In [ ]:

```
# class ,object and method creation
```

In [1]:

```
class MyClass:
    def myfunc(self):
        pass
    def display(self,name):
        print('Name is :',name)
na=MyClass()  #object na
na.myfunc()
na.display('satish')
```

```
Name is : satish
```

In [ ]:

```
# instance mathod and static method difference
```

In [3]:

```
class Vehicle:
    def car(self):
        print("car has 4 wheels ")
    @staticmethod
    def cycle(self):
        print("cycle has",self,"wheels")
veh=Vehicle()
veh.car()
Vehicle().cycle(2)
veh.cycle(2)
```

```
car has 4 wheels
cycle has 2 wheels
cycle has 2 wheels
```

In [ ]:

```
# Declaring variables inside the class
```

In [6]:

```
class Age:
    a,b=20,18    # class variables
    def add(self):  #instance method
        print(self.a+self.b)
    def mp(self):
        print(self.a*self.b)
ab=Age()
ab.add()
ab.mp()
```

```
38
360
```

In [ ]:

```
#local variables (variables within the method)
#class variables (variables inside the class outside the method , we can acces by using sel
#global variables (variables which are outside the class)
```

In [7]:

```
i,j=10,11 #global variables
class Var:
    a,b=12,13 #class variables
    def add(self,x,y):  #x,y are local variables
        #variables accessing
        print(x+y)
        print(self.a+self.b)
        print(i+j)
va=Var()
va.add(5,7)
```

```
12
25
21
```

In [2]:

```
# variables with same name
a,b=10,11 #global variables
class Var:
    a,b=12,13 #class variables
    def add(self,a,b):  #x,y are local variables
        #variables accessing
        print(a+b)
        print(self.a+self.b)
        print(globals()['a']+globals()['b'])
va=Var() #named object
va.add(5,7)
#Var().add(5,7) #nameless object
```

```
12
25
21
```

In [19]:

```
#checking memory locations of the objects
class MyClass:
    def m1(self):
        pass
a=MyClass()
b=MyClass()
c=MyClass()
print(id(a),id(b),id(c))
print(a is b) # checking locations
print(b is not c)
```

```
1016607497088 1016607495792 1016607494976
False
True
```

In [40]:

```
#method  contains logic and constructor contains initialisation
# constructer will be invoked when object is created
class MyClass:
    def display(self,name):
        print('Name is :',name)
    def __init__(self):
        print("---------")
na=MyClass()  #object na
na.display('satish')
```

```
---------
Name is : satish
```

In [12]:

```
# converting local variables into global variables
class Age:
    def __init__(self,a,b):
        print(a+b)
        self.c=a   # c,d are now class variables
        self.d=b
    def mul(self):
        print(self.c*self.d)
ob=Age(2,3)   #constructor arguments
ob.mul()
```

```
5
6
```

In [20]:

```
#calling current class method in another method
class Gend:
    def m1(self):
        print("method1")
        self.m2(100)
    def m2(self,a):
        print("method2",a)
ge=Gend()
ge.m1()
```

```
method1
method2 100
```

In [17]:

```
# consturctor arguments
class A:
    a="reddi"
    def __init__(self,a):
        print(a)
        print(self.a)
c=A("satish")
```

```
satish
reddi
```

In [30]:

```
#problem
class Emp:
    def __init__(self,eid,sal):
        self.eid=eid
        self.sal=sal
    def display(self):
        print("Employee ID : {} \t Employee Salary : {}".format(self.eid,self.sal))
        print("Employee ID : %d \t Employee Salary : %d" %(self.eid,self.sal))
em=Emp(12,2222)
em.display()
```

```
Employee ID : 12         Employee Salary : 2222
Employee ID : 12         Employee Salary : 2222
```

In [ ]:

```
# __str__ executes automatically when you print reference variable
# __del__ destroy the object
```

In [31]:

```
#__str__
class Age:
    pass
c=Age()
print(c)  #printinf reference variable
```

```
<__main__.Age object at 0x0000009AD3E29A00>
```

In [27]:

```
class Age:
    def __str__(self):  #str constructor returns only string not number.
        return "Welcome"
c=Age()
print(c)  #overrided here
```

```
Welcome
```

In [28]:

```
#problem
class Emp:
    def __init__(self,eid,sal):
        self.eid=eid
        self.sal=sal
    def __str__(self):
        return("Employee ID : {} \t Employee Salary : {}".format(self.eid,self.sal))
em=Emp(12,2222)
print(em)
```

```
Employee ID : 12         Employee Salary : 2222
```

In [7]:

```
#inheritance (single,multiple,hierarchical,multilevel,hybrid)
class A:
    def m1(self):
        print("This is method m1 from class A")
class B(A):
    def m2(self):
        print("This is method m2 from class B")
aobj=A()
aobj.m1()

bobj=B()
bobj.m1()
bobj.m2()
```

```
This is method m1 from class A
This is method m1 from class A
This is method m2 from class B
```

In [15]:

```
#single inheritance ( child class inherites properities from parent class)
class A:
    x,y=10,20
    def m1(self):
        print(self.x+self.y)
class B(A):
    a,b=100,200
    def m2(self):
        print(self.a+self.b)

bo=B()
bo.m2()
bo.m1()
```

```
300
30
```

In [16]:

```
#multilevel inheritance(A is parent class of B , B is parent class of C ,C is parent class
class A:
    x,y=10,20
    def m1(self):
        print(self.x+self.y)
class B(A):
    a,b=100,200
    def m2(self):
        print(self.a+self.b)
class C(B):
    i,j=300,500
    def m3(self):
        print(self.i+self.j)

bo=B()
bo.m2()
bo.m1()
co=C()
co.m1()
co.m2()
co.m3()
```

```
300
30
30
300
800
```

In [19]:

```
#hiearchical inheritance(one parent having multiple childs)
class A:
    x,y=10,20
    def m1(self):
        print(self.x+self.y)
class B(A):
    a,b=100,200
    def m2(self):
        print(self.a+self.b)
class C(A):
    i,j=300,500
    def m3(self):
        print(self.i+self.j)

bo=B()
bo.m1()
bo.m2()
co=C()
co.m1()
co.m3()
```

```
30
300
30
800
```

In [20]:

```
#multiple inheritance(one child class have multiple parent class properites)
class A:
    x,y=10,20
    def m1(self):
        print(self.x+self.y)
class B:
    a,b=100,200
    def m2(self):
        print(self.a+self.b)
class C(A,B):
    i,j=300,500
    def m3(self):
        print(self.i+self.j)

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

```
30
300
800
```

In [17]:

```
#super() keyword in inheritance
#To invoke Parent class methods,varaibles and constructers
class A:
    a,b=10,20
    def m1(self):
        print("This is method m1 from A")
        b=5
class B(A):
    def m2(self):
        print("This is method m2 from B")
        super().m1()  #invoke parent class method
        print(self.a+self.b) # parent class variables
b=B()
b.m2()
```

```
This is method m2 from B
This is method m1 from A
30
```

In [2]:

```
a,b=1,2
class A:
    a,b=10,20
    def m1(self):
        print("This is method m1 from A")
class B(A):
    a,b=1000,2000
    def m2(self,a,b):
        print(a+b)   #local varibles
        print(self.a+self.b) # class variables
        print(super().a+super().b) #invoke parent class variables
        print(globals()['a']+globals()['b'])
bobj=B()
bobj.m2(100,200)
```

```
300
3000
30
3
```

In [3]:

```
#invoking constructer by using super() keyword
class A:
    def __init__(self):
        print("class A")
class B(A):
    pass
bobj=B()
```

```
class A
```

In [4]:

```
#invoking constructer by using super() keyword
class A:
    def __init__(self):
        print("class A")
class B(A):
    def __init__(self):
        print("class B")
bobj=B()
```

```
class B
```

In [8]:

```
#invoking constructer by using super() keyword
class A:
    def __init__(self):
        print("class A")
class B(A):
    def __init__(self):
        super().__init__()  #1. parent class constructor invoked
        print("class B")
        A.__init__(self)  #2. parent class constructor invoked
bobj=B()
```

```
class A
class B
class A
```

In [9]:

```
#polymorphism (samething can behave in multiple ways)
#achieved by overriding and overloading
# overriding : having 2 methods but doing different tasks
# overriding can used for both methods and variables.
```

In [19]:

```
#overriding variables example
class Parent:
    name="reddi"
class Child(Parent):
    name="satish"
obj=Child()
print(obj.name)
```

```
satish
```

In [20]:

```
#overriding methods
class Bank:
    def RateOfInterest(self):
        return 0
class ICICI(Bank):
    def RateOfInterest(self):
        return 10.5
obj=ICICI()
print(obj.RateOfInterest())

obj1=Bank()
obj1.RateOfInterest()
```

```
10.5
```

Out[20]:

```
0
```

In [ ]:

```
#overloading (calling method in multiple ways)
#giving a single method , we can specify the no of parameters ourselves.
```

In [18]:

```
#overloading methods example
class Human:
    def SayHello(self,name=None):
        if name is not None:
            print("Hello",name)
        else:
            print("Hello")
obj=Human()
obj.SayHello("Satish")
```

```
Hello Satish
```

In [ ]:

```
#Encapsulation (Restricting the access to the methods and variables)
#we can achieved encapsulation by private variables and private methods.
```

In [19]:

```
#private variables
class MyClass:
    __a=10   #private variable
    def display(self):
        print(self.__a)
obj=MyClass()
obj.display()
```

```
10
```

In [25]:

```
#private methods
class A:
    def __d1(self):
        print("this is private method")
    def d2(self):
        print("this is public method")
        self.__d1()  #invoking private method
obj=A()
obj.d2()
```

```
this is public method
this is private method
```

In [1]:

```
#accessing private variables indirectly
class A:
    __a=100
    def dis(self,b):
        c=self.__a
        self.__a=b
        print(b)
        print(c)
    def dis1(self):
        print(self.__a)
obj=A()
obj.dis(105)
obj.dis1()
```

```
105
100
105
```

In [ ]:

```
#abstraction
#an abstract method only contains definition but doesn't contains implementation and cannot
```

In [5]:

```
#ABC is name of the predefined abstract class
from abc import ABC,abstractmethod
class A(ABC):     #abstract class
    @abstractmethod
    def display(self):     #abstract method
        None
class B(A):
    def display(self):
        print("it is B class")
obj=B()
obj.display()
```

```
it is B class
```

In [22]:

```
#example
from abc import ABC,abstractmethod
class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass
class Tiger(Animal):
    def eat(self):
        print("Eat Non-Veg")
class Cow(Animal):
    def eat(self):
        print("Eat Veg")
obj1=Tiger()
obj2=Cow()
obj1.eat()
obj2.eat()
```

```
Eat Non-Veg
Eat Veg
```

In [6]:

```
from abc import ABC,abstractmethod
class X(ABC):        #abstract class
    @abstractmethod
    def m1(self):
        pass

    @abstractmethod
    def m2(self):
        pass
class Y(X):
    def m1(self):
        print("this is m1")
class Z(Y):
    def m2(self):
        print("this is m2")
obj=Z()
obj.m2()
obj.m1()
```

```
this is m2
this is m1
```

In [10]:

```
from abc import ABC,abstractmethod
class Cal(ABC):
    def __init__(self,value):
        self.value=value   #class variable
    @abstractmethod
    def add(self):
        pass
    @abstractmethod
    def sub(self):
        pass

class C(Cal):
    def add(self):
        print(self.value+100)
    def sub(self):
        print(self.value-10)
obj=C(100)
obj.add()
obj.sub()
```

```
200
90
```

In [ ]:

```

```

In [7]:

```
class Solution():
    def add(self,l1,l2):
        self.l1=l1
        self.l2=l2
        l1=l1[::-1]
        l2=l2[::-1]
        a,b=0,0
        for i in range(len(l1)):
            x=l1[i]
            y=l2[i]
            a+=(10**i*x)
            b+=(10**i*y)
        res=a+b
        rem=0
        rl=[]
        for i in range(len(str(res))):
            rem=res%10
            res=res//10
            rl.insert(0,rem)
        print(rl[::-1])
obj=Solution()
obj.add([6,4,3],[5,6,4])
```

```
[7, 0, 2, 1]
```

In [6]:

```
class Solution():
    def findMedianSortedArrays(self,nums1,nums2):
        self.nums1=nums1
        self.nums2=nums2
        nums1+=nums2
        nums1.sort()
        l=len(nums1)
        median=0
        if(l%2==0):
            value=nums1[l//2]+nums1[(l//2)-1]
            median =(value/2)
            return median
        else:
            median=nums1[l//2]
            return median
obj=Solution()bj.findMedianSortedArrays([1,2],[3,4])
```

Out[6]:

```
2.5
```

In [ ]: