OOPS!

Class:

a=[]

print(type(a))

o/p:

<class ‘list’>

- In python, everything will viewed in the form of class and objects.
- List is stored in different locations even if the contents of list are same, because list is mutable and list is viewed as object
- And if we check the id of the contents of 2 lists then it’ll be stored in the same location as the contents of list can’t be changed.

- CLASS is a user-defined datatype.
- There are few in-built classes such as list, tuple and dictionary.
- Whenever we refer class, we refer collectively
- maybe animals ,students, places etc.
- When we refer an object, we refer a particular thing
- maybe a person’s name, animals name etc.

Attributes and Behavior:

attributes: variables.

Behavior: functions.

collectively writing attributes and behavior forms class.

how to define class:

class class_name:

ex:

```
def __init __(self,x,y,z): { self is object, create a constructor and initialize 3 values which will be initialized below}
self.nickname=x
self.rollno=y
self.height=z
def run(self):
    print(”i can run”, self.height, self.roll) { self is object that we are passing}
harsha=person(”chintu”,78,6) {person() is a constructor)
anjali=person(”mary”,89,5.6)
harsha.run()
anjali.run()
```

o/p:

i can run chintu 78

i can run mary 89

#if we don’t create a constructor the contents will be common for all the objects.

constructor will initialize objects that are created.

difference between function and method:

method is also a function which is written inside a class
method is accessed using objects where function is directly accessed

Abstraction:

- it’s an idea which is not implemented.
- it is used to hide the unnecessary data and show only necessary data.
- in python logically abstraction, polymorphism and encapsulation does not work.

abstract method:

it is a method which doesn’t contain any body.

class person:

    def mobile():

        pass

```
'''abstraction'''
class mobile: #abstract class
    def functions(self): #abstract method
        pass
class iphone: #class
    def functions(self):
        print("This is iphone")
class samsung:
    def functions(self):
        print("This is samsung")
iphone13=iphone()
iphone13.functions()
samsungs3=samsung()
samsungs3.functions()
```

#polymorphism

class mobile: #abstract class

```
    def functions(self): #abstract method
        pass
    def functions(self,camera,display,battery):
        self.camera=camera
        self.display=display
        self.battery=battery
        print(self.camera)
        print(self.display)
        print(self.battery)
iphone=mobile()
iphone.functions(”12mp”,”4k”,”60mh”)
samsung=mobile()
samsung.functions(”24mp”,”6k”,”80mh”)
```

abstract class:

it is a class that contain abstract method.

#inheritance:

```
class mobile: #abstract class
    def functions(self): #abstract method
        pass
class iphone(mobile): '''passing the idea'''
    def functions(self):
        print("This is iphone")
class samsung(mmobile):
    def functions(self):
        print("This is samsung")
iphone13=iphone()
iphone13.functions()
```

```
samsungs3=samsung()
samsungs3.functions()
```

Encapsulation:

if anything is private we cant access them directly instead we access then using methods.”

“_” =private variables.

“__”=protected variables.

in python we can change the private variables using a public variable(loophole)

```
class car:
    _engine="v8"
    _wires="blue"
bmw=car()
bmw._engine="v9"
```

encapsulation can be implemented using getter and setter methods

methods are always public

variables can be either private, protected or public

```
class car:
    _engine="v8"
    _wires="blue"
    def getter(self):
        print(self._engine)
        print(self._wires)
    def setter(self,engine,wires):
        self._engine=engine
        self._wires=wires
bmw=car()
```

```
bmw.setter("v9","red")
bmw.getter()
```

```
class person:
   def __init__(self,x,y,z):      #self is object, create a cons
        self.nickname=x
        self.rollno=y
        self.height=z
   def run(self):
       print("i can run",self.height, self.roll)    #{ self is c
harsha=person("chintu",78,6)                          #{person()
anjali=person("mary",89,5.6)
harsha.run()
anjali.run()
```

Inheritance:

1. single
2. multiple
3. multilevel
4. hierarchical
5. hybrid

1. single

inheriting from one single class

```
class parents:
    def coolness(self):
        print("parents are cool")

class child(parents):
```

```
    def coding(self):
        print("i know coding")
yashu=child()
yashu.coolness()
yashu.coding()
```

2. multilevel

a class(child2) is inherited from another class(child) but that class(child) is inherited from its parent class(parents)

```
class parents:
    def coolness(self):
        print("parents are cool")
class child(parents):
    def coding(self):
        print("i know coding")
class child2(child):
    def singing(self):
        print("i can sing")
yashu=child2()
yashu.coolness()
yashu.coding()
yashu.singing()
```

3.multiple:

a child will inherit from 2 parents

```
class dad:
    def coolness(self):
        print("parents are cool")
class mom:
    def coding(self):
        print("i know coding")
class child(dad,mom):
    def singing(self):
```

```
        print("i can sing")
yashu=child()
yashu.coolness()
yashu.coding()
yashu.singing()
```

4. hierarchical:

```
class grandfather:
    def coolness(self):
        print("parents are cool")
class father(grandfather):
    def coding(self):
        print("i know coding")
class daughter(father):
    def singing(self):
        print("i can sing")
yashu=daughter()
yashu.coolness()
yashu.coding()
yashu.singing()
```

5. hybrid:

```
class grandfather:
    def coolness(self):
        print("i'm cool")
class father(grandfather):
    def coding(self):
        print("i know coding")
class mother(grandfather):
    def cooking(self):
        print("i can cook")
```

```
class daughter(father,mother):
    def singing(self):
        print("i can sing")
yashu=daughter()
yashu.coolness()
yashu.coding()
yashu.cooking()
yashu.singing()
```

Polymorphism:

poly: many

morphism: forms

overloading:

same name and different parameters.

```
class add:
    def sum(self,x,y):
        print(x+y)
    def sum(self,x,y,z):
        print(x+y+z)
i=add()
i.sum(10,8)
i.sum(10,8,1)
```

o/p:

error

```
class add:
    def sum(self,x,y):
        print(x+y)
class child(add):
    def sum(self,x,y,z):
```

```
        print(x+y+z)
i=add()
i.sum(10,8)
```

o/p:

18

overriding: