OOPS (python)

Class & Object

- class is logical entity and blue print of object.
- class contain variables and methods.
- logic should be implemented in a method
- inside the class called "method".
- outside the class called "function".

- procedural means using "functions" and "data".
- oops means using "class "and "object"
- function is block of code, when we need we can call that function.
- Python support both function & method we can create both.
- Function created with in the class called “method”.
- Function created outside the class called “Function”.
- Every method in the class by default take self-key word.

---

creating class and object

Ex:

```
class take ():              # class name
   def clss(self):        # method inside the class, # self is a default keyword
      pass
   def display (self, name):        # method inside the cls
      print("name is: ",name)
c=take()          # creating object to the class
c.clss()
c.display("manga")
```

---

instance method and static method

- 2 types of method are **instance** and **static methods.**
- when u create a method inside the class by default it is instance method.
- instance method will be calling by using object.
- instance method self is must and should present.
- static method can call directly with class name and method name.
- by default, static method will not take any parameter (not take self also).
- **if we pass self in static method, we should pass variable when calling it. IMP**

for static method mention -----> @staticmethod is "imp"

---

ex:

instance method and static method

```
class revenge ():
   def m1(self):
      print ("instance methos")       # instance methos
   @staticmethod
   def m2():
      print ("static method")      # static method
take=revenge ()
take.m1()
revange.m2()
```

if we pass self in static method, we should pass variable when   calling it.

ex:

```
class r():
   @staticmethod
   def revenge(self):
      print ("static method")
r.revenge(10)
```

---

declaring the variables inside the class.

- ✓ class variables are accessed using "self" keyword.

```
class revenge():
   a,b=10,20         # class variables
   def add(self):
      print(self.a + self.b)      # self using to access the class variables
   def mul(self):
      print(self.a * self.b)       # self using to access the class variables
R=revenge()
R.add()
R.mul()
```

Local variables, class variables, Global variables

```
x=5      # x,y are global variables
y=10
class revange:
   a,b=10,20               # a,b are class variables
   def add(self,i,j):                    # local variables
      print(i+j)     # 40                 # accessing local variables
      print(self.a + self.b)  #30   # accessing class variables
      print(x+y)   #15                # globales can print directly bec different variables
R=revange()
R.add(20,20)
```

Local variables, class variables, Global variables having same variable names

---

```
x=5                  # x,y are global variables
```

```
y=10
class revange:
   x,y=10,20                   # X, Y are class variables
   def add(self,x,y):                  # local variables
      print(x+y)     # 40              # accessing local variables
      print(self.x + self.y) #30                # accessing class variables
      print(globals()['x']+globals()['y'])   #15                 # globales can print directly bec different variables
R=revange()
R.add(20,20)
```

----

creating multiple objects to the single class.

```
class revange:
   def add(self,a,b):
      print("a+b is: ",a+b)
   def sub(self,a,b):
      print("a-b is: ",a-b)
   def mul(self,a,b):
      print("a*b is: ",a*b)

R=revange()
R.add(5,5)

r1=revange()
r1.sub(10,5)

r2=revange()
r2.mul(5,5)
```

Named Object & Name Less Object

```
class revenge:
   def add(self,a,b):
      print("a+b is: ",a+b)

R=revenge()
R.add(10,20)

revenge.add(5,5,5)  # name less object # for nameless object we hv to pass something to "self".
```

```
class revenge:
   def add(self):
      print("take revange:")

R=revenge()
```

```
R.add()

revenge.add(1)  # # name less object # for nameless object we hv to pass something to "self".
```

---

location of memory address

```
class revange:
    def m1(self):
        print("taken:")
obj=revange()
obj1=revange()

obj.m1()
obj1.m1()

print(id(obj))      # 2714821881744
print(id(obj1))     # 2714825051664
obj3=obj
print(id(obj3))     # 2714821881744
```

C, O & Constructor. Part 2

---

constructor is created using __init__ keyword.

- normally constructor is used to initialize the values.
- method contain logic, Constructor contain initialization.
- methods are called using object, constructor called automatically when object is created. no need to call constructor explicitly.

Creating constructor

Ex:

```
class revenge:  # or class revenge ():
     def m1(self):
         print("wakeup")
     def __init__(self):
         print ("this is a constructor")
c=revenge ()
c.m1()
```

O/P

1st constructor will execute then m1 method will be executed.

- constructor will call automatically when object is created.

---

converting local variable into class variable

---

ex 1:

```
class revange():
   def value(self,val1,val2):      # val1 & val2 are local variables
      print(val1)     # 10
      print(val2)     # 20
      self.val1=val1         # converting L V in to cls V
      self.val2=val2         # converting L V in to cls V
   def add(self):
      print(self.val1 + self.val2)

obj=revange()
obj.value(10,20)
obj.add()  # 30
```

Ex 2: we can use constructors also.

```
class a:
   def __init__(self,val1,val2):
      print(val1)     # 20
      print(val2)     # 10
      self.val1=val1
      self.val2=val2
   def sub(self):
      print(self.val1 - self.val2)    # 10
obj=a(20,10)
obj.sub()
```

---

how to call current class method in another method with in the same class.

```
class revange:
   def m1(self):
      print("wake up")
      self.m2("yes")      # calling m2 method in m1 by converting in to class method
   def m2(self,a):
      print("take the revange: ",a)
obj=revange()
obj.m1()
# no need to create object to the m2 method bec calling in the m1 method by converting in to class method.
```

---

constructor out with arguments

```
class A:
   a="manga1"
   def __init__(self):   # constructor with arguments
      print (“this is constructor”)                  # local variable
      print (self.a )        # calling class variable by using self-key ward
obj=A("manga2")
```

---

constructor with arguments

```
class A:
   a="manga1"
   def __init__(self,a):   # constructor with arguments
      print(a)              # local variable
      print(self.a)         # calling class variable by using self-key ward
obj=A("manga2")
```

Ex: home work

```
class emp:
   def __init__(self,eid,ename,sal):
      self.eid=eid               # converting LV in to class Variable
      self.ename=ename           # converting LV in to class Variable
      self.sal=sal               # converting LV in to class Variable

   def display(self):
      print(f"eid:{self.eid},ename:{self.ename},sal:{self.sal}")
obj=emp(1,"manga",1000)
obj.display()
```

O/P

```
eid:1,ename:manga,sal:1000
```

- pre-defined methods/functions are __str__ & __delete__.
- __str__ will execute automatically when u print the reference variable.
- __delete__ will invoke when u destroy the object.

__str__   # str will print when u print the reference variable of the object.

```
class revange:
   def A(self):
      print("taken")
obj=revange ()              # obj is the reference variable.
print(obj)                  # it will print the memory location of the object.
```

- ✓ Str will return some value that is only a STRING.
- ✓ Otherwise TypeError.
- ✓ Instead of printing we have to write **return in __str__.**
- ✓ **__str__ will automatically invoke when we print the reference variable**

Ex:

```
Class Myclass:
         Pass
```

```
Obj=Myclass()
Print(obj)
```

Ex:

```
Calss Myclass:
        Def __str__(self):
                Return "welcome"
Obj=Myclass()
Print (Obj)
```

__del__

- ✓ **__del__ invoke when u destroy the object.**

```
class A:
   def m1(self,B):
      print("destroyed: ",B)
obj=A()
obj.m1("neena")

obj1=A()
del obj1
```

Inheritance

- one class can inherit the features of another class called inheritance.
- **types of inheritance:** **single, multi-level, hierarchical, multiple, hybrid.**

parent class, super class, base class

child class, derived class, class

- **single inheritance:** **one parent class and one child class.**

```
class A:
   def m1(self,a):
      print("mrthod from m1: ",a)
class B(A):
   def m2(self,b):
      print("method from m2: ",b)
obj=A()
obj.m1("m1")      # # mrthod from m1:  m1

obj=B()
obj.m1("m1")      # mrthod from m1:  m1
obj.m2("m2")      # method from m2:  m2
```

Ex: for single inheritance with class variables

```
class A:
   x,y=10,20
   def m1(self):
      print(self.x + self.y)         # 30
```

```
class B(A):
   a,b=20,20
   def m2(self):
      print(self.a + self.b)          # 40
obj=B()
obj.m1()
obj.m2()
```

---

multi-level inheritance: **one parent class and more child class.**

```
class A:
   a,b=10,20
   def m1(self):
      print(self.a + self.b)

class B(A):
   i,j=20,20
   def m2(self):
      print(self.i + self.j)

class C(B):
   x,y=30,30
   def m3(self):
      print(self.x + self.y)

obj=C()
obj.m1()      # 30
obj.m2()      # 40
obj.m3()      # 60
```

created object for the C class we can access all the class methods bec c is extended from B and B is extended from A.

Hierarchical inheritance: **one parent class having multiple child classes.**

```
class A:
   def m1(self,a,b):
      print("a+b is: ",a+b)   # 30

class B(A):
   def m2(self,i,j):
      print("i+j is: ",i+j)   # 40

class C(A):
   def m3(self,x,y):
      print("x+y is: ",x+y)  # 30 ,30
```

```
obj=A()
obj.m1(10,20)

obj=B()
obj.m2(20,20)
obj.m1(10,20)

obj=C()
obj.m3(30,30)
obj.m1(45,45)
```

Multiple inheritance: **one child class having so many parents' classes.**

```
class A:
   def m1(self,a,b):
      print("a+b is: ",a+b)       # 30

class B:
   def m2(self,i,j):
      print("i+j is: ",i+j)       # 60

class C(A,B):
   def m3(self,x,y):
      print ("x+y is: ",x+y)       # 90

obj=C()
obj.m1(15,15)
obj.m2(30,30)
obj.m3(45,45)
```

Hybrid is the combination of multiple & hierarchical

----

inheritance part 2

super ()

- **how to invoke super () key word.**
  - `super keyword is used to invoke parent class method.`
  - `super keyword is used invoke parent class variable.`
  - `super keyword is used to invoke parent class constructor.`

Ex: super keyword is used to invoke parent class method in child class.

- **How to invoke parent class method in child class.**

```
class A:
   def m1(self):
        print ("method from A")

class B(A):
    def m2(self):
        print ("method from B")
        super().m1()                # method from A, calling parent class method
obj=B ()
obj.m2()                # method from B
```

- **super keyword is used invoke parent class variable.**

```
class A:
    a,b=2,2
    def m1(self):
        print (self.a + self.b)

class B(A):
    i,j=4,4
    def m2(self):
        print (self.i  + self.j)     # 8
        super().m1()        # 4
obj=B()
obj.m2()
```

GV, PV, CV, LV all variables’ names are same accessing the variables

```
a,b=10,10     # global variables

class A:
    a,b=20,20     # class A variables

class B(A):
    a,b=30,30     # class B variables
    def m1(self,a,b):     # local variables
        print(a+b)     # 40
        print(self.a + self.b)     # 60
        print(super().a + super().b)    # 40   using super() keyword bec variables name are same.
        print(globals()['a'] + globals()['b'])     # 20
obj=B ()
obj.m1(40,40)
```

to invoke parent class constructor.

Ex 1:

```
class A:
    def __init__(self):
        print ("this is parent class constructor")
class B(A):
   pass
b=B ()                          # this is parent class constructor,
# "no need call method for constructor bec constructor invoke automatically when object is created"
```

Ex 2:

```
class A:
   def __init__(self):
      print ("this is A constructor")

class B(A):
   def __init__(self):
      print ("this is B constructor")
      super().__init__()             # calling parent class constructor
         or
      A.__init__(self)     # calling parent class constructor this also same
b=B ()
```

o/p:

```
this is B c
this is A c
```

POLYMORPHISUM

- pymphs means one name, many forms.
- same method performs different tasks depending on the object.
- polymorphism can be achieved by overriding method & overriding variables.

over riding a variable

```
class parent:
   name="scott"
class child(parent):
   name="tiger"       # here overriding the variables
obj=child ()
obj.name                      #tiger
# latest variable which we have passed that will be printed
```

method over riding

```
class Bank:
   def rateofinterest(self):
      return 0
class IDBI(Bank):
   def rateofinterest(self):
      return 11
obj=IDBI ()
print(obj.rateofinterest())
```

```
obj1=Bank ()
print (obj1.rateofinterest())
```

Over loading method

```
class Human:
    def hello(self,name=None):
        if name is not None:
            print ("hello manga" " "+name)
        else:
            print("manga")
obj=Human ()
obj.hello("bharath")                        # hello manga bharath

obj.hello()                                  # hello manga
```

Ex:

```
class Bird:
    def fly(self,name=None):
        if name=="parrot":
            print ("can fly")
        if name=="monkey":
            print("neene",name)
        if name=="cow":
            print("god")
        if name==None:
            print ("getlost from here")
obj=Bird ()
obj.fly("parrot")      # can fly
obj.fly("cow")      # god
obj.fly("monkey",)  # neene monkey
obj.fly()       # getlost from here
```

Encapsulation

- process of wrapping up variables & methods into a single entity.

  or

- E means hiding the details of class & only allowing to access through special functions(methods).

Ex:

putting things inside a box & giving key(methods) to open it.
you directly can't touch what's inside, you must use the key provided.

- E can achieve by private variables & methods.
- P variables & P methods can access only inside the class.

Ex: private variables can access only within the class.

```
class Myclass:
    __a=10                    # __a represents Private variable.
    def display(self):
        print(self.__a)
```

```
obj=Myclass()
obj.display()    # 10
print (Myclass.__a)  # Error because outside the class we can't access private variable.
```

Ex: private method can access only within the class.

```
class Myclass ():
   def __display(self):                          # __display represent private method.
      print ("this is d1 method:")         # this is d1 method:

   def display2(self):
      print ("this is D2 methods: ")          # this is D2 methods:
      self.__display()
obj=Myclass()
obj.display2()
```

Private variable can access outside the class by using method.

```
class Myclass:
   __a=10          # PV
   def d1(self,a):
      self.__a=a
   def d2(self):
      print(self.__a)

obj=Myclass ()
obj.d1(100)
obj.d2()
```

Abstraction.

- Abstract class are the class that contain one or more abstract method.
- Abstract method can declare but no implementation.
- we can't create obj directly to the abstract class.
- if we want to access the abstract class, we have to create sub class extended abstract class, & implementation done in sub class.

Ex:

```
from abc import ABC, abstractmethod
class A(ABC):       # parent/super/base class
   @abstractmethod
   def m1(self):
      None
class B(A):     # child/sub/derived class
   def m1(self):
      print ("this is AB method: ")
obj=B ()
```

```
obj.m1()
#  implementation in the child class so same method name.
```

- ABC is a pre-defined class.
- we can create multiple class & sub class foe the abstract class.
- we can implement abstract method in multiple class also.
- along with the abstractmethod we can create constructor.
-
- **when to use abstract class.**
- **requirement clear agi edu, but implementation gotila andray go for abstract class.**

Ex:

```
from abc import ABC, abstractmethod
class animal (ABC):
    @abstractmethod
    def eat(self):
        pass
class tiger(animal):
    def eat(self):
        print ("tiger eat non veg:  ")
class cow(animal):
    def eat(self):
        print ("cow eat veg:  ")
obj=tiger ()
obj.eat()

obj=cow ()
obj.eat()
```

Ex: multiple sub classes

```
from abc import ABC, abstractmethod
class A(ABC):
    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass

class B(A):
    def m1(self):
        print ("this is m1 in A: ")

    def m2(self):
        print ("this is m2 in A: ")
obj=B ()
obj.m1()
obj.m2()
        #  or

class C(B):
    def m2(self):
        print ("this is m2 from A: ")

obj=C ()
obj.m1()
obj.m2()
```

Ex: **Constructor can be used 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 mul(self):
        pass

class B(A):
    def add(self):
        print(self.value+1)
    def mul(self):
        print(self.value-2)

obj=B (100)
obj.add()
obj.mul()
```