

# OOP's

## What is Class:

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

## How to define a Class?

We can define a class by using class keyword.

### Syntax:

class className:

''' documenttation string '''

variables:instance variables,static and local variables

methods: instance methods,static methods,class methods

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

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

### Example:

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

Within the Python class we can represent data by using variables.

There are 3 types of variables are allowed.

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

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

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

### Example for Class:

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

## What is Object:

Physical 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 Create an object to it. Call the method talk() to display student details

```
1) class Student:
2)
3)     def __init__(self,name,rollno,marks):
4)         self.name=name
5)         self.rollno=rollno
6)         self.marks=marks
7)
```

```

8) def talk(self):
9)     print("Hello My Name is:",self.name)
10)    print("My Rollno is:",self.rollno)
11)    print("My Marks are:",self.marks)
12)
13) s1=Student("Ravi",101,80)
14) s1.talk()

```

### Output:

```

D:\pyclasses>py test.py
Hello My Name is: Ravi
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 executed only once.
- ☞ Constructor can take atleast one argument(atleast self)
- ☞ Constructor is optional and if we are not providing any constructor then python will provide default constructor.

### Example:

```

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

```

### Program to demonstrate Constructor will execute only once per Object:

```
1) class Test:
2)
3)     def __init__(self):
4)         print("Constructor exeuction...")
5)
6)     def m1(self):
7)         print("Method execution...")
8)
9) t1=Test()
10) t2=Test()
11) t3=Test()
12) t1.m1()
```

#### Output

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

### Program:

```
1) class Student:
2)
3)     """ This is student class with required data"""
4)     def __init__(self,x,y,z):
5)         self.name=x
6)         self.rollno=y
7)         self.marks=z
8)
9)     def display(self):
10)        print("Student Name:{}\nRollno:{} \nMarks:{}".format(self.name,self.rollno,self
        .marks))
11)
12) s1=Student("Ravi",101,80)
13) s1.display()
14) s2=Student("Sunny",102,100)
15) s2.display()
```

#### Output

Student Name:Ravi  
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 <code>__init__</code>
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.

```
1) class Employee:
2)
3)     def __init__(self):
4)         self.eno=100
5)         self.ename='Ravi'
```

```
6)     self.esal=10000
7)
8) e=Employee()
9) print(e.__dict__)
```

Output: {'eno': 100, 'ename': 'Ravi', '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 that method.

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

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

## 3) Outside of the Class by using Object Reference Variable:

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

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

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

## How to Access Instance Variables:

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

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

```
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}
```

## 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.

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

### Output

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

## Various Places to declare Static Variables:

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

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

## How to access Static Variables:

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

```

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()

```

### 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

```

```
*****
```

## 1) 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.

```
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

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

## 1) Instance Methods:

- ⊗ Inside method implementation if we are using instance variables then such type of methods are called instance methods.
- ⊗ Inside instance method declaration, we have to pass self variable. `def m1(self):`
- ⊗ By using self variable inside method we can able to access instance variables.
- ⊗ Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```

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

```

#### Ouput:

```

D:\python_classes>py
test.pyEnter number of
students:2
Enter Name:Raghu
Enter Marks:90
Hi Raghu
Your Marks are: 90
You got First Grade

```

```

EnterName:Ravi
Enter Marks:12
Hi Ravi
Your Marks are: 12
You are Failed

```

## 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.

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

### Output

```
D:\python_classes>pytest.py
Dog walks with 4 IEgs...
Cat walks with 4 IEgs...
```

## Static Methods:

- ⊗ In general these methods are general utility methods.
- ⊗ Inside these methods we won't use any instance or class variables.
- ⊗ Here we won't provide self or cls arguments at the time of declaration.
- ⊗ We can declare static method explicitly by using @staticmethod decorator
- ⊗ We can access static methods by using classname or object reference

```
1) class abcMath:
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) abcMath.add(10,20)
16) abcMath.product(10,20)
17) abcMath.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.

# 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:**

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

### **Output**

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

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

- 1) o = Outer()  
    i = o.Inner()  
    i.m1()
- 2) i = Outer().Inner()  
    i.m1()
- 3) Outer().Inner().m1()

### **Demo Program-2:**

```
1) class Person:
2)     def __init__(self):
3)         self.name='ravi'
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: ravi

Dob=10/5/1947

### Demo Program-3:

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

```
1) class Human:
2)
3)     def __init__(self):
4)         self.name = 'Sunny'
5)         self.head = self.Head()
6)         self.brain = self.Brain()
7)     def display(self):
8)         print("Hello..",self.name)
9)
10)    class Head:
11)        def talk(self):
12)            print('Talking...')
13)
14)    class Brain:
15)        def think(self):
16)            print('Thinking...')
17)
18) h=Human()
19) h.display()
20) h.head.talk()
21) h.brain.think()
```

### Output

Hello.. Sunny

Talking...

Thinking...

## Types of Inheritance:

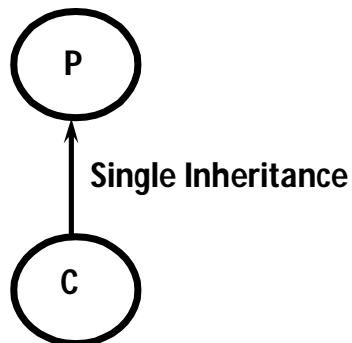
### 1) Single Inheritance:

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

```
1) class P:  
2)     def m1(self):  
3)         print("Parent Method")  
4) class C(P):  
5)     def m2(self):  
6)         print("Child Method")  
7) c=C()  
8) c.m1()  
9) c.m2()
```

#### Output:

Parent Method  
Child Method



### 2) Multi Level Inheritance:

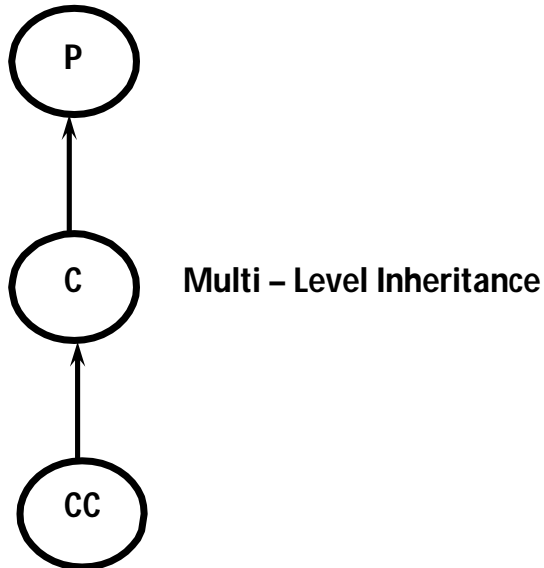
The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance.

```
1) class P:  
2)     def m1(self):  
3)         print("Parent Method")  
4) class C(P):  
5)     def m2(self):  
6)         print("Child Method")  
7) class CC(C):  
8)     def m3(self):  
9)         print("Sub Child Method")  
10) c=CC()
```

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

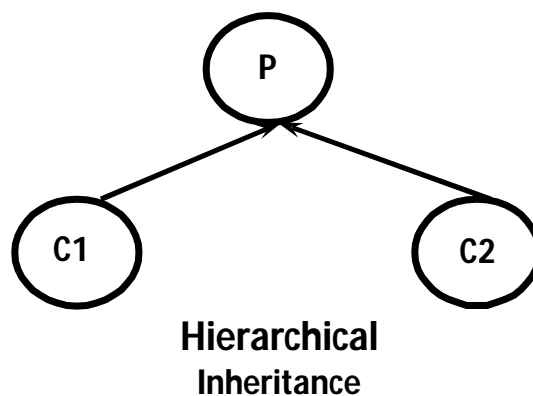
**Output:**

Parent Method  
Child Method  
Sub Child Method



### 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



```
1) class P:
2)     def m1(self):
3)         print("Parent Method")
4) class C1(P):
```

```

5) def m2(self):
6)     print("Child1 Method")
7) class C2(P):
8)     def m3(self):
9)         print("Child2 Method")
10) c1=C1()
11) c1.m1()
12) c1.m2()
13) c2=C2()
14) c2.m1()
15) c2.m3()

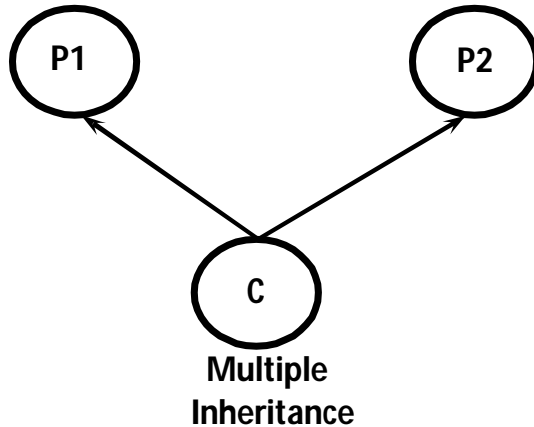
```

#### 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.



```

1) class P1:
2)     def m1(self):
3)         print("Parent1 Method")
4) class P2:
5)     def m2(self):
6)         print("Parent2 Method")
7) class C(P1,P2):
8)     def m3(self):
9)         print("Child2 Method")

```

```
10) c=C()
11) c.m1()
12) c.m2()
13) 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

```
1) class P1:
2)     def m1(self):
3)         print("Parent1 Method")
4) class P2:
5)     def m1(self):
6)         print("Parent2 Method")
7) class C(P1,P2):
8)     def m2(self):
9)         print("Child Method")
10) c=C()
11) c.m1()
12) c.m2()
```

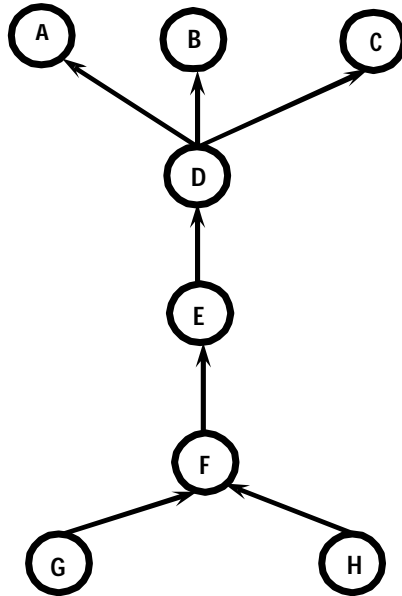
**Output:**

Parent1 Method

Child Method

## 5) Hybrid Inheritance:

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



# 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

## 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('coding'+ 'brain')#codingbrain
```

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

```
print(10*20)#200  
print('code'*3)#codecodecode
```

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('coding'+ 'brain')#codingbrain
```

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

```
print(10*20)#200  
print('code'*3)#codecodecode
```

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

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

## 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:

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

Output: two-arg method

## 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()
```

## 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)
5)     def __init__(self,a):
6)         print('One-Arg constructor')
7)
8)     def __init__(self,a,b):
9)         print('Two-Arg constructor')
10) #t1=Test()
11) #t1=Test(10)
12) t1=Test(10,20)
```

Output: Two-Arg constructor

## Constructor with Default Arguments:

```
1) class Test:  
2)     def __init__(self,a=None,b=None,c=None):  
3)         print('Constructor with 0|1|2|3 number of arguments')  
4)  
5) t1=Test()  
6) t2=Test(10)  
7) t3=Test(10,20)  
8) 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

## 3) Overriding

### Method 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.

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

```
1) class P:
2)     def college(self):
3)         print('abc college')
4)     def edu(self):
5)         print('Bsc')
6) class C(P):
7)     def edu(self):
8)         super().edu()
9)         print('Btech')
10)
11) c=C()
12) c.college()
13) c.edu()
```

### Demo Program for Constructor Overriding:

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

**Output:** Child Constructor

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

## 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

```
1) class Test:  
2)     @abstractmethod  
3)     def m1(self):  
4)         pass
```

NameError: name 'abstractmethod' is not defined

Eg:

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

## 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:

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

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

### Case-2:

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

## Interfaces In Python:

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

### test.py

```
1) from abc import *
2) class Printer(ABC):
3)     @abstractmethod
4)     def printit(self, text): pass
5)
6)     @abstractmethod
7)     def disconnect(self): pass
8)
9) class EPSON(Printer):
10)     def printit(self, text):
11)         print("Printing from EPSON Printer...")
12)         print(text)
```

```

13) def disconnect(self):
14)     print('Printing completed on EPSON Printer...')
15)
16) class HP(Printer):
17)     def printit(self,text):
18)         print('Printing from HP Printer...')
19)         print(text)
20)     def disconnect(self):
21)         print('Printing completed on HP Printer...')
22)
23) with open('config.txt','r') as f:
24)     pname=f.readline()
25)
26) classname=globals()[pname]
27) x=classname()
28) x.printit('This data has to print...')
29) x.disconnect()

```

### **Output:**

Printing from EPSON Printer...

This data has to print...

Printing completed on EPSON Printer...

## **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 = 'ravi'

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 prefixing with `_` symbol.

**Syntax:** `_variablename = value`

**Eg:** `_name='ravi'`

But it is just convention and in reality does not exist 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 prefixing with 2 underscore symbols.

**syntax:** `__variablename=value`

**Eg:** `.name='ravi'`

```
1) class Test:
2)     x=10
3)     _y=20
4)     z=30
5)     def m1(self):
6)         print(Test.x)
7)         print(Test._y)
8)         print(Test._z)
9)
10) t=Test()
11) t.m1()
12) print(Test.x)
13) print(Test._y)
14) 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`

```
1) class Test:
2)     def __init__(self):
3)         self._x=10
4)
5) t=Test()
6) print(t._Test_x)#10
```