In [3]:
```
#OOPS concepts

#Inheritance

class A:
    def m1(self):
        print("class A-Method1")

class B(A):
    def m2(self):
        print("class-B-Method2")
aobj=A()
aobj.m1()
bobj=B()
bobj.m2()
```

```
class A-Method1
class-B-Method2
```

In [6]:
```
#Single Inheritance
class Demo:
    x,y=10,20
    def add(self):
        print(self.x+self.y)
class demo1(Demo):
    a,b=1,2
    def add2(self):
        print(self.a+self.b)
d2=demo1()
d2.add()
d2.add2()
```

```
30
3
```

In [8]:
```
#multilevel inheritance
class Demo:
    x,y=10,20
    def add(self):
        print(self.x+self.y)
class demo1(Demo):
    a,b=1,2
    def add2(self):
        print(self.a+self.b)
class demo2(demo1):
    i="amulya"
    j="arshanapelli"
    def cont(self):
        print(self.i+self.j)
d2=demo2()
d2.add()
d2.add2()
d2.cont()
```

```
30
3
amulyaarshanapelli
```

In [12]:
```
#Hierarchical inheritance
class A:
    x,y=10,20
    def add1(self):
        print(self.x+self.y)
class B(A):
    a,b=2,4
    def add2(self):
        print(self.a*self.b)
class C(A):
    i,j="amulya","arshanapelli"
    def add3(self):
        print(self.i+self.j)
b=B()
b.add1()
b.add2()

c=C()
c.add1()
c.add3()
```

```
30
8
30
amulyaarshanapelli
```

In [15]:
```
#Multiple inheritance
class A:
    x,y=10,20
    def add1(self):
        print(self.x+self.y)
class B():
    a,b=2,4
    def add2(self):
        print(self.a*self.b)
class C(A,B):
    i,j="amulya","arshanapelli"
    def add3(self):
        print(self.i+self.j)
c=C()
c.add1()
c.add2()
c.add3()
```

```
30
8
amulyaarshanapelli
```

In [19]:
```
#invoke parent class methods
#calling parent class methodsusing super keyword
class A:
    def m1(self):
        print("this is method from A class")
    def m2(self):
        print("this is the second method in A class")
class B(A):
    def m2(self):
        super().m1()
        super().m2()
        print("this is method from B class")

b=B()
b.m2()
```

```
this is method from A class
this is the second method in A class
this is method from B class
```

In [20]:
```
#accessing parent class variables to the child class
class A:
    i,j=10,20
class B(A):
    a,b=100,200
    def m1(self,x,y):
        print(x+y)
        print(self.a+self.b)
        print(self.i+self.j)
b=B()
b.m1(1,2)
```

```
3
300
30
```

In [26]:
```
#invoke parent class variables
#accessing parent class variables to the child class
x,y=2,4
class A:
    x,y=10,20
class B(A):
    x,y=100,200
    def m1(self,x,y):
        print(x+y)   #local variables
        print(self.x+self.y)  #class variables
        print(super().x+super().y) #parent class variables
        print(globals()['x']+globals()['y']) #global variables
b=B()
b.m1(1,2)
```

```
3
300
30
6
```

In [30]:
```
class A:
    def __init__(self):
        print("constructor of class A")
class B(A):
    pass
b=B()
```

```
constructor of class A
```

In [37]:
```
#invoke parent class constructor using super keyword
class A:
    def __init__(self):
        print("constructor of class A")
class B:
    def __init__(self):
        print("Constructor of Class B")
class C(A,B):
     def __init__(self):
        print("constructor of class C")
        super().__init__()
        B.__init__(self)

c=C()
```

```
constructor of class C
constructor of class A
Constructor of Class B
```

In [40]:
```
#Polymorphism
#overriding a variable
class parent:
    name="amulya"
class child(parent):
    name="radha"
    #pass
obj=child()
print(obj.name)
```

```
radha
```

In [46]:
```
#overriding a method
class model:
    def add(self):
        return "sg"
class prize(model):
    def add(self):
        return 1000
p=prize()
print(p.add())
```

```
1000
```

In [51]:
```
#overloading a method
class A:
    def m1(self,name=None):
        if name is not None:
            print("hello" +  name)
        else:
            print("not available")
a=A
a.m1(1,"amulya")
```

```
helloamulya
```

In [64]:
```
#overloading methods
class cars:
    def brand(selfself,name=None):
        if name=="Toyota":
            print("Name of the car:"+name)
        if name=="Suzuki":
            print("Name of the car:"+name)
        if name=="Kreta":
            print("Name of the car:"+name)
        if name is None:
            print("No cars available"+name)
c1=cars()
c1.brand("Kreta")
```

```
Name of the car:Kreta
```

In [73]:
```
#Encapsulation
#accessing private variables within the method
class A:
    __a=10  #private variable
    def disp(self):
        print(self.__a)  #accessing private variable
a=A()
a.disp()
#print(A.__a)  #cannot access private variable outside the class
```

```
10
```

In [76]:
```
class A:
    def __m1(self):
        print("hello world")
    def m2(self):
        print("welcome python")
        self.__m1()
a=A()
a.m2()
#a.__m1()   #cannot access private method from out side the class
```

```
welcome python
hello world
```

In [103]:
```
class emp:
    __eid=101
    def m1(self,eid):
        self.__eid=eid
    def disp(self):
        print(self.__eid)
e1=emp()
e1.m1(104) #modify private variables outside the class
e1.disp()
```

```
104
```

In [107]:
```
#Abstraction
from abc import ABC,abstractmethod
class A(ABC):    #ABC is predefined abstract class present in python
    @abstractmethod
    def disp(self):
        None
class B:
    def disp(self):
        print("abstract class implemented")
b1=B()
b1.disp()
```

```
abstract class implemented
```

In [112]:
```
from abc import ABC,abstractmethod
class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass
class Birds(Animal):
    def eat(self):
        print("Animals eat birds only")
class Insects(Animal):
    def eat(self):
        print("Animals eats birds and insects")
i=Insects()
i.eat()
b=Birds()
b.eat()
```

```
Animals eats birds and insects
Animals eat birds only
```

In [114]:
```
#implement abstract methods
from abc import ABC,abstractmethod
class M(ABC):
    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
class N(M):
    def m1(self):
        print("m1 is implemented")
class O(N):
      def m2(self):
        print("m2 is implemented")


o1=O()
o1.m1()
o1.m2()
```

```
m1 is implemented
m2 is implemented
```

In [116]:
```
#implement constrctor in abstract class
from abc import ABC,abstractmethod
class A(ABC):
    def __init__(self,value):
        self.value=value
    @abstractmethod
    def add(self):
        pass
    @abstractmethod
    def sub(self):
        pass
class B(A):
    def add(self):
        print(self.value+100)
    def sub(self):
        print(self.value-100)
b1=B(200)
b1.add()
b1.sub()
```

```
300
100
```

In [ ]:
```

```