Python OOPS (Object Oriented Programming)

Dharavath Ramdas
linkedin: https://www.linkedin.com/in/dharavath-ramdas-a283aa213/

Classes:

1.Clasees logical entity
2.Classes is a Blue print
3.it contains Name, Attributes and Functions/Methods

Example:
1.Name : CAR (classes name)

2.Car properties / Atttribute
1.Brand
2.Color
3.Car function / Behaviour:
1.Moving()
2.reverse()

Object

1.Object is a Physical entity
2.Object follows class (blue print) so object is instance of classes
3.access class member then we create the object to access the class members

Example:

```
|Class       | ---->  Acess
|Attributes  |         /|\
|Methods     | -----> Object ----> Attributes ,Methods
```

Example:

```
logical entity              Object
    ^                          ^
    |                          |

[Human]                    [ Ram ]
[height, color] | <------  [170cm, black]
[walk(), run()]            [walk(), run(),sit() ]
```

1.We donot acess direct to class thats way we are created the object to access

2.self parameter refer paticular element

Class and object creating

In [6]:

```
class human: #class
    color = "white"
    height = 5.6
    def run(self):
        print("running.....")
    def walk(self):
        print("walking.....")

obj1 =human()  #object1
ram = human()  # object2
print("Color Attribute :",obj1.color)
print("Height Attribute :",obj1.height)
print("Method :",obj1.run())
print("Method :",obj1.walk())
print(" ")
print("Color Attribute :",ram.color)
print("Height Attribute :",ram.height)
print("Method :",ram.run())
print("Method :",ram.walk())
```

```
Color Attribute : white
Height Attribute : 5.6
running.....
Method : None
walking.....
Method : None

Color Attribute : white
Height Attribute : 5.6
running.....
Method : None
walking.....
Method : None
```

Constructor:

constructor using for object intialozation
when we create object constructor is called object intialization
__init__(self):

In [7]:

```
class human: #class
    color = "white"
    height = 5.6
    def __init__(self,c,h):
        self.color=c
        self.height=h
    def run(self,n):
        print(n, "running.....")
    def walk(self,m):
        print(m, "walking.....")

ram =human("yello",5.7)  #object
print(ram.color,ram.height)
tharun = human("black",6.8)
print(tharun.color,tharun.height)

ram.run('ram')
tharun.walk('tharun')
```

```
yello 5.7
black 6.8
ram running.....
tharun walking.....
```

Inheritance:-

1.parent - child relationship
2.Base class (Parent class)
3.Derived class (Child class)
4. child class aquaring the properties of parent class
5.only create derived class object donot create base class object

Single Inheritance:-

Child class aquaring the properties of parent class

single inheritance

```
base class
    |
   \|/
derived class
```

In [21]:

```
class baseclass:    #parent class
    a=10
    b=100
    def display(self):
        print("base class")
class derivedclass(baseclass):   #child class
    c=20
    d=200
    def show(self):
        print("derived class")
obj = derivedclass()
print(obj.a,obj.b,obj.c,obj.d)
obj.display()
obj.show()
```

```
10 100 20 200
base class
derived class
```

Multilevel Inheritance:-

Child class aquaring the properties from both parent and gand parents

multilevel inheritance

```
base class    #grand parent
    |
   \|/
derived class # parent
    |
   \|/
derived class   # child
```

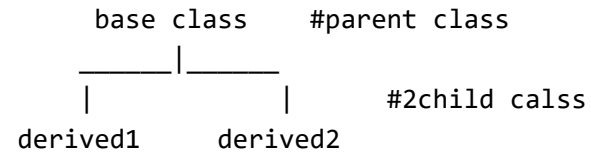
In [8]:

```
class gp:
    def gpdisplay(self):
        print("grandparent")
class parent(gp):
    def display(self):
        print("parent method")
class child(parent):
    def cdisplay(self):
        print("child display")
c=child()
c.gpdisplay()
c.display()
c.cdisplay()
```

```
grandparent
parent method
child display
```

Hierarchical Inheritance:-

1.One base class different derived class
2.two different derived classes aquaring the properties from the same parent calss



In [9]:

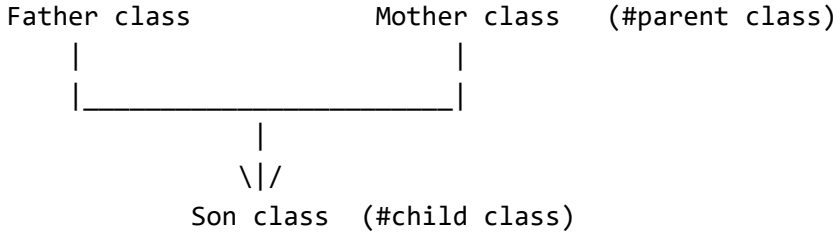
```
class parent:            # parent
    def pdis(self):
        print("parent class")
class son(parent):       #child1
    def sdis(self):
        print("son class")
class daughter(parent):    #child2
    def ddis(self):
        print("daughter class")

s=son()
s.pdis()
s.sdis()
d=daughter()
d.ddis()
d.pdis()
```

```
parent class
son class
daughter class
parent class
```

Multiple Inheritance:-

1.A child aquaring the properties from both father calss and mother class
2.Multiple base calss to one derived class



In [7]:

```
class father:
    def fdis(self):
        print("father")
class mother:
    def mdis(self):
        print("mother")
class child(father,mother):
    def chidis(self):
        print("child class")
c=child()
c.fdis()
c.mdis()
c.chidis()
```

```
father
mother
child class
```

In [ ]:

```

```

Polymarphism:-

1.implementing same thing in different ways

2 types of polymarphism:

1.compile time (method overloding)
    1.python not support
    2.but default parameter to support

2. runtime (method overriding)
    1.same method name override
    2.parent method overriding in child class like cycle to byke

Method overloding

In [11]:

```
# method overloding
class demo:
    def add(self,a,b=10,c=44):
        print(a+b+c)
a=demo()
a.add(10)
a.add(100,30)
a.add(20,40,60)
```

```
64
174
120
```

Method overriding

In [13]:

```
# method overriding
class parent:
    def transport(self):
        print("cycle")
class child(parent):
    def transport(self):
        print("byke")
c=child()
c.transport()
```

```
byke
```

Abstraction:-

Abstraction in python is defined as a process of handling complexity by hiding unn ecessary information from the user.

In [22]:

```
from abc import ABC,abstractmethod
class abstractdemo(ABC):
    @abstractmethod
    def housinginterest(self):
        None
    @abstractmethod
    def vehicleinterest(self):
        None
class sbi(abstractdemo):
    def housinginterest(self):
        print("housing 33")
    def vehicleinterest(self):
        print("3.3 ins")
o=sbi()
o.housinginterest()
```

```
housing 33
```

In [ ]:

```

```

Encapsulation:-

1.wraping of data and methods
2.we are storing the data of class,variable and method in a single class is called encapsulation
3.Data hiding --- Security
    1.public --> with in class and out side class
    2.private --> within class only

In [28]:

```
class encap:
    __a=10
    b=100
    def __display(self):
        print(self.__a)
        print("display method in demo class")
    def show(self):
        self.__display()
obj=encap()
obj.show()
```

```
10
display method in demo class
```

In [ ]:

```

```