Python Programming

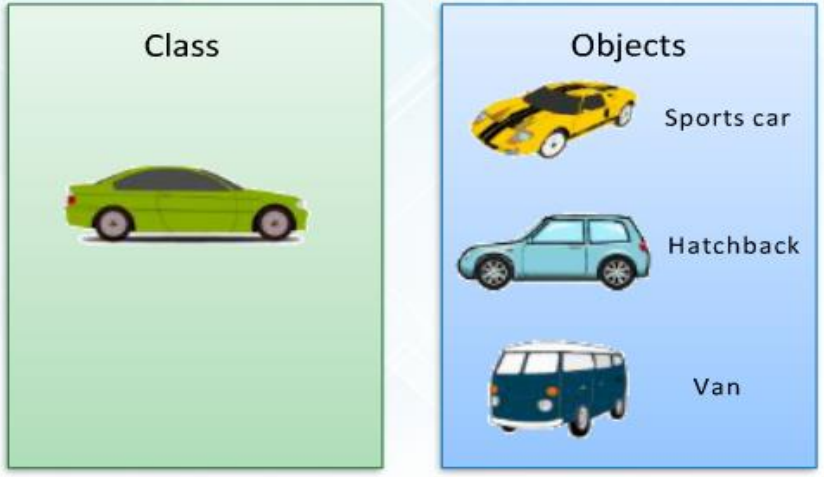
Lecture04

Class and object

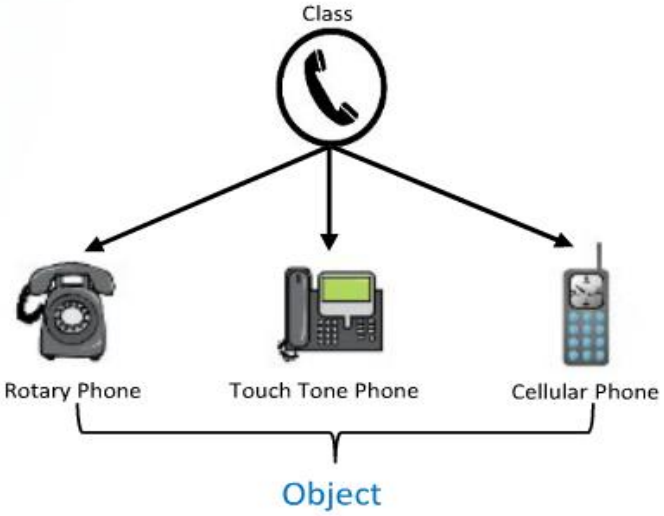
- In Python every thing is an object. To create objects we required some Model or Plan or Blue print, which is nothing but class.
- We can write a class to represent properties (attributes) and actions (behavior) of object.
- Properties can be represented by variables.
- Actions can be represented by Methods.
- Hence class contains both variables and methods.

Class and object

- **Class** is a blueprint used to create objects having same property or attribute as its class



- An **Object** is an instance of class which contains variables and methods



How to define a Class?

Syntax:

```
class className:

    ''' documentation 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)**

How to define a Class?

class Student:

```
        """ This is student class with required data"\"
print(Student.__doc__)
help(Student)
```

O/P : This is student class with required data
This is student class with required data ## with other information

Inside Class–3 types of variables & 3 types of method

Variables

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

Methods

1) Instance Methods
2) Class Methods
3) Static Methods

Example for Class

```
1) class Student:
2)         '''Developed by DPSharma for python demo'''
3)         def __init__(self):
4)                 self.name=`DPSharma'
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)
```

Object / Reference Variable

- Physical existence of a class is nothing but object. We can create any number of objects for a class.
- The variable which can be used to refer object is called reference variable. By using reference variable, we can access properties and methods of the object.
- **Syntax to Create Object:**
  - referencevariable = classname()
- **Example:**
  - s = Student()

Example for Class

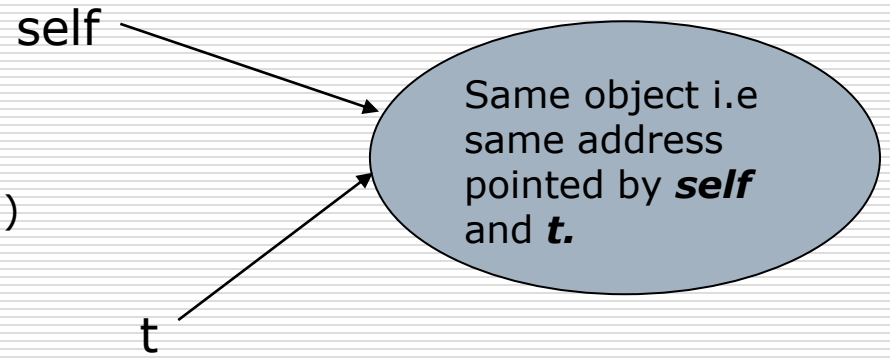
```
1) class Student:
2)         '''Developed by DPSharma for python demo'''
3)         def __init__(self, name,age,marks):
4)                 self.name=name
5)                 self.age=age
6)                 self.marks=marks
7)
8)         def talk(self):
9)                 print("Hello I am :",self.name)
10)                print("My Age is:",self.age)
11)                print("My Marks is:",self.marks)
12)
13) s1=Student("DPSharma",101,80)
14) s1.talk()
```

Class--→ Self Variable

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

- class Test:

```
    def __init__(self):
            print("Address of object, pointed by self", id(self))
t= Test()
print( "Address of object point by t", id(t))
```



- Inside the class , we can't use the object/reference variable for accessing the instance variables/methods.

- By using self , inside the class ,we can access instance variables and instance methods of object.

Class--→ Self Variable

- self should be first parameter inside constructor

```
def __init__(self):    ### Constructor
```

- self should be first parameter inside instance methods

```
def talk(self , a, b):     ####  Instance Method
```

- self value to the constructor or inside instance method need not to be pass by programmer , rather python environment will take care automatically.
- t= Test() # object creation , hence constructor get called by environment. Here again self value passed by environment .
- t. talk(10,20) ### self value again taken by environment automatically not passed by programmer.

Class--→ Self Variable

- The purpose of self variable is to declare and initiate the instance variable. we can use self to access the value of instance variable.

- In place self we can take any name. The pointer to the current object is not fixed with the name.

Class--→ 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, The "**constructor**" will be executed only once.
- Constructor can take at least one argument(at least self).

Class--→ Constructor Concept

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

- if you call constructor explicitly, then it will be executed like a normal method , but no new object will not be created.

- ***method overloading*** in python is not allowed. hence constructor overloading is also not allowed , however if we define more then two methods (constructor) with same name , the ***PVM*** will consider the last one. Hence to avoid the error, need to take more care when we supposed to call the method.

Method Vs constructor

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

Method name same as Class name

- Method name can be same as Class name; however it is recommended not to have the same name.

```
Class Test:
        def Test(self):
                print(" This is a special function")

t= Test()  #  Constructor will be executed __init()__
t.Test()   #   Test Method will be executed
```

Types of Variables:

- **Inside Python class 3 types of variables are allowed.**
  - Instance Variables (Object Level Variables)
  - Static Variables (Class Level Variables)
  - Local variables (Method Level Variables)

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:**
  - Inside Constructor by using self variable
  - Inside Instance Method by using self variable
  - Outside of the class by using object reference variable

Instance Variables

```
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__)
```

Inside Constructor

Instance Method

Outside of the class

Instance Variables

How to Access Instance Variables:

- We can access instance variables within the class by using self variable and outside of the class by using object reference.

How to delete Instance Variable from the Object:

- Within a class we can delete instance variable as follows

  del self.variableName

- From outside of class we can delete instance variables as follows

  del objectreference.variableName

Instance Variables

How to delete Instance Variable from the Object:

```
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.

Static Variables

- If the value of a variable is not varied from object to object, such type of variables we must declare with in the class, 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.

Static Variables

Various Places to declare Static Variables

- In general we can declare within the class directly but from outside of any method
- Inside constructor by using class name
- Inside instance method by using class name
- Inside class method by using either class name or cls variable
- Inside static method by using class name

Static Variables

```
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__)
```

Static Variables

How to access Static Variables:

- inside constructor: by using either self or classname
- inside instance method: by using either self or classname
- inside class method: by using either *cls* variable or classname
- inside static method: by using classname
- From outside of class: by using either object reference or classname

Static Variables

How to access Static Variables:

```
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)
8)          print(Test.a)
9)      @classmethod
10)     def m2(cls):
11)         print(cls.a)
12)         print(Test.a)
13)     @staticmethod
14)     def m3():
15)         print(Test.a)
```

```
16) t=Test()
17) print(Test.a)
18) print(t.a)
19) t.m1()
20) t.m2()
21) t.m3()
```

Static Variables

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.

```
1)  class Test:
2)      a=777
3)      @classmethod
4)      def m1(cls):
5)          cls.a=888
6)      @staticmethod
7)      def m2():
8)          Test.a=999
9)  print(Test.a)
10) Test.m1()
11) print(Test.a)
12) Test.m2()
13) print(Test.a)
```

Output

```
777
888
999
```

. . . . .

Static Variables

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.

```
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
```

Static Variables

If we change the Value of Static Variable by using either *self* OR *Object Reference* Variable:

```
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
```

Static Variables

If we change the Value of Static Variable by using either *self* OR *Object Reference* Variable:

```
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

Static Variables

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

Static Variables

****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.
- If we are trying to delete by reference variable then we will get error.

```
1)  class Test:
2)     a=10
3)
4)  t1=Test()
5)  del t1.a     ===>AttributeError: a
```

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.

Local Variables

```
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
```

```
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

- Instance Methods
- Class Methods
- Static Methods

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 must 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 Methods

- Inside method implementation, if we are using only class variables (static variables), then such type of methods ,should be 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 by object reference variable.

Class Methods

```
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()
```

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

Static Methods

```
1) class DPMath:
2)
3)      @staticmethod
4)      def add(x,y):
5)              print('The Sum:',x+y)
6)
7)      @staticmethod
8)      def product(x,y):
9)              print('The Product:',x*y)
10)
11)     @staticmethod
12)     def average(x,y):
13)             print('The average:',(x+y)/2)
14)
15) DPMath.add(10,20)
16) DPMath.product(10,20)
17) DPMath.average(10,20)
```

Output

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

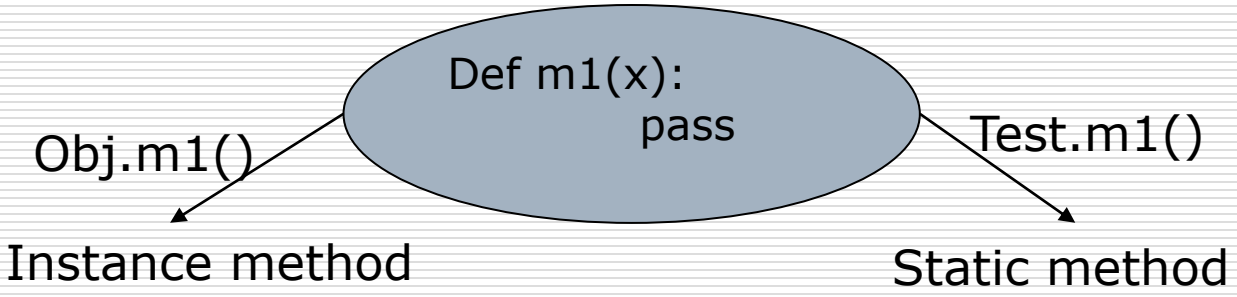
Methods of Classes – General View

Instance Method	Class Method	Static Method
If at least one variable is instance variable ,then method must be declared as *instance method*	static variable. No Instance variable. In such case method type must be *class method*.	No instance variable, No static variable. In such case method type must be *static method*.
No decorator is required.	@classmethod--(Mandatory)	@staticmethod—(Optional)
self is required.	***cls*** is required.	No ***self*** , No ***cls***.
Obj reference for calling	**Class Name** or **Obj reference** for calling	**Class Name** or **Obj reference** for calling
Eg1. instance variable & static variable --- instance method. **Eg2.** instance variable & local variable.	**Eg1.** static variable and local variable	Only local variable.

Methods of Classes – General View

Note:

- For class method : @classmethod--(Mandatory).
- For static method : @staticmethod—(Optional).
- So if no decorator ,then method may be a static or instance method.



- Hence, based upon its calling, method can be treated as instance or static.

Methods of Classes – General View

```
class Test:
        def m1():
                print("some method")
T=Test()
T.m1()  ### o/p error as this will treated as instance method , no self variable
                declaration
Test.m1()  ### o/p  some method
```

Methods of Classes – General View

```
class Test:
        def m1(x):
                print("some method")
T=Test()
T.m1()  ### o/p: some method , here x will act as self. Instance method
T.m1(10)  ### o/p  error  as passing two arguments but defined one variable.
Test.m1() #### o/p error  invalid as function is static but no arguments passed at
                          the time of calling.
Test.m1(10) #### o/p: some method.
```

Methods of Classes – General View

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.

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:
                ......
```

Inner Classes

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 existing ***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.

Inner Classes

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()
```

Inner Classes

Demo Program-2:

```
1) class Person:
2)         def __init__(self):
3)                 self.name='DPSharma'
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: DPSharma
Dob=10/5/1947
```

Inner Classes

Demo Program-2: Modified

```
1) class Person:
2)         def __init__(self,name,dd,mm,yyyy):
3)                 print("Person Object Creation…")
4)                 self.name=name
5)                 self.db=self.Dob(dd,mm,yyyy)
6)         def info(self):
7)                 print('Name:',self.name)
8)                 self.db.display()
9)         class Dob:
10)                def __init__(self,dd,mm,yyyy):
11)                        print("Dob Object Creation…")
12)                        self.dd=dd
13)                        self.mm=mm
14)                        self.yyyy=yyyy
15)                def display(self):
16)                        print('Dob={}/{}/{}'.format(self.dd,self.mm,self.yyyy))
14) p=Person('DPSharma',28,8,1947)
15) p.info()
```

Output

```
Person Object Creation…
Dob Object Creation
Name: DPSharma
Dob=10/5/1947
```

Nested Function

```
Class Test:
        def m1(self):
                def cal(a,b):
                        print(" The Sum is = ",a+b)
                        print(" The Product is = ",a*b)
                        print(" The Difference is = ",a-b)
                        print(" The Average is = ",(a+b)/2)
                        print()
                cal(10,20)
                cal(100,200)
T=Test()
T.m1()
```

Garbage Collection

- In old languages like C++,Java etc 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.

Garbage Collection -*Enable and Disable*

- By default Garbage 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

Q. Why Disable??
Ans. To Improve the performance

```
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. Destroying the object will take care by PVM.
- Reference count mechanism is used for deletion.

Destructors:

```
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
```

Destructors:

```
class Test:
        def __init__(self):
                print("Object Initialization...")
        def __del__(self):
                print("Fulfilling Last Wish and performing clean up
                      activities...")
t1=Test()
t2=Test()
print("End of application")
```

Output

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

Destructors:

```
class Test:
        def __init__(self):
                print("Object Initialization...")
        def __del__(self):
                print("Fulfilling Last Wish and performing clean up
                      activities...")
t1=Test()
t2=Test()
T1=None
T2=None
print("End of application")
```

Output

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

Destructors:

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

Destructors:

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

Output

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

Destructors:

Q. Difference between ***del t*** and ***t=None*** .?

Ans:

del t -----→ Both ***Object*** and ***reference variable*** , will be deleted.

T=None ----→ Object will be deleted , However reference variable will be exist and going to point ***None*** object.

Q. How to find the number of references of an object?

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

sys.getrefcount (objectreference)

```
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.

Members of One class inside another class

- We can use members of One class inside another class by using the following two ways:

1. By Composition (HAS-A Relationship)
2. By Inheritance (IS-A Relationship)

HAS-A relationship

- By creating an object, we can access members of one class inside another class. This approach is nothing but composition or HAS-A relationship.

- The main advantage of HAS-A relationship is code reusability.

HAS-A relationship

```
Class Engine:
        def  useEngine(self):
                print("Engine Specific Functionality")

Class Car:
        def __init__(self):
                self.engine=Engine()
        def  useCar(self):
                print(" Car required Engine Functionality")
                self.engine.useEngine()
C=Car()
C.useCar()
```

Note:

Class Car HAS_A Class Engine reference.

HAS-A relationship

```
class Car:
    def __init__(self,name,model,color):
            self.name=name
            self.model=model
            self.color=color
    def getinfo(self):
            print('\tCarName:{}\n\tModel:{}\n\tColor:{}'.format(self.name,self.model,self.color))
```

HAS-A relationship

```
class Employee:
        def __init__(self,name,age,eno,esal,car):
                self.name=name
                self.age=age
                self.eno=eno
                self.esal=esal
                self.car=car
        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 Information:')
                self.car.getinfo() # Employee using Car Functionality
```

HAS-A relationship

```
car=Car('Innova','2.5V','Grey')
e=Employee('DPSharma',48,Muj0055,10000,car)
e.empinfo()
```

IS-A Relationship

- Parent to Child Relationship.
- Parent Class members are by default available to the child class and hence child can reuse parent class functionality without rewriting.(Code Reusability).
- Child Class can also define new members. Hence child class can extend Parent class functionality.(Code Extendibility)

IS-A Relationship

```
Class P:
        def m1(self):
                print("Parent Method")
Class C(P):
        def m2(self):
                print("Child Method")
Ob=C()
Ob.m1()   #
Ob.m2()   #
```

IS-A Relationship

```
Class P:
        a=10
        def __init__(self)
                print("Parent Constructor")
                self.b=20
        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
Ob=C()
```

Child class does not have anything even in this case constructor is not also define. In such cases the constructor of Parent Class will be executed.

O/P:

Parent Constructor

IS-A Relationship

```
Class P:
        a=10
        def __init__(self)
                print("Parent Constructor")
                self.b=20
        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
Ob=C()   # Child class Object
print(Ob.a)  # parent class Static Variable
print(Ob.b)  # parent class Instance Variable
Ob.m1       # parent class Instance Method
Ob.m2       # parent class Class Method
Ob.m3       # parent class Static Method
```

O/P:

```
Parent Constructor
10
20
Parent Instance Method
Parent Class Method
Parent Static Method
```

IS-A *Vs* HAS-A

- If we want to extend existing functionality with some more extra functionality then we should go for IS-A i.e Inheritance.

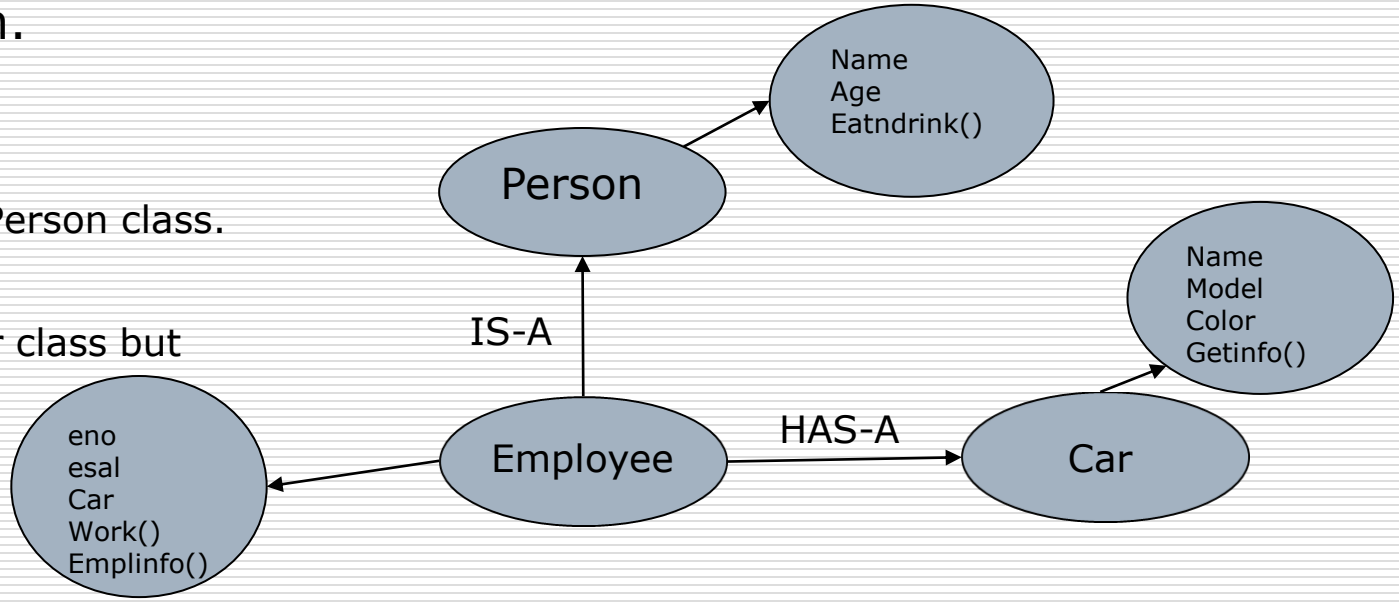
- If we do not want to extend and just we have to use existing functionality then we should go for HAS-A i.e Composition.

- Employee ***IS-A*** Person.

  Employee extending the functionality of Person class.

- Employee ***HAS-A*** Car.

  Employee is using the functionality of Car class but not extending the functionality



IS-A *Vs* HAS-A

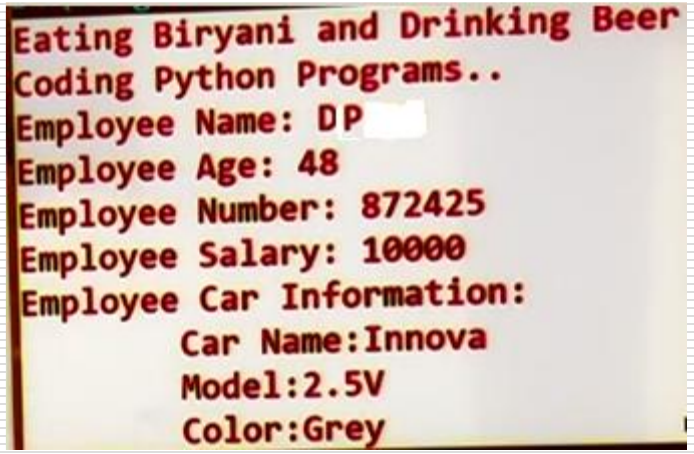
```
class Car:
    def __init__(self,name,model,color):
        self.name=name
        self.model=model
        self.color=color
    def getinfo(self):
        print('\tCar Name:{}\n\tModel:{}\n\tColor:{}'.format(self.name,self.model,self.color))

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

IS-A *Vs* HAS-A

```
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 Programs..')
    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 Information:')
            self.car.getinfo() # Employee using Car Functionality
```

IS-A *Vs* HAS-A

```
car=Car('Innova','2.5V','Grey')
e=Employee('DP',48,872425,10000,car)
e.eatndrink() #Employee using Person class functionality
e.work()
e.empinfo()
```



Composition

```
class University:
        Def __inti(self):
                self.department=self.department()

        class Department:
                pass

U=University()
```

Department Class/object can not exist without existence of University class/object.

Composition

- Without existing container object , if there is no chance of existing contained objects then container and contained objects are strongly associated and this association is nothing but composition.

- Eg. University contains several Departments. Without existing University, there is no chance of existing Department object. Hence University and Department 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 container and contained objects are weakly associated and this weak association is nothing but aggregation.

- Eg. Several professors may work in the Department. Without existing Department still there may be a chance of existing professor. Hence Department and Professor objects are weakly associated and this weak association is nothing but aggregation.

Aggregation

```
class Professor:
        pass
Class Department:
        def __inti__(self,professor):
                self.professor=professor

professor=Professor()
csdept=Department(professor)
itdept=Department(professor)
```

Composition *Vs* Aggregation

- In Composition objects are strongly associated where as in Aggregation objects are weakly associated.

- In Composition, container object holds directly Contained objects, while as in Aggregation container object just holds references of Contained objects.

Inheritance -- Type

- Single Inheritance
- Multi Level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance
- Cyclic Inheritance

Single Inheritance

- The concept of inheriting members from one class to another class is known as single inheritance.

- Single parent ***and*** single child. i.e ***one to one***.

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

Multi Level Inheritance

- The concept of inheriting members from multiple classes to a single class with the concept of one after another is known as multi level inheritance.
- Eg. Multiple levels of inheritance =➔ Multi Level 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()
c.m1()
c.m2()
c.m3()
```

Hierarchical Inheritance

- The concept of inheriting members from one class to multiple classes which present at same level is known as Hierarchical Inheritance.
- Eg. One Parent but Multiple child classes and all child classes are at same level.

```
class P:
        def m1(self):
                print ("Parent  Method")
class C1(P):
        def m2(self):
                print ("Child1  Method")
class C2(P):
        def m3(self):
                print ("Child2  Method")

c1=C1()          c2=C2()          C1.m3()  ### Attribute Error
c1.m1()          c2.m1()          C2.m2()  ### Attribute Error
c1.m2()          c2.m3()
```

Multiple Inheritance

- Reverse of Hierarchical Inheritance.
- **Hierarchical :** One Parent and Multiple Child classes.
- **Multiple :** Multiple Parents and Single Child class.
- The concept of inheriting the members from multiple classes to a single class at a time is known as multiple inheritance.
- Eg. Multiple Parents but Single Child.

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 (“Child  Method”)
c=C()
c.m1()
c.m2()
c.m3()
```

Multiple Inheritance

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

- 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()  ### Parent P1 get chance.
```

Hybrid Inheritance

- Hybrid means : mixing / combination
- Combination of Single, Multilevel , Multiple and Hierarchical inheritances is known as Hybrid inheritance.
- **Note:** In Hybrid inheritance , method resolution is based on MRO algorithm.

Cyclic Inheritance

- The concept of inheriting members from one class to another class in cyclic way , is called cyclic inheritance.

- Note : Really Cyclic inheritance is not required. Hence programming languages like java , python won’t provide support.

Method Resolution Order(MRO)

- In Hybrid inheritance the method resolution order is decided based on MRO algorithm.
- We can find MRO of any class by using ***mro()*** function.
- Print(classname.mro())

Method Resolution Order(MRO)

- 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)
- Where X is Class , and P1,P2 etc are immediate parents. i.e We have to consider the immediate parents

Method Resolution Order(MRO)

Head Element vs Tail Terminology:

- Assume C1,C2,C3,...is a list of classes.
- In the list: C1C2C3C4C5....
- First element is considered as Head Element and Remaining is considered as Tail Part.
- Head Element: C1
- Tail Part : C2C3C4…..

Method Resolution Order(MRO)

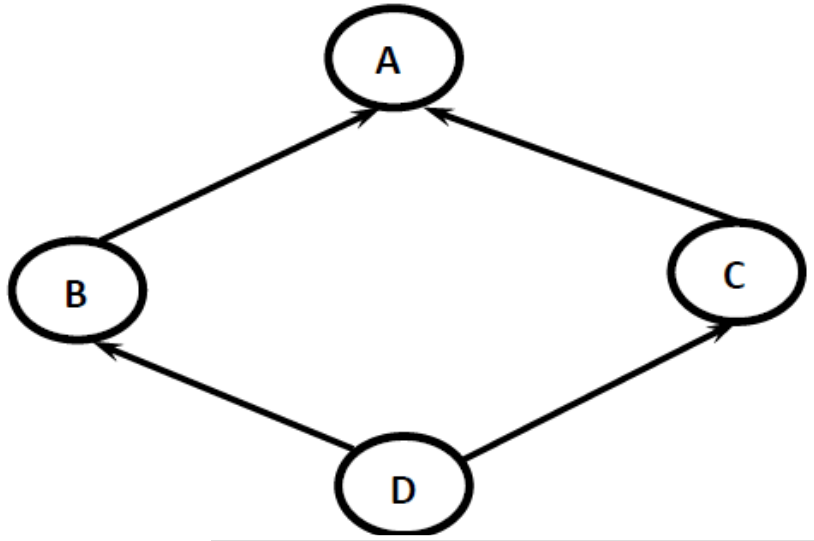
How Merge Works

- Take the head of first list.
- If the head is not in the tail part of any other list, then add this head element to the result and remove it from all the lists.
- 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.

Demo Program-1 for Method Resolution Order:

```
1) class A:pass
2) class B(A):pass
3) class C(A):pass
4) class D(B,C):pass
5) print(A.mro())
6) print(B.mro())
7) print(C.mro())
8) print(D.mro())
```

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



Output:

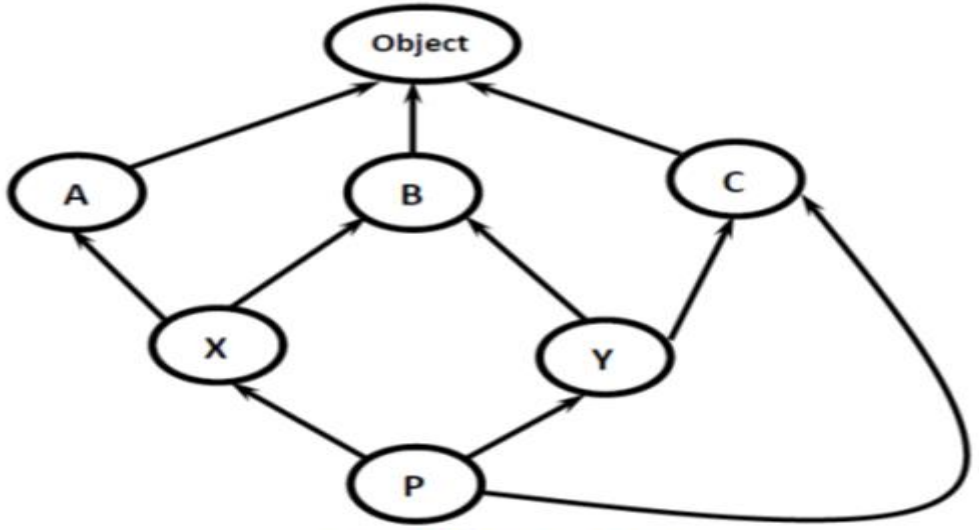
```
[<class '__main__.A'>, <class 'object'>]
[<class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

Demo Program-2 for Method Resolution Order:

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



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

Demo Program-2 for Method Resolution Order:

```
1)  class A:
2)      def m1(self):
3)          print('A class Method')
4)  class B:
5)      def m1(self):
6)          print('B class Method')
7)  class C:
8)      def m1(self):
9)          print('C class Method')
```

```
10) class X(A,B):
11)     def m1(self):
12)         print('X class Method')
13) class Y(B,C):
14)     def m1(self):
15)         print('Y class Method')
16) class P(X,Y,C):
17)     def m1(self):
18)         print('P class Method')
19) p=P()
20) p.m1()
```

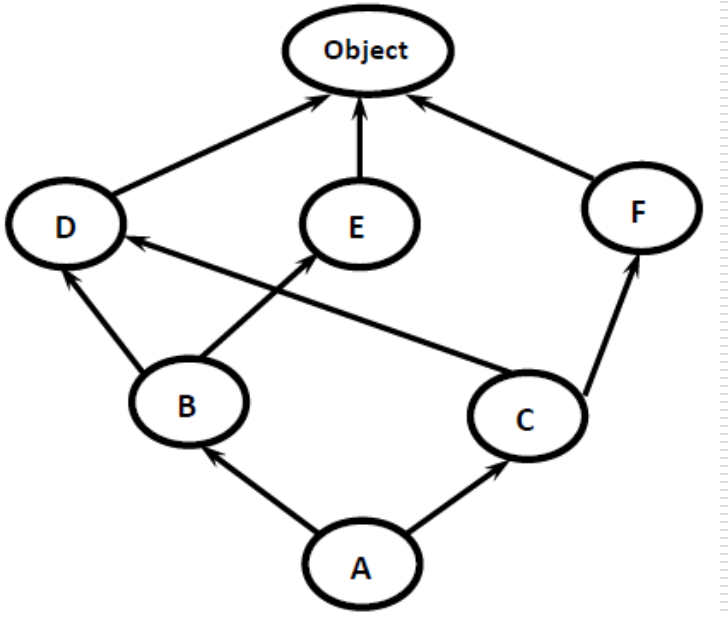
In this example P class m1() Method will be considered. If P class does not contain m1() method then as per MRO, X class method will be considered. If X class does not contain then A class method will be considered and this process will be continued.
The method resolution in the following order: PXAYBCO

Output: P class Method

Demo Program-3 for Method Resolution Order:

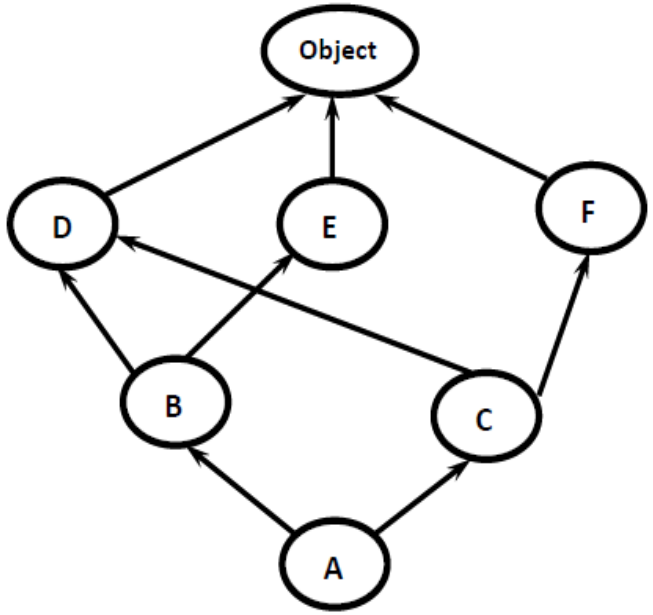
mro(o) = object
mro(D) = D,object
mro(E) = E,object
mro(F) = F,object
mro(B) = B,D,E,object
mro(C) = C,D,F,object

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



Demo Program-3 for Method Resolution Order:

```
1)  class D:pass
2)  class E:pass
3)  class F:pass
4)  class B(D,E):pass
5)  class C(D,F):pass
6)  class A(B,C):pass
7)  print(D.mro())
8)  print(B.mro())
9)  print(C.mro())
10) 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

- Parents class members are by default available to the child class. In the child class we can access parent class members directly.
- If parent class and child class contains a member with the same name, then to call explicitly parent class members from the child class we should use super().
- super() is a built-in method which is useful to call the super class constructors, variables and methods explicitly from the child class.

Demo Program-1 for super():

20)
s1=Student('DPSharma',22,101,90)

21)
s1.display()

Output:
Name: DPSharma
Age: 22
Roll No: 101
Marks: 90

In the above program super() method has used to call parent class constructor and display() method.

```
1)  class Person:
2)      def __init__(self,name,age):
3)          self.name=name
4)          self.age=age
5)      def display(self):
6)          print('Name:',self.name)
7)          print('Age:',self.age)
8)
```

```
9)  class Student(Person):
10)     def __init__(self,name,age,rollno,marks):
11)         super().__init__(name,age)
12)         self.rollno=rollno
13)         self.marks=marks
14)
15)     def display(self):
16)         super().display()
17)         print('Roll No:',self.rollno)
18)         print('Marks:',self.marks)
```

How to Call Method of a Particular Super Class: In-Multilevel Inheritance

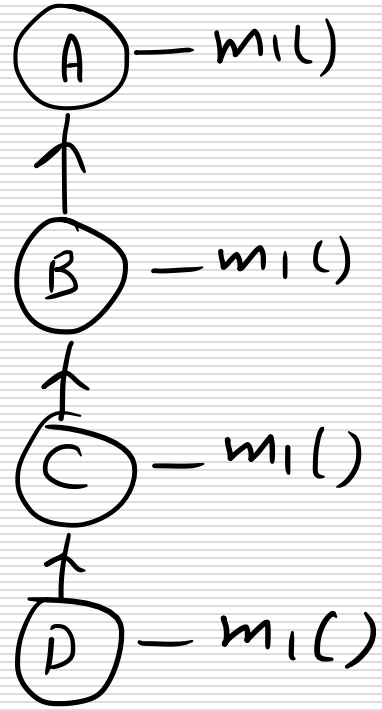
We can use the following approaches

1. super(D, self).m1()

   *It will call m1() method of **super class of D**.*

1. A.m1(self)

   *It will call **A class** m1() 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().

```
1)  class P:
2)      a=10
3)      def __init__(self):
4)          self.b=20
5)
6)  class C(P):
7)      def m1(self):
8)          print(super().a)#valid
9)          print(self.b)#valid
10)         print(super().b)#invalid
11) c=C()
12) c.m1()
```

Output:

10
20

AttributeError: 'super' object has no attribute 'b`.

Various Important Points about super():

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

Output:

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

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)      def m1(self):
5)          print('Parent instance method')
6)      @classmethod
7)      def m2(cls):
8)          print('Parent class method')
9)      @staticmethod
10)     def m3():
11)         print('Parent static method')
12)
```

```
13) class C(P):
14)     def __init__(self):
15)         super().__init__()
16)         super().m1()
17)         super().m2()
18)         super().m3()
19)
20)     def m1(self):
21)         super().__init__()
22)         super().m1()
23)         super().m2()
24)         super().m3()
25)
26) c=C()
27) c.m1()
```

Various Important Points about super():

- **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.

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)      def m1(self):
5)          print('Parent instance method')
6)      @classmethod
7)      def m2(cls):
8)          print('Parent class method')
9)      @staticmethod
10)     def m3():
11)         print('Parent static method')
12)
```

```
13) class C(P):
14)     @classmethod
15)     def m1(cls):
16)         #super().__init__()--->invalid
17)         #super().m1()--->invalid
18)         super().m2()
19)         super().m3()
20)
21) C.m1()
```

Output:

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

Various Important Points about super():

- **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.

- **Reason :** Class method no way related to object. Without object also we can call class method. But constructor and instance methods are always associated with object.

Various Important Points about super():

- **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.
- From Class Method of Child Class, how to call Parent Class Instance Methods and Constructors:(Indirect method)

```
1)  class A:
2)      def __init__(self):
3)          print('Parent constructor')
4)
5)      def m1(self):
6)          print('Parent instance method')
7)
8)  class B(A):
9)      @classmethod
10)     def m2(cls):
11)         super(B,cls).__init__(cls)
12)         super(B,cls).m1(cls)
13)
14) B.m2()
```

Output:
Parent constructor
Parent instance method

Various Important Points about super():

- **Case-4:** In child class ***static method*** , we cannot use super(), to call parent class members . (But indirectly we can call parent class static and class methods)

```
1)  class P:
2)      def __init__(self):
3)          print('Parent Constructor')
4)      def m1(self):
5)          print('Parent instance method')
6)      @classmethod
7)      def m2(cls):
8)          print('Parent class method')
9)      @staticmethod
10)     def m3():
11)         print('Parent static method')
12)
```

```
13) class C(P):
14)     @staticmethod
15)     def m1():
16)         super().m1()-->invalid
17)         super().m2()--->invalid
18)         super().m3()--->invalid
19)     Super().__init__() ---> invalid
20) C.m1()
```

RuntimeError:

super(): no arguments

Various Important Points about super():

- **Case-4:** In child class static method , we cannot use super(), to call parent class members . (But indirectly we can call parent class static and class methods).
- ***How to Call Parent Class Static Method from Child Class Static Method by using super():***

```
1) class A:
2)       @staticmethod
3)       def m1():
4)                print('Parent static method')
5)  class B(A):
6)       @staticmethod
7)       def m2():
8)                super(B,B).m1()
9) B.m2()
```

Output: Parent static method

POLYMORPHISM

- **Poly** means **Man**y.
- **Morphs** means **Forms**.
- Polymorphism means **'Many Forms'**.

- One name but multiple forms is the concept of polymorphism.

- Eg1: Yourself is best example of polymorphism. In front of Your parents You will have one type of behavior and with friends another type of behavior. Same person but different behaviors at different places, which is nothing but polymorphism.
- Eg2: + operator acts as concatenation and arithmetic addition. Operator Overloading .
- Eg3: * operator acts as multiplication and repetition operator.
- Eg4: The Same method with different implementations in Parent class and child
- classes.(overriding , this is also polymorphism.)

Related to Polymorphism

- **Overloading:**
  - Operator Overloading
  - Method Overloading
  - Constructor Overloading
- **Overriding**
  - Method Overriding
  - Constructor Overriding
- **Pythonic Behavior**
  - Duck Typing
  - Easier to Ask Forgiveness than Permission(EAFP)
  - Monkey Patching

Polymorphism→ 1.Operator Overloading

- We can use the same operator for multiple purposes, which is nothing but operator overloading.
- Python supports operator overloading. Java provide very limited support for operator overloading.

- Eg 1: + operator can be used for Arithmetic addition and String concatenation.

  - print(10+20)#30
  - print(‘DP'+‘Sharma')#DPSharma

- Eg 2: * operator can be used for multiplication and string repetition purposes.

  - print(10*20)#200
  - print(‘DPSharma'*3)# DPSharmaDPSharmaDPSharma

Demo program to use + operator for our class objects:

```
1) class Book:
2)       def __init__(self,pages):
3)                 self.pages=pages
4) b1=Book(100)
5) b2=Book(200)
6) print(b1+b2)
```

O/P:

```
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 use + operator for our class objects:

```
1) class Book:
2)         def __init__(self,pages):
3)                 self.pages=pages
4)         def __add__(self,other):
5)                 return self.pages+other.pages
6) b1=Book(100)
7) b2=Book(200)
8) print('The Total Number of Pages:',b1+b2)
9) print('The Total Number of Pages:',b1+b2+b3)
```

int Book → both are not compatible.

Output: The Total Number of Pages: 300

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:

```
1)  class Student:
2)      def __init__(self,name,marks):
3)          self.name=name
4)          self.marks=marks
5)      def __gt__(self,other):
6)          return self.marks>other.marks
7)      def __le__(self,other):
8)          return self.marks<=other.marks
9)  print("10>20 =",10>20)
10) s1=Student("Durga",100)
11) s2=Student("Ravi",200)
12) print("s1>s2=",s1>s2)
13) print("s1<s2=",s1<s2)
14) print("s1<=s2=",s1<=s2)
15) print("s1>=s2=",s1>=s2)
```

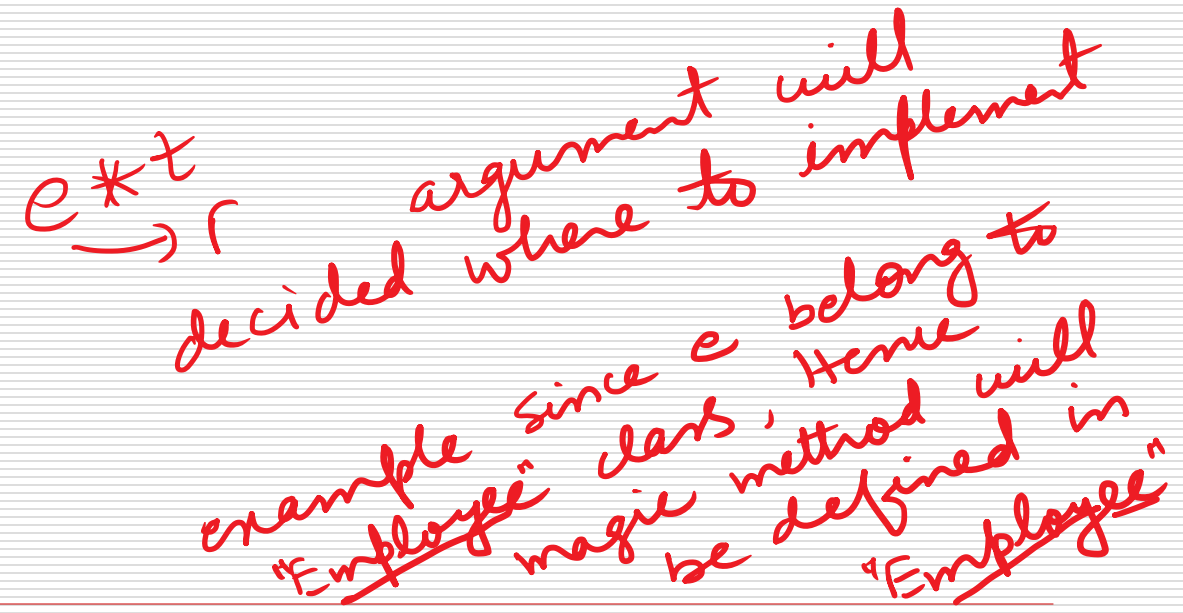
Output

```
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:

```
1)  class Employee:
2)      def __init__(self,name,salary):
3)              self.name=name
4)              self.salary=salary
5)      def __mul__(self,other):
6)              return self.salary*other.days
7)  class TimeSheet:
8)      def __init__(self,name,days):
9)              self.name=name
10)             self.days=days
11) e=Employee('Durga',500)
12) t=TimeSheet('Durga',25)
13) print('This Month Salary:',e*t)
```

Output: This Month Salary: 12500



Importance of __str__() method:

- Whenever we are printing any object reference, internally __str__() method will be called ,which is returns string in the following format.

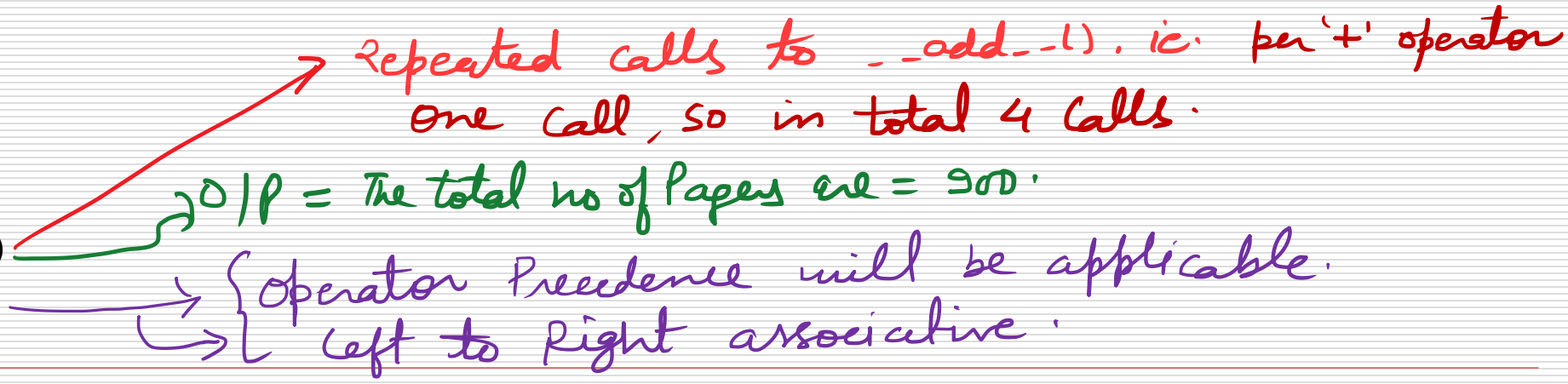
<__main__.classname object at 0x022144B0>

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

```
1)  class Student:
2)      def __init__(self,name,rollno):
3)          self.name=name
4)          self.rollno=rollno
5)      def __str__(self):
6)          return 'This is Student with Name:{} and Rollno:{}'.format(self.name,self.rollno)
7)  s1=Student('DP',101)
8)  s2=Student('Ravi',102)
9)  print(s1)
10) print(s2)
```

Operator overloading –Advance Example

```
class Book:
        def __init__(self,nopages):
                self.nopages=nopages
        def __add__(self,other):
                return Book(self.nopages+other.nopages)
        def __mul__(self,other):
                return Book(self.nopages*other.nopages)
        def __str__(self):
                return ` The total Number of pages are ={}.'.format(self.nopages)
B1=Book(100)
B2=Book(200)
B3=Book(300)
B4=Book(400)
print(B1+B2)
print(B1+B2+B3+B4)
print(B1*B2+B3*B4)
```



2.Method Overloading

- If 2 or more 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.

Method Overloading

Demo Program:

```
1)  class Test:
2)      def m1(self):
3)          print('no-arg method')
4)      def m1(self,a):
5)          print('one-arg method')
6)      def m1(self,a,b):
7)          print('two-arg method')
8)  t=Test()
9)  #t.m1()
10) #t.m1(10)
11) t.m1(10,20)
```

Output:
two-arg method

In the above program python will consider only last method.

Method Overloading-case1

- Two methods are said to be overloaded if both methods having same name , but different arguments types.

  Eg :

  sqrt(int)

  sqrt(float)

- But in Python , we cannot declare type explicitly. Based on provided value , type will be considered automatically (Dynamically Typed). As type concept is not applicable. Method overloading concept is not applicable in python.

Method Overloading-case1

- The advantage of Dynamic Type is that we need not to defined different functions for different data types.

```
class Test:
        def m1(self,x):
                print(" {}-argument method".format(x.__class__.__name__))
t = Test()
t.m1(10)
t.m2(10.5)
t.m3("DPSharma")
```

→ type will be int.

O/P int--argument method.
O/P float--argument method.
O/P str--argument method.

→ Since type of x will be decided at run time, Hence no need to write separate functions for each type.

Method Overloading-case2

- Two methods are said to be overloaded if both methods having same name , but having different number of arguments.

eg m1(a): OR m1(a,b) OR m1(a,b,c) etc // Different arguments.

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 this with ***default arguments*** or with ***variable number of argument*** methods.

Method Overloading-case2

Demo Program with Default Arguments:

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

Output

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

Method Overloading-case2

Demo Program with Variable Number of Arguments:

```
1)  class Test:
2)      def sum(self,*a):
3)          total=0
4)          for x in a:
5)              total=total+x
6)          print('The Sum:',total)
7)
8)  t=Test()
9)  t.sum(10,20)
10) t.sum(10,20,30)
11) t.sum(10)
12) t.sum()
```

3) Constructor Overloading:

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

```
1)  class Test:
2)      def __init__(self):
3)          print('No-Arg Constructor')
4)      def __init__(self,a):
5)          print('One-Arg constructor')
6)      def __init__(self,a,b):
7)          print('Two-Arg constructor')
8)  #t1=Test()
9)  #t1=Test(10)
10) t1=Test(10,20)
```

Output:

Two-Arg constructor

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

Overriding

- What ever members available in the parent class are by default 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.

Overriding

Demo Program for Method Overriding:

```
1)  class P:
2)      def property(self):
3)              print('Gold+Land+Cash+Power')
4)      def marry(self):
5)              print('Arrange Marriage')
6)  class C(P):
7)      def marry(self):
8)              print('Love Marriage')
9)  c=C()
10) c.property()
11) c.marry()
```

Output:

```
Gold+Land+Cash+Power
Love Marriage
```

Overriding

Demo Program for Method Overriding:

```
1)    class P:
2)        def property(self):
3)                print('Gold+Land+Cash+Power')
4)        def marry(self):
5)                print('Arrange Marriage')
6)     class C(P):
7)        def marry(self):
8)                super().marry()
9)                print('Love Marriage')
10)     c=C()
11)   c.property()
12)   c.marry()
```

Output:

```
Gold+Land+Cash+Power
Arrange Marriage
Love Marriage
```

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

Overriding

Demo Program for Constructor Overriding:

```
1) class P:
2)         def __init__(self):
3)                   print('Parent Constructor')
4) class C(P):
5)         pass
6) c=C()
```

O/P:
Parent Constructor

Note:
if child class does not contain constructor, then parent class constructor will be executed

Overriding

Demo Program for Constructor Overriding:

```
1) class P:
2)         def __init__(self):
3)                 print('Parent Constructor')
4) class C(P):
5)         def __init__(self):
6)                 print('Child Constructor')
7) c=C()
```

O/P:
Child Constructor

Overriding

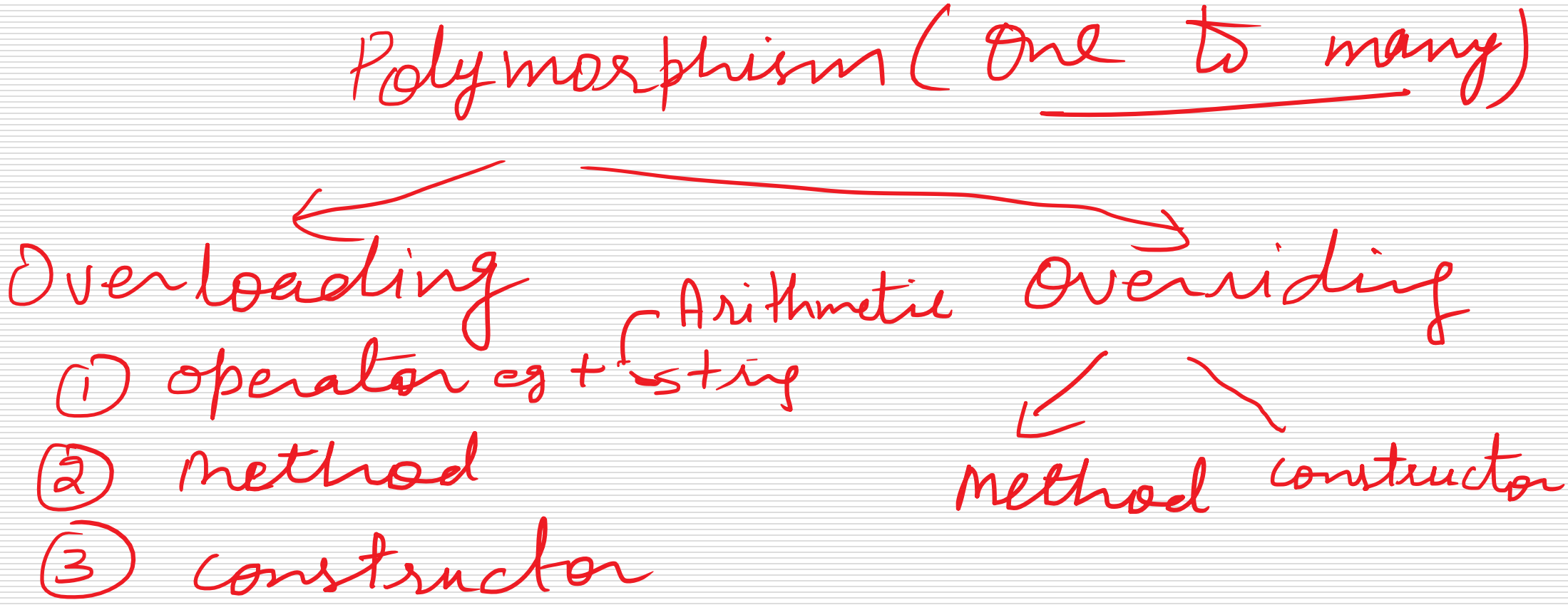
Demo Program for Constructor Overriding:

```
1) class P:
2)         def __init__(self):
3)                 print('Parent Constructor')
4) class C(P):
5)         def __init__(self):
6)                 super().__inti__()
7)                 print('Child Constructor')
8) c=C()
```

O/P:

Parent Constructor
Child Constructor

Polymorphism-Summary



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 (i.e empty 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.

Abstract Method:

```
1) from abc import abstractmethod
2) class Vehicle:
3)         @abstractmethod
4)         def noofwheels(self):
                    Pass
```

Child classes are responsible to provide implementation 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.

  Abstract class ===➔ Partially Implemented class

- Every abstract class in Python should be derived from ***ABC*** class, which is present in ***abc*** module.

  ABC===➔ Abstract Base Class

Abstract Class:

```
6)  class Bus(Vehicle):
7)      def noofwheels(self):
8)          return 6
9)  class Auto(Vehicle):
10)     def noofwheels(self):
11)         return 3
```

abstract class

abstract method

derived class responsible for implementing abstract method

```
12) b=Bus()
13) print(b.noofwheels())#6
14) a=Auto()
15) print(a.noofwheels())#3
```

BC, abstractmethod
:
od
self):

Abstract Class & Methods - FAQ

- What is abstract method?
  The method which has only declaration but not implementation (i.e. empty implementation).
- How to declare abstract method?
  By using ***@abstractmethod*** decorator.
- What is abstract class?
  Partially implemented class is nothing but abstract class.
- How to declare abstract class in python?
  The class should be child class of ABC.
- Who is responsible to provide implementation for parent class abstract methods?
  Child classes are responsible to provide implementation for parent class abstract methods.
- What is the advantage of declaring abstract methods in Parent class ?
  By declaring abstract methods in Parent class , we can provide guidelines to child classes, such that which methods, compulsory they should implement.

Important Conclusions of Abstract Class & Methods

Case-1:

```
1) class Test:
2)       pass
3) t=Test()
```

complete class

Case-2:

```
1) from abc import *
2) class Test(ABC):
3)       pass
4) t=Test()
```

① Inspite of not having Abstract method, this class is Abstract as it is a child class of class ABC.

② This class contains '0' no of abst. method

③ ie Abstract class with zero Abstract method is possible

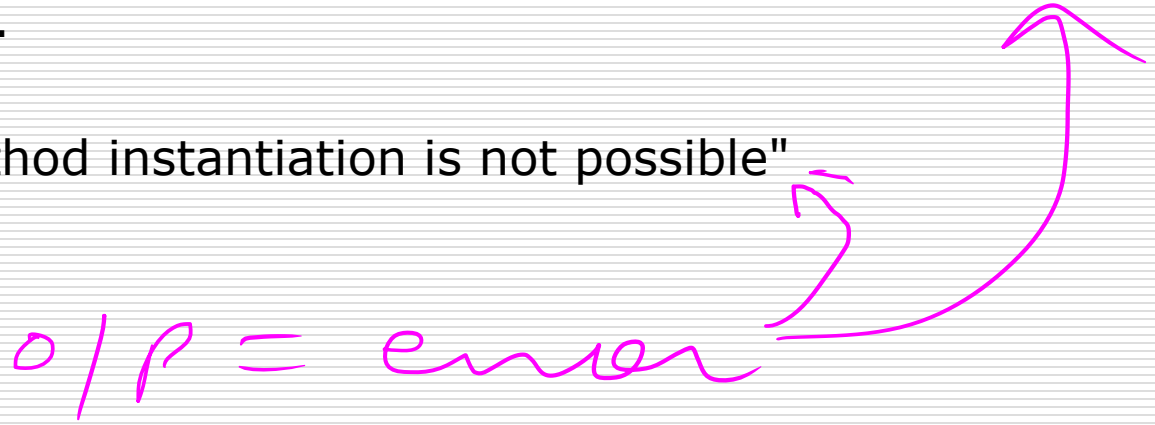
④ since this class does not contain abstract method, Hence object of class 'Test' is possible.

NOTE: Point 1–4 for Case-2.

Important Conclusions of Abstract Class & Methods

- If a class contains at least one abstract method and if we are extending ABC class then instantiation is not possible.

- "abstract class with abstract method instantiation is not possible"



Case -3:

```
1) from abc import *
2) class Test(ABC):
3)         @abstractmethod
4)         def m1(self):
5)                 pass
6) t=Test()
```

TypeError:

Can't instantiate abstract class Test with abstract methods m1.

Important Conclusions of Abstract Class & Methods

Case -4:

```
1) from abc import *
2) class Test:
3)         @abstractmethod
4)         def m1(self):
5)                 pass
6) t=Test()
```

We can create object even class contains abstract method, because we are not extending **ABC** class.

This programm will work perfectly.

NOTE: Since class "Test" is not a subclass of class "ABC". Hence this class "Test" is not a Abstract class (inspite of having abstract method)

Important Conclusions of Abstract Class & Methods

- If we are creating Child class(es) from Parent abstract abstract class , then for ***every abstract methods*** of parent class , compulsory we should provide implementation in child class(es) , otherwise child class is also abstract and we can not create object for child class.

- **Note:** Abstract class (Parent/Child) can contain both abstract and non-abstract methods also.

Interfaces in Python

Directly interface concept is not supported in python. ie no 'interface' keyword.

- An abstract class can contains both abstract and non-abstract methods.
- If an abstract class contains only abstract methods, such type of abstract class is nothing but interface.
- 100% pure abstract class is nothing but interface.
- Interface simply acts as Software/Service Requirement Specification(SRS).

while gathering client requirement, the functionality/services required by client, as a software developer, you can use interface as a tool.

Concreate class vs Abstract Class vs Inteface

- If we don't know anything about implementation and just we have requirement specification, then we should go for interface.(SRS-Service Requirement Specification)

- If we are talking about implementation but not completely, then we should go for abstract class. (partially implemented class).

- If we are talking about implementation completely and ready to provide service ,then we should go for concrete class. (Fully Implemented class)

Concreate class vs Abstract Class vs Inteface

```
1)  from abc import *
2)  class CollegeAutomation(ABC):
3)      @abstractmethod
4)      def m1(self): pass
5)      @abstractmethod
6)      def m2(self): pass
7)      @abstractmethod
8)      def m3(self): pass
```

Interface: As no Implementation

```
9)  class AbsCls(CollegeAutomation):
10)     def m1(self):
11)         print('m1 method implementation')
12)     def m2(self):
13)         print('m2 method implementation')
```

Abstract class
m_1, m_2 : Implemented
m_3: NOT Implemented

```
14) class ConcreteCls(AbsCls):
15)     def m3(self):
16)         print('m3 method implemnentation')
```

Concrete class
All method Implemented

```
17) c=ConcreteCls()
18) c.m1()
19) c.m2()
20) c.m3()
```

Public, Private and Protected Members

Public:

1. If a member(either method or variable) is public, then we can access that member from anywhere , either within the class or from outside the class.

2. By default every member present in python is public. We can access from anywhere either within the class or from outside of the class.

```
class Test:
    def __init__(self):
        self.x=10
    def m1(self):
        print('It is public method')
    def m2(self):
        print(self.x)
        self.m1()
t=Test()
t.m2()
print(t.x)
t.m1()
```

with in the class, can access member

outside of the class.

Public, Private and Protected Members

Private:

- If a member is private then we can access that member only within the class and from outside of the class we cannot access.

- We can declare a member as private explicitly by prefixing with two underscore symbols.

```
1)  class Test:
2)      def __init__(self):
3)              self.__x=10  # Private Variable
4)      def __m1(self):      # Private Method
5)              print("It is Private Method")
6)      def m2(self):
7)              print(self.__x)
8)              self.__m1()
9)  t=Test()
10) t.m2()
11) # print(t.__x)
12) # t.__m1()
```

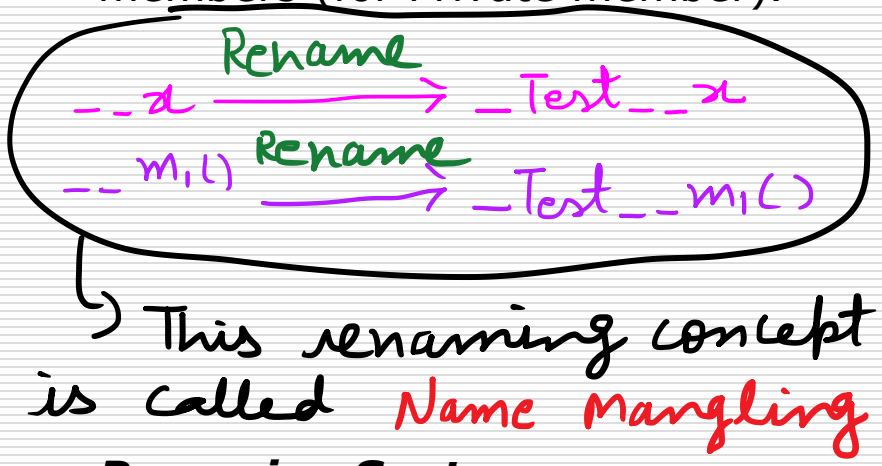
Private members but calling at inside the class, Hence valid

Private members but calling at outside the class, Hence Invalid.

Public, Private and Protected Members

Private: (Imp)

- Internally ,PVM rename all the members (for Private member).



This renaming concept is called Name Mangling.

- ***Renaming Syntax:***

_classname__variablename

- By using New name , Private members can be accessed out side of the class as well.

```
1)  class Test:
2)      def __init__(self):
3)              self.__x=10  # Private Variable
4)      def __m1(self):      # Private Method
5)              print("It is Private Method")
6)      def m2(self):
7)              print(self.__x)
8)              self.__m1()
9)  t=Test()
10) t.m2()
11) print(t. _Test__x)
12) t. _Test__m1()
```

⇒ Private members out side of the class by using new name.

Public, Private and Protected Members

Protected:

- Protected members 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 prefixing with _ symbol.

X=10 =➔ Public
__X=10=➔ Private (Double under score)
_X=10=➔ Protected (single under score)

- But it is just naming convention and it is not implemented in python, may be for the future versions purpose.

```
class Test:
    def __init__(self):
        self._x=10 #protected variable
    def m1(self):
        print(self._x)

class SubTest(Test):
    def m2(self):
        print(self._x)

t=SubTest()
t.m1()
t.m2()
print(t._x)
```

with in the class

In child class but out side the class.

out side the class. (yet to implement at language level)

Data Hiding

- Our internal data should not go out directly. i.e. outside person should not access our internal data directly.
- This OOP feature is nothing but data hiding. Main advantage is ***Security***.
- By declaring data members as private, we can implement Data Hiding.

Abstraction

- Hiding internal implementation and just highlight the set of services is the concept of Abstraction.

Eg:

Through Bank ATM GUI Screen, Bank people are highlighting the set of services what they are offering, without highlighting internal implementation. This is nothing but Abstraction.

Abstraction

- **How to implement Abstraction :**

  By Using GUI Screens, APIs etc we can implement abstraction.

- **Advantages:**

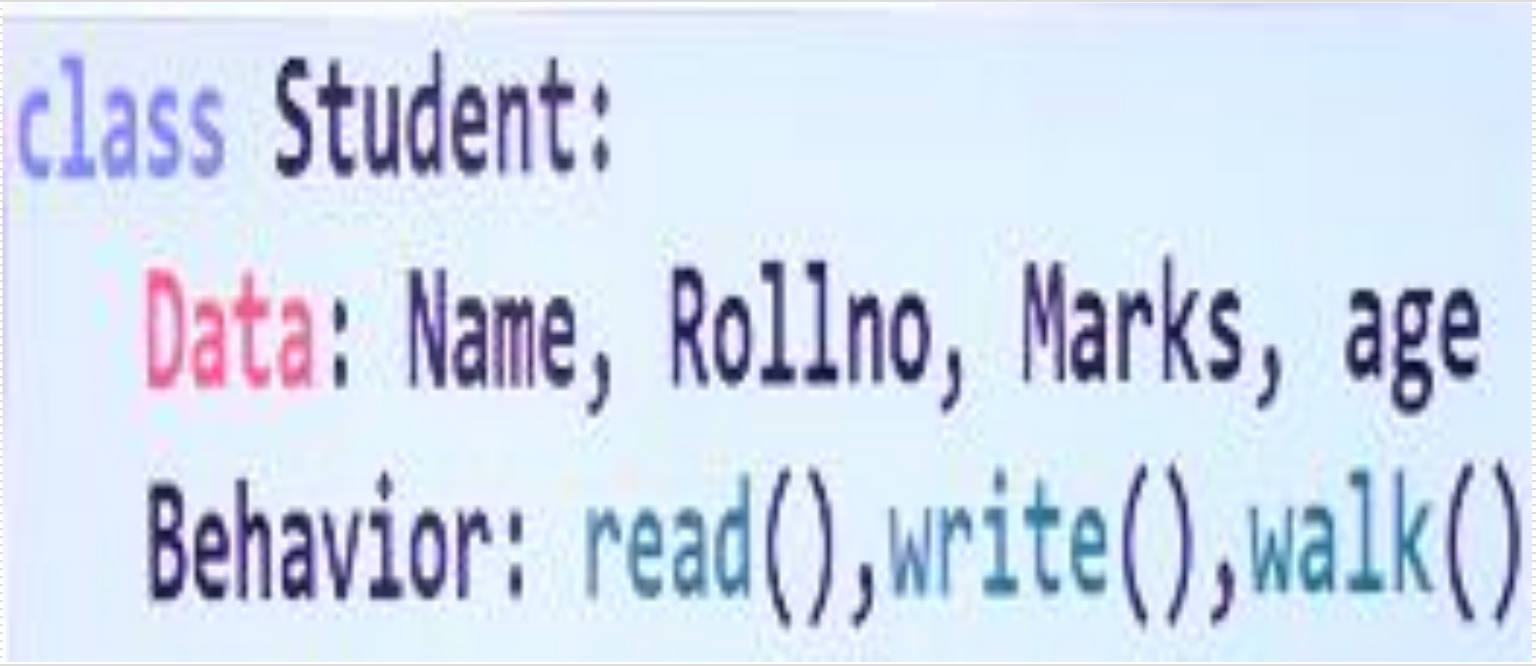
  Security
  Enhancement will become very easy.
  Maintainability and Modularity of the application.

Encapsulation:

- ***Medical Capsule:***

Encapsulation:

- ***Programming Capsule:***



Encapsulation:

- The process of Binding/ Grouping / encapsulating data(Variables) and corresponding behavior(methods) into a single unit is nothing but encapsulation.
- Every python class is an example of encapsulation.
- If any component follows data hiding and abstraction, such component is said to be encapsulated component.
- Encapsulation= Data Hiding + Abstraction

Encapsulation:

```
class Account:
    def __init__(self,initital_balance):
        self.__balance=initital_balance
    def getBalance(self):
        #Validations|Authentication
        return self.__balance
    def deposit(self,amount):
        #Validations|Authentication
        self.__balance=self.__balance+amount
    def withdraw(self,amount):
        #Validations|Authentication
        self.__balance=self.__balance-amount
```

Encapsulation

- Hiding data behind methods is the central concept of encapsulation.
- **Advantages:**

  Security
  Enhancement will become very easy.
  Maintainability and Modularity will be improved.

Encapsulation

- The main advantage of encapsulation is Security.
- The main limitation of encapsulation is that, it increases length of the code and slows down the execution. i.e. compromise with performance.
- If we want Security , we should compromise with Performance.
- If we want Performance , we should compromise with Security.