Rollno:{}'.format(self.name,self.rollno)

s1=Student('Durga',101)
s2=Student('Ravi',102)
print(s1)
print(s2)
```

### Output without Overriding str():

```
<__main__.Student object at 0x022144B0>
<__main__.Student object at 0x022144D0>
```

### Output with Overriding str():

```
This is Student with Name: Durga and Rollno: 101
This is Student with Name: Ravi and Rollno: 102
```

### Difference between str() and repr()
### OR
### Difference between __str__() and __repr__()

- str() internally calls __str__() function and hence functionality of both is same.
- Similarly,repr() internally calls __repr__() function and hence functionality of both is same.
- str() returns a string containing a nicely printable representation object.
- The main purpose of str() is for readability.It may not possible to convert result string to original object.

```python
import datetime
today=datetime.datetime.now()
s=str(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
```

```
D:\durgaclasses>py test.py
2018-05-18 22:48:19.890888
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    d=eval(s)#converting str object to datetime
  File "<string>", line 1
    2018-05-18 22:48:19.890888
          ^
SyntaxError: invalid token
```

But repr() returns a string containing a printable representation of object.
The main goal of repr() is unambigouous. We can convert result string to original object by using eval() function,which may not possible in str() function.

```python
import datetime
today=datetime.datetime.now()
s=repr(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
print(d)
```

**Output:**

```
datetime.datetime(2018, 5, 18, 22, 51, 10, 875838)
2018-05-18 22:51:10.875838
```

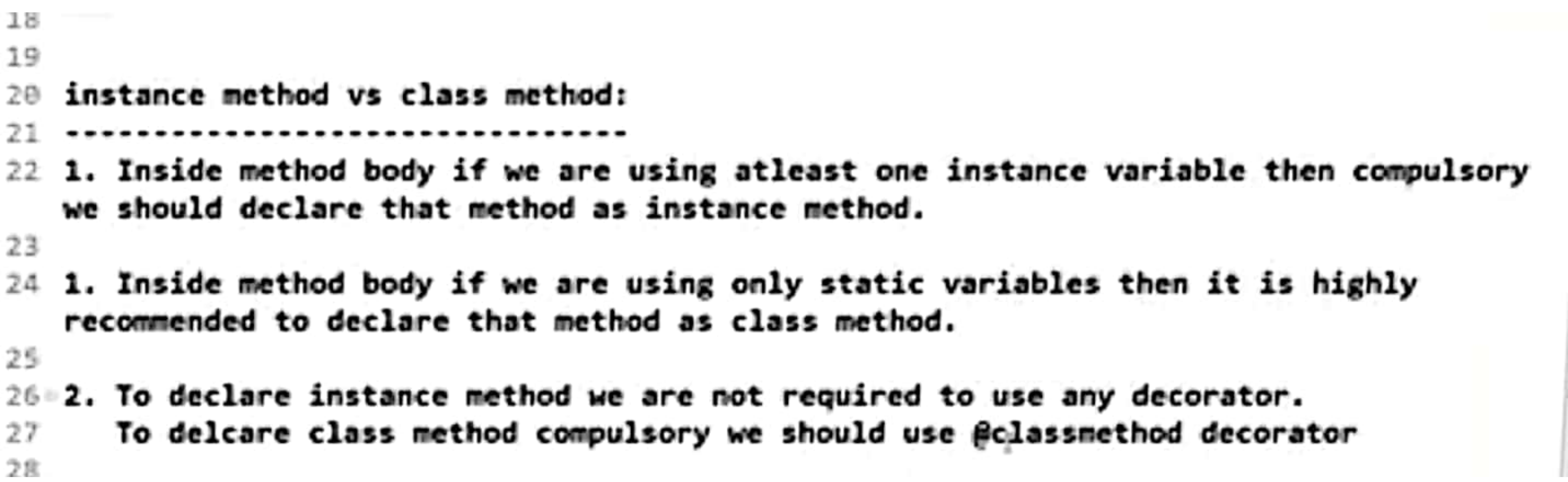
**Note:** It is recommended to use repr() instead of str()

## Mini Project: Banking Application

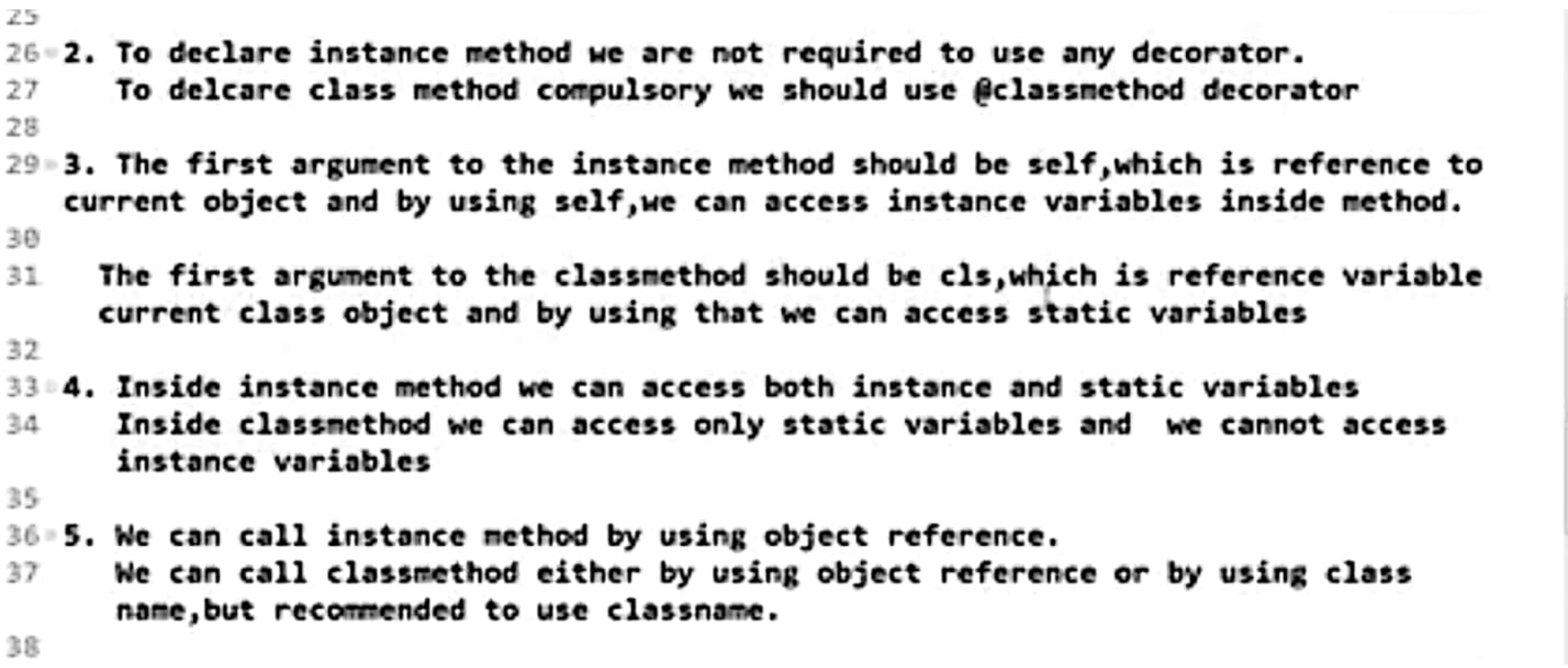
```python
class Account:
    def __init__(self,name,balance,min_balance):
        self.name=name
        self.balance=balance
        self.min_balance=min_balance

    def deposit(self,amount):
        self.balance +=amount

    def withdraw(self,amount):
        if self.balance-amount >= self.min_balance:
            self.balance -=amount
        else:
            print("Sorry, Insufficient Funds")

    def printStatement(self):
        print("Account Balance:",self.balance)

class Current(Account):
    def __init__(self,name,balance):
        super().__init__(name,balance,min_balance=-1000)
    def __str__(self):
        return "{}'s Current Account with Balance :{}".format(self.name,self.balance)

class Savings(Account):
    def __init__(self,name,balance):
        super().__init__(name,balance,min_balance=0)
    def __str__(self):
        return "{}'s Savings Account with Balance :{}".format(self.name,self.balance)

c=Savings("Durga",10000)
print(c)
c.deposit(5000)
c.printStatement()
c.withdraw(16000)
c.withdraw(15000)
print(c)

c2=Current('Ravi',20000)
c2.deposit(6000)
print(c2)
c2.withdraw(27000)
print(c2)
```

**Output:**

```
D:\durgaclasses>py test.py
Durga's Savings Account with Balance :10000
Account Balance: 15000
Sorry, Insufficient Funds
Durga's Savings Account with Balance :0
Ravi's Current Account with Balance :26000
Ravi's Current Account with Balance :-1000
```



```
instance method vs class method:
--------------------------------
1. Inside method body if we are using atleast one instance variable then compulsory
we should declare that method as instance method.

1. Inside method body if we are using only static variables then it is highly
recommended to declare that method as class method.

2. To declare instance method we are not required to use any decorator.
   To delcare class method compulsory we should use @classmethod decorator
```



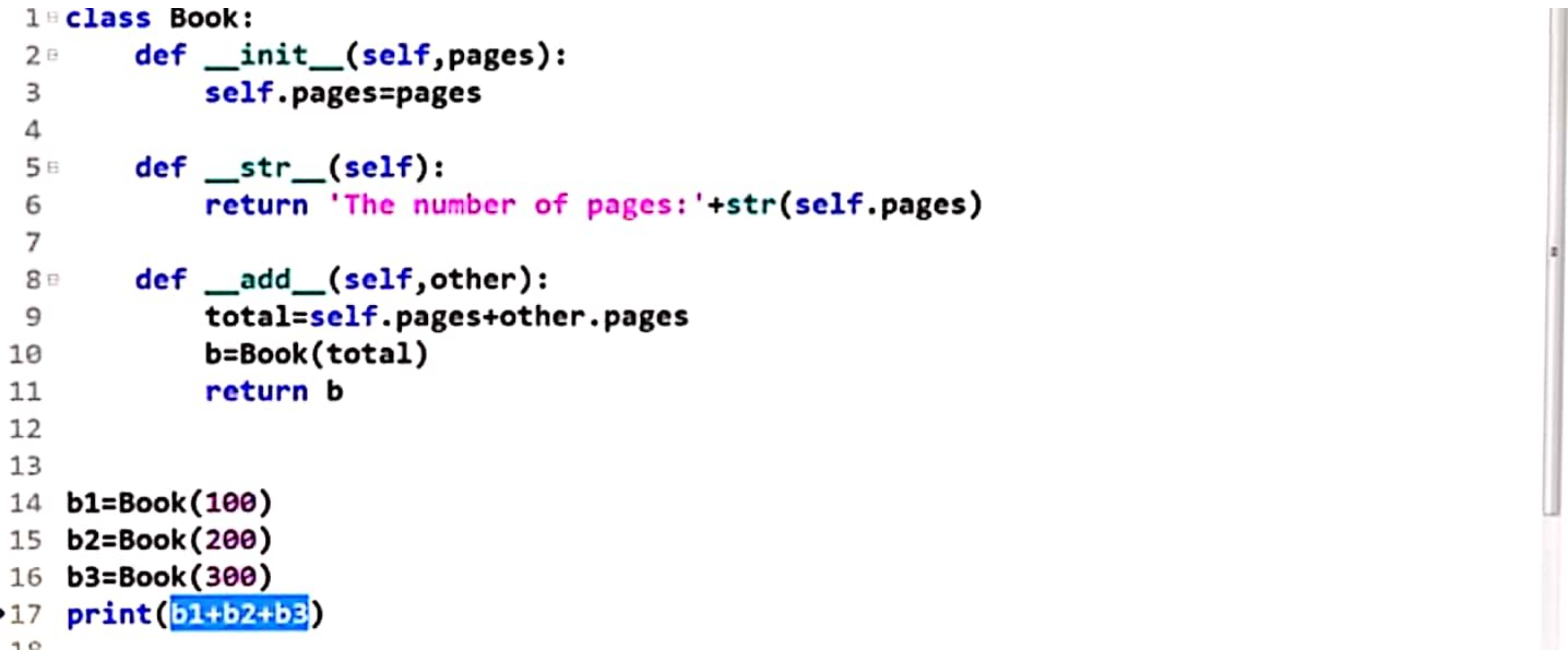
2. To declare instance method we are not required to use any decorator.
   To delcare class method compulsory we should use @classmethod decorator

3. The first argument to the instance method should be self,which is reference to current object and by using self,we can access instance variables inside method.

   The first argument to the classmethod should be cls,which is reference variable current class object and by using that we can access static variables

4. Inside instance method we can access both instance and static variables
   Inside classmethod we can access only static variables and we cannot access instance variables

5. We can call instance method by using object reference.
   We can call classmethod either by using object reference or by using class name,but recommended to use classname.



```
class Book:
    def __init__(self,pages):
        self.pages=pages

    def __str__(self):
        return 'The number of pages:'+str(self.pages)

    def __add__(self,other):
        total=self.pages+other.pages
        b=Book(total)
        return b


b1=Book(100)
b2=Book(200)
b3=Book(300)
print(b1+b2+b3)
```