OOPS:

Python is object oriented programing language.

Object has two characterstics:

Attributes

Action

Human: Object

Attributes: Name, age , Gender

Action : Walking ,sleeping

Python:

Class:

Object

class : Class is a blueprint of a object.

Blueprint of Home is class. Consution of home based on blueprint of Home(Physical) is Obj.

class className:

-------

-------

Ex:

class Computer2:

pass

Prog 1:

class computer:

    color="Black"

    def \_\_init\_\_(self,computer,ram,cpu):

        self.computer=computer

        self.ram=ram

        self.cpu=cpu

    def description(self,hd):

        print("com config: com Name:",self.computer,"CPU :",self.cpu,"RAM::",self.ram)

        print('Hard Disk:',hd)  #Instance method. so not to use self.

c1=computer('Dell',8,'I7')  # Object creation

c2=computer("Lenova",16,'I5') #Object creation

print("Before------------")

print("c1 attributes:",c1.computer,c1.ram,c1.cpu) # Dell 8 17

print("c2 attributes:",c2.computer,c2.ram,c2.cpu) # Lenova,16 I5

print("After-------------------")

c1.computer="Inspiron"

c2.ram=24

print("c1 attributes:",c1.computer,c1.ram,c1.cpu)  #Inspiron 8 17

print("c2 attributes:",c2.computer,c2.ram,c2.cpu)  #Lenova,24,I5

c1.description(500)   #Com Config: Com Name: Inspiron Ram size: 8 cpu: I7  ard Disk: 500

c2.description(1000)  #Com Config: Com Name: Lenova Ram size: 24 cpu: I5 and Hard Disk: 1000

OUTPUT:

$ python oops1.py

Before------------

c1 attributes: Dell 8 I7

c2 attributes: Lenova 16 I5

After-------------------

c1 attributes: Inspiron 8 I7

c2 attributes: Lenova 24 I5

com config: com Name: Inspiron CPU : I7 RAM:: 8

Hard Disk: 500

com config: com Name: Lenova CPU : I5 RAM:: 24

Hard Disk: 1000

2.

FUNCTION overloading

Function with Object have many forms.

class Tomoto:

    def Type(self):

        print("Veg")

    def color(self):

        print("Red")

class Apple:

    def Type(self):

        print("Fruit")

    def color(self):

        print("Red")

def Func(obj): #Function with obj have many forms.

    obj.Type()

    obj.color()

Obj\_Tomoto=Tomoto()

Obj\_Apple=Apple()

Func(Obj\_Tomoto) #Veg , Red

Func(Obj\_Apple) #Fruit, Red

3.

Inheritance:

Inheritance allows us to define a class that inherits all the methods and properites from another class.

Base class (Attributes/Methods) <---------------- Derived class (Attributes/Methods) . THe object can created in Derived class(Child) and access the methods /Attributes from base class.

class Base\_Class:

Attributes

Methods

class Der\_Class(Base\_Class):

Attributes

Methods

Object creation in Derived class only. obj1=Der\_Class(\_). obj1 can access all the methods and attributes in Base class.

class Calculation1:

    def sum(self,a,b):

        return a+b

 #Derived Class

class Calc(Calculation1):

    def \_\_init\_\_(self,a,b):

        self.a=a

        self.b=b

    def div(self,a,b):

        return a/b

obj1=Calc(8,4)

div=obj1.div(obj1.a,obj1.b) #Derived class accessing

sum=obj1.sum(obj1.a,obj1.b) #Base class accessing

print("Div: ",div) #2.0

print("Sum: ",sum) #12

Code 3:

Inheritance 2: Friend function

class Calculation1:

    def \_\_init\_\_(self,c,d):

        self.c=c

        self.d=d

    def sum(self,a,b):

        return a+b

 #Derived Class

class Calc(Calculation1):

    def \_\_init\_\_(self,a,b,c,d):

        self.a=a

        self.b=b

        super().\_\_init\_\_(c,d)  #This is for to send the parameter to \_\_init method of base class.

    def div(self,a,b):

        return a/b

obj1=Calc(8,4,22,11)

div=obj1.div(obj1.a,obj1.b)

sum=obj1.sum(obj1.c,obj1.d) #The calculation is based on parameters passed to base class through Derived class.

print("Div: ",div)

print("Sum: ",sum)

Output:

Div: 2.0

Sum: 12

Code 4:

Multiple Inheritance:

class Mother():

    mother\_name=' '

    def print\_mother(self):

        print("Mother name: ",self.mother\_name)

#Base class 2

class Father():

    father\_name=' '

    def print\_father(self):

        print("Father name: ",self.father\_name)

#Child class

class child(Mother,Father):

    def print\_parent(self):

        print("Child class prints-----------")

        print("Father :",self.father\_name)

        print("Mother :",self.mother\_name)

obj1=child()

obj1.mother\_name='sita'#Mother name:  sita

obj1.father\_name='Ram' #Father name:  Ram

print("Base class prints ------------------ ")

obj1.print\_mother() #Father : Ram

obj1.print\_father() #Mother : sita

Output:

$ python inheritance3.py

Base class prints ------------------

Mother name: sita

Father name: Ram

Code 5:

Multilvel Inheritance

  #Base Class 1. Multiple Inheritance.

class Grandparent:

    Grandparent\_name=' '

    def gp\_name(self):

        print("Gp name:",self.Grandparent\_name)

#Base Class 2:

class Mother:

    mother\_name=' '

    def print\_mother(self):

        print("Mother name: ",self.mother\_name)

#Base class 3

class Father(Grandparent):

    father\_name=' '

    def print\_father(self):

        print("Father name: ",self.father\_name)

#Child class

class child(Mother,Father):

    def print\_parent(self):

        print("Child class prints-----------")

        print("Father :",self.father\_name)

        print("Mother :",self.mother\_name)

obj1=child()

obj1.mother\_name='sita'

obj1.father\_name='Ram'

obj1.Grandparent\_name='Ravi'

print("Base class prints ------------------ ")

obj1.print\_mother()

obj1.print\_father()

obj1.gp\_name()

obj1.print\_parent()

GP

FATHER MOTHER

CHILD

Hierachical Inheritance:

Base Class <---- Derived class 1

<----- Derived class 2

Multiple inheritance invovles multiple inheritance taking place in a single program.

GrandFather-->Father ----------Single Level Inheritance

GP ---> Father --->Child -----------------> Multilevel Inheritance

Mother/Father ----> Child -- Multiple Inheritance

Gp-->Mother/Father --> Child ----> Hybrid Inheritance

Single : Base / Derived

Multiple : Base 1 /Base 2 -- Derived

Multilevel : Base 1---> Base2--->Base3---> Derived

Hierarchical : Base 1---> Derived 1/Derived 2

Hybrd: Mix of above all.

POLY MORPHISM:

class Table:

    def \_\_init\_\_(self,fare):

        self.fare=fare

    def \_\_add\_\_(self,other1):

    #return self.fare + other1.fare #110

        return self.fare \*   other1.fare #3000. We can have our own logic for operator

    def \_\_lt\_\_(self,other):

        return self.fare < other.fare

t1=Table(50)

t2=Table(60)

print("Addition:",t1+t2)

print("Real meaning:",3+4)

print("Operator comparision:",t1<t2) #TRUE

#print("Operator comparision:",t1<=t2) #TRUE.TypeError: '<=' not supported between instances of 'Table' and 'Table'

Output:

$ python Poly.py

Addition: 110

Real meaning: 7

Operator comparision: True

Code 5:

METHOD Overloading:

class Tomoto:

    def Type(self):

        print("Veg")

    def color(self):

        print("Red")

class Apple:

    def Type(self):

        print("Fruit")

    def color(self):

        print("Red")

def Func(obj):

    obj.Type()

    obj.color()

Obj\_Tomoto=Tomoto()

Obj\_Apple=Apple()

Func(Obj\_Tomoto) #

Func(Obj\_Apple)

Output:

Veg

Red

Fruit

Red

Code 6:

METHOD OVERLOADING 2:

class India():

    def capital(self):

        print("New Delhi")

    def language(self):

        print("Hindi , English")

class USA():

    def capital(self):

        print("Washingon DC")

    def language(self):

        print("English")

obj\_India=India()

obj\_USA=USA()

for country in(obj\_India,obj\_USA): #Methods classes

    country.capital()     #New Delhi  Hindi , English

    country.language()    # Washingon DC    English . Methods classes overloading

OUTPUT:

$ python method\_overloading2.py

New Delhi

Hindi , English

Washingon DC

English

CODE 5:

METHOD\_OVERRINDG

class Bird():

    def about(self):

        print("They are diff types of birds")

    def fly(self):

        print("Some birds can fly and some or not fly")

class parrot(Bird):

    def fly(self):

        print("Parrot can fly")

class penguin(Bird):

    def fly(self):

        print("Penguin cannot fly")

obj\_Bird=Bird()

obj\_parrot=parrot()

obj\_penguin=penguin()

print('--------- Bird---------')

obj\_Bird.about() #They are diff types of birds

obj\_Bird.fly() #Some birds can fly and some or not fly

print("parraot------------------")

obj\_parrot.about() #They are diff types of birds

obj\_parrot.fly() #Parrot can fly

obj\_penguin.about()#They are diff types of birds

obj\_penguin.fly() # Priority will given to Derived class. Penguin cannot fly

OUTPUT:

$ python method\_overriding.py

--------- Bird---------

They are diff types of birds

Some birds can fly and some or not fly

parraot------------------

They are diff types of birds

Parrot can fly

They are diff types of birds

Penguin cannot fly

PYTHON Automation:

OOPS is about class, objects, inheritance, Polymorphism, Data Abstraction and Data Encapusulation

A screenshot of a computer

AI-generated content may be incorrect.

* To group related functions (Methods)
* To create a template /blueprint
* OOPs is a concept where characteristic and functions of a real-life objects is packaged as a single entity in the code

#!/bin/python

def main():

tomcat7=”/home/Automation/tomcat7/conf/server.xml”

tomcat9=”/home/Automation/tomcat9/conf/server.xml”

If \_\_name\_\_==”\_\_main\_\_”:

Main()