OOPs:

----

1.class

2.object

3.Reference variable

1.class:

--------

class is blue print or paln or model or design to create objects.

class represents properties and actions of object

2.object:

--------

Physical existance of class is nothing but object or physical instance of class is nothing

but object.

By one class to develop any number of objects.

3.Reference variable:

------------------

The variable which can used to reffer an object is called Reference variable.

By useing reffernce variable we can access properties (variables) and actions(methods)

of object.

syntax:

------

class Class\_Name:

''' Doc string '''

variables

methods

eg:

class Student:

'''class devloped for student attandance'''

#variables

#methods

print(Student.\_\_doc\_\_ )

output:

class devloped for student attandance

eg:

class Student:

'''class devloped for student attandance'''

#variables

#methods

print(help(Student))

output:

Help on class Student in module \_\_main\_\_:

class Student(builtins.object)

| class devloped for student attandance

|

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

None

In side python class three types variables are allowed

1.Instance variables(object level variables)

2.Static variables(class level variables)

3.Local variables(method level variables)

In side python class three types methods are allowed

1.Instance method

2.class method

3.static method

eg:

----

class Student:

'''This is devloped by jagan'''

def \_\_init\_\_(self):

self.name="jagan"

self.rollno=222

self.marks=70

def info(self):

print("Name:",self.name)

print("Roll No:",self.rollno)

print("Marks:",self.marks)

s=Student()

print(s.name)

print(s.rollno)

print(s.marks)

s.info()

output:

jagan

222

70

Name: jagan

Roll No: 222

Marks: 70

eg:

class Student:

def \_\_init\_\_ (self,name,rollno,marks):

self.name=name

self.rollno=rollno

self.marks=marks

def info(self):

print("Name:",self.name)

print("Rollno:",self.rollno)

print("Marks:",self.marks)

s1=Student("Jagan",222,70)

s2=Student("Ram",100,80)

s3=Student("Raj",200,90)

print(s1.name)

print(s1.rollno)

print(s1.marks)

print(s2.name)

print(s2.rollno)

print(s2.marks)

print(s3.name)

print(s3.rollno)

print(s3.marks)

s1.info()

s2.info()

s3.info()

ouput:

Jagan

222

70

Ram

100

80

Raj

200

90

Name: Jagan

Rollno: 222

Marks: 70

Name: Ram

Rollno: 100

Marks: 80

Name: Raj

Rollno: 200

Marks: 90

About self:

-----------

self is a references variable which is pointing to the current object.

eg:

class Test:

def \_\_init\_\_(self):

pass

t=Test()

't' is reference variable for this object But 't' will be declared outside of the class

To reffer object inside of the class we can go for self

eg:

class Test:

def \_\_init\_\_(self):

print("Addres of object pointed by self:",id(self))

t=Test()

print("Addres of object pointed by t:",id(t))

output:

Addres of object pointed by self: 1899563247600

Addres of object pointed by t: 1899563247600

eg:

class Test:

def \_\_init\_\_(self):

print("Addres of object pointed by self:",id(self))

t1=Test()

print("Addres of object pointed by t1:",id(t1))

t2=Test()

print("Addres of object pointed by t2:",id(t2))

t3=Test()

print("Addres of object pointed by t3:",id(t3))

output:

Addres of object pointed by self: 2392176520104

Addres of object pointed by t1: 2392176520104

Addres of object pointed by self: 2392176520216

Addres of object pointed by t2: 2392176520216

Addres of object pointed by self: 2392176520272

Addres of object pointed by t3: 2392176520272

self is first argument inside the constructor and Instance method

we no need to pass value for self argument in constructor and instance method

self is not a key word it is a variable.Insted of self we can take any name

eg:

class Test:

def \_\_init\_\_ (xyz):

xyz.name="Jagan"

xyz.marks=75

def info(xyz):

print("Name:",xyz.name)

print("Marks:",xyz.marks)

t1=Test()

print(t1.name)

print(t1.marks)

t1.info()

output:

Jagan

75

Name: Jagan

Marks: 75

constructor:

------------

Itis special method in python

The constructor name is always fixed i.e \_\_init\_\_

when ever creating an object constructor will exicuted

Each object constructor will excicute only once

eg:

class Test:

def \_\_init\_\_(self):

print("Constructor")

t1=Test()

t2=Test()

t3=Test()

t4=Test()

output:

Constructor

Constructor

Constructor

Constructor

The main purpose of constructor is to declare and intilige instance variables

eg:

class Student:

def \_\_init\_\_ (self,name,rollno,marks):

self.name=name

self.rollno=rollno

self.marks=marks

s1=Student("Jagan",222,70)

s2=Student("Ram",999,90)

print(s1.name,s1.rollno,s1.marks)

print(s2.name,s2.rollno,s2.marks)

output:

Jagan 222 70

Ram 999 90

constructor is optional

we didn't write constructor to our code. python will execuite default constructor

eg:

class Test:

def m1(self):

print("Method1")

t=Test()

t.m1()

output:

Method1

eg:

class Test:

def \_\_init\_\_(self):

print("constructor")

t=Test()

t.\_\_init\_\_()

t.\_\_init\_\_()

output:

constructor

constructor

constructor

More than one constructor to code lastet constructor only it is taking

eg:

class Test:

def \_\_init\_\_(self):

print("constructor1")

def \_\_init\_\_(self):

print("constructor2")

t=Test()

output:

constructor2

eg:

class Test:

def \_\_init\_\_(self,x):

print("constructor1")

def \_\_init\_\_(self):

print("constructor2")

t=Test()

output:

constructor2

eg:

class Test:

def \_\_init\_\_(self):

print("constructor1")

def \_\_init\_\_(self,x):

print("constructor2")

t=Test()

output:

Traceback (most recent call last):

File "test.py", line 6, in <module>

t=Test()

TypeError: \_\_init\_\_() missing 1 required positional argument: 'x'

eg:

class Test:

def \_\_init\_\_(self):

print("constructor1")

def \_\_init\_\_(self,x):

print("constructor2",x)

t=Test("Hello")

output:

constructor2 Hello

eg:

class States:

def \_\_init\_\_(self,sname,scaptial):

self.sname=sname

self.scaptial=scaptial

def info(self):

print("State Name:",self.sname)

print("State Captial:",self.scaptial)

list\_of\_states=[]

print(list\_of\_states)

while True:

sname=input("Enter State Name:")

scaptial=input("Eneter Captial Name:")

s=States(sname,scaptial)

list\_of\_states.append(s)

print("State information added sucessfully")

option=input("Do you want enter one more state [yes/no]")

if option.lower()=='no':

break

print(list\_of\_states)

for state in list\_of\_states:

state.info()

print()

output:

[]

Enter State Name:"TS"

Eneter Captial Name:"HYD"

State information added sucessfully

Do you want enter one more state [yes/no]yes

Enter State Name:"AP"

Eneter Captial Name:"AMARAVATHI"

State information added sucessfully

Do you want enter one more state [yes/no]no

[<\_\_main\_\_.States object at 0x000002248E1EEBA8>, <\_\_main\_\_.States object at 0x000002248E1EEC18>]

State Name: "TS"

State Captial: "HYD"

State Name: "AP"

State Captial: "AMARAVATHI"

Variables inside python class:

-------------------------------

1.Instance variables or Object level variables:

-----------------------------------------------

The variables which are varied from object to object are known as Instance variables

Eg:

name,rollno,marks

Every object a separate instance variables are available

Declare the instance variables inside construtor or inside instance method

by using self

eg:

class Student:

def \_\_init\_\_ (self,name,rollno,marks):

self.name=name

self.rollno=rollno

self.marks=marks

def info(self):

self.grade="A"

2.static variables or class level variables:

-------------------------------------------

1>The variables which are same for every object or the variable which is not changed from object

to object such type of variable declareed at class level .This type of variables are known as

Static variables

2>Only one set of static variables are available.These static variables shared to every object.

eg:

class Student:

college\_Name="ABCD College"

def \_\_init\_\_ (self,name,rollno,marks):

self.name=name

self.rollno=rollno

self.marks=marks

print(Student.college\_Name)

print(self.college\_Name)

s1=Student("Jagan",222,70)

s2=Student("Raj",202,70)

output:

ABCD College

ABCD College

ABCD College

ABCD College

3. local variables or method level variables:

---------------------------------------------

Th variables which are used to meet the temporry requirements such type of variables

are known as locla variables.

eg:

class Student:

def info(self):

x=10

for i in range(x):

print(i)

eg:

class Test:

school\_name="ABCD school"

def \_\_init\_\_(self,name,marks):

self.name=name

self.marks=marks

print(Test.school\_name)

def info(self):

x=10

for i in range(x):

print(i)

t1=Test("Jagan",200)

print(t1.name)

print(t1.marks)

output:

ABCD school

Jagan

200

methods inside python class:

-------------------------------

1.Instance method:

------------------

Inside method we are accessing instance variables and the method allways talks about

perticular object only .Such cases we can go for instance method

The first argument inside instance method is self. self is reffrence variable which is pointing

only current object

eg:

class Test:

def info(self):

print(self.name)

print(self.marks)

2.class method:

--------------

Inside method we can access the only class level variables or static variables we go

for class method.

To declare class method by using @classmethod decartor

The first argument inside class method is cls

eg:

class Test:

school\_name="ABCD school"

@classmethod

def m2(cls):

print("Scool Nmae:",cls.school\_name)

eg:

class Test:

school\_name="ABCD"

@classmethod

def f1 (cls):

print(id(cls))

print("Name:",cls.school\_name)

t1=Test()

t1.f1()

print(id(Test))

output:

2144905879944

Name: ABCD

2144905879944

3.Static method:

-----------------

Inside static method if you not using any object level information or class level information

then we go for static method

static method declared using @staticmethod decartor

eg:

class Test:

@staticmethod

def sum1(a,b):

return a+b

t1=Test()

print(t1.sum1(5,6))

output:

11

eg:

class Student:

school\_name="ABCD school"

def \_\_init\_\_(self,name,marks):

self.name=name

self.marks=marks

def getinfo(self):

print("Name is:",self.name)

print("Marks are:",self.marks)

@classmethod

def get\_school(cls):

print("Schhol Name:",cls.school\_name)

@staticmethod

def sum1(a,b):

return a+b

s1=Student("Jagan",70)

s1.getinfo()

s1.get\_school()

print(s1.sum1(10,20))

output:

Name is: Jagan

Marks are: 70

Schhol Name: ABCD school

30

eg:

class Bank:

'''Deposit and Withdraw application'''

bankname="SBI"

def \_\_init\_\_ (self,name,balance=0.0):

self.name=name

self.balance=balance

def deposit(self,ammount):

self.balance=self.balance+ammount

print("After deposit total ammount:",self.balance)

def withdraw(self,ammount):

if ammount > self.balance:

print("Insufficent balance to with draw")

else:

self.balance=self.balance-ammount

print("After withdraw balance:",self.balance)

print("Welcome to :",Bank.bankname)

name=input("Enter your Name:")

b=Bank(name)

while True:

print("""D - Deposit

W - WithDraw

E - Exit """)

option=input("Choose your option:")

if option.lower()=='d':

ammount=float(input("Enter ammount to deposit:"))

b.deposit(ammount)

elif option.lower()=="w":

ammount=float(input("Enter ammount to withdraw:"))

b.withdraw(ammount)

elif option.lower()=='e':

print("Thanks for Banking")

break

else:

print("Enterd option is Invalid")

output:

Welcome to : SBI

Enter your Name:"Jagan"

D - Deposit

W - WithDraw

E - Exit

Choose your option:D

Enter ammount to deposit:1000

After deposit total ammount: 1000.0

D - Deposit

W - WithDraw

E - Exit

Choose your option:D

Enter ammount to deposit:500

After deposit total ammount: 1500.0

D - Deposit

W - WithDraw

E - Exit

Choose your option:w

Enter ammount to withdraw:200

After withdraw balance: 1300.0

D - Deposit

W - WithDraw

E - Exit

Choose your option:w

Enter ammount to withdraw:10000

Insufficent balance to with draw

D - Deposit

W - WithDraw

E - Exit

Choose your option:w

Enter ammount to withdraw:1000

After withdraw balance: 300.0

D - Deposit

W - WithDraw

E - Exit

Choose your option:e

Thanks for Banking

Garbage collector:

--------------------

If the object doesnot having any reference variables then only object is eligible

garbage collections

eg:

import gc

print(gc.isenabled())

gc.disable()

print(gc.isenabled())

gc.enable()

print(gc.isenabled())

output:

True

False

True

Destructor:

----------

Destructor is special method and the name of destructor is \_\_del\_\_()

Just before destroying an object Garbage collector calls the destructor to perform some

cleanup activites eg: close db connections

once destructor exevution is completed automaticll gc destroy the object.

eg:

class Test:

def \_\_init\_\_ (self):

print("Object creation")

def \_\_del\_\_(self):

print("clean up activities")

t=Test()

t=None

print("End of application")

output:

Object creation

clean up activities

End of application

eg:

class Test:

def \_\_init\_\_(self):

print("object created")

def \_\_del\_\_(self):

print("Cleanup activities")

t1=Test()

t2=Test()

print("End application")

output:

object created

object created

End application

Cleanup activities

Cleanup activities

eg:

class Test:

def \_\_init\_\_(self):

print("object created")

def \_\_del\_\_(self):

print("Cleanup activities")

l=[Test(),Test(),Test(),Test(),Test()]

del l

output:

object created

object created

object created

object created

object created

Cleanup activities

Cleanup activities

Cleanup activities

Cleanup activities

Cleanup activities

Inhertance:

----------

The concept of inherting members from one class to another class is known as inhertance

Advantage:

---------

1>There is no code duplicate

2>Length of code will be decreases

3>Performance improved

Types of inhertance are:

------------------------

1.single inhertance

2.multilevel inhertance

3.Hireachial inhertance

4.multiple inhertance

5.Hybrid inhertance

6.cyclic inhertance

1.single inhertance:

--------------------

The concept of inherting members from one class to another class is known as

single inhertance.

In single inhertance only one Parent and one child class

eg:

class P:

def m1(self):

print("House+Land+Car")

class C:

def m2(self):

print("Bicycle")

c=C()

c.m2()

c.m1()

output:

Bicycle

Traceback (most recent call last):

File "test.py", line 10, in <module>

c.m1()

AttributeError: 'C' object has no attribute 'm1'

eg:

class P:

def m1(self):

print("House+Land+Car")

class C(P):

def m2(self):

print("Bicycle")

c=C()

c.m2()

c.m1()

output:

Bicycle

House+Land+Car

eg:

class P:

def m1(self):

print("House+Land+Car")

class C(P):

def m2(self):

print("Bicycle")

c=C()

c.m2()

c.m1()

p=P()

p.m1()

p.m2()

output:

Bicycle

House+Land+Car

House+Land+Car

Traceback (most recent call last):

File "test.py", line 14, in <module>

p.m2()

AttributeError: 'P' object has no attribute 'm2'

2.multilevel inhertance:

-------------------------

The concept of inherting members from multiple classes to single class is known as

multilevel inhertance

eg:

class GP:

def m1(self):

print("House+Land")

class P:

def m2(self):

print("Car+Shop")

class C(P):

def m3(self):

print("Bicycle")

c=C()

c.m2()

c.m3()

output:

Car+Shop

Bicycle

eg:

class GP:

def m1(self):

print("House+Land")

class P:

def m2(self):

print("Car+Shop")

class C(P):

def m3(self):

print("Bicycle")

c=C()

c.m2()

c.m3()

c.m1()

output:

Car+Shop

Bicycle

Traceback (most recent call last):

File "test.py", line 14, in <module>

c.m1()

AttributeError: 'C' object has no attribute 'm1'

eg:

class GP:

def m1(self):

print("House+Land")

class P(GP):

def m2(self):

print("Car+Shop")

class C(P):

def m3(self):

print("Bicycle")

c=C()

c.m2()

c.m3()

c.m1()

output:

Car+Shop

Bicycle

House+Land

3.Hireachial inhertance:

--------------------------

The concept of inherting members of one class to mutiple classes which are present at same

level is known as Hireachial inhertance

eg:

class P:

def m1 (self):

print("Parent class m1 method")

class C1(P):

def m2(self):

print("Child 1 m1 method")

class C2(P):

def m3(self):

print("Child 2 m3 method")

c1=C1()

c1.m2()

c1.m1()

c2=C2()

c2.m3()

c2.m1()

output;

Child 1 m1 method

Parent class m1 method

Child 2 m3 method

Parent class m1 method

eg:

class P:

def m1 (self):

print("Parent class m1 method")

class C1(P):

def m2(self):

print("Child 1 m1 method")

class C2(P):

def m3(self):

print("Child 2 m3 method")

c1=C1()

c1.m2()

c1.m1()

c1.m3()

output:

Child 1 m1 method

Parent class m1 method

Traceback (most recent call last):

File "test.py", line 13, in <module>

c1.m3()

AttributeError: 'C1' object has no attribute 'm3'

4.multiple inhertance:

-----------------------

It is reverse of Hireachial inhertance

The concept of inherting members from multiple classes to single class is known as

multiple inhertance

eg:

class P1:

def m1 (self):

print("Parent class1 m1 method")

class P2:

def m2 (self):

print("Parent class2 m2 method")

class C(P1,P2):

def m3(self):

print("It is child class m3 method")

c=C()

c.m3()

c.m2()

c.m1()

output:

it is child class m3 method

Parent class2 m2 method

Parent class1 m1 method

eg:

class P1:

def m1 (self):

print("Parent class1 m1 method")

class P2:

def m2 (self):

print("Parent class2 m2 method")

class C(P2,P1):

def m3(self):

print("It is child class m3 method")

c=C()

c.m3()

c.m2()

c.m1()

output:

It is child class m3 method

Parent class2 m2 method

Parent class1 m1 method

5.Hybrid inhertance:

-----------------

Hybrid inhertance is combination of single,multilevel ,multiple and Hireachial inhertance

Note:

In Hybrid inhertance method resolution is based on MRO (Method Resolution Order) algorithm

6.cyclic inhertance:

--------------------

The concept of inherting members from one class to another class in cyclic way is known as

cyclic inhertance

Note: cyclic inhertance does not support python

eg:

class A(B):

pass

class B(A):

pass

a=A()

output:

Traceback (most recent call last):

File "test.py", line 1, in <module>

class A(B):

NameError: name 'B' is not defined

super():

--------

eg:

class P:

def m1(self):

print("Parent method")

class C(P):

def m2(self):

print("Child method")

c=C()

c.m2()

output:

Child method

eg:

class P:

def m1(self):

print("Parent method")

class C(P):

def m2(self):

self.m1()

print("Child method")

c=C()

c.m2()

output:

Parent method

Child method

if parent class and child class having same method names

eg:

class P:

def m1(self):

print("Parent method")

class C(P):

def m1(self):

self.m1()

print("Child method")

c=C()

c.m1()

output:

Traceback (most recent call last):

File "test.py", line 9, in <module>

c.m1()

File "test.py", line 6, in m1

self.m1()

File "test.py", line 6, in m1

self.m1()

File "test.py", line 6, in m1

self.m1()

[Previous line repeated 995 more times]

RecursionError: maximum recursion depth exceeded

super() is a python in built method to access parent class methods and variables

eg:

class P:

def m1(self):

print("Parent method")

class C(P):

def m1(self):

super().m1()

print("Child method")

c=C()

c.m1()

output:

Parent method

Child method

eg:

class P:

def \_\_init\_\_(self):

print("Parent class construtor")

def m1(self):

print("Parent class m1 method")

class C(P):

def \_\_init\_\_(self):

print("Child class construtor")

def m2 (self):

print("childclass m2 method")

c=C()

c.m2()

c.m1()

output:

Child class construtor

childclass m2 method

Parent class m1 method

eg:

class P:

def \_\_init\_\_(self):

print("Parent class construtor")

def m1(self):

print("Parent class m1 method")

class C(P):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def m2 (self):

print("childclass m2 method")

c=C()

c.m2()

c.m1()

output:

Parent class construtor

childclass m2 method

Parent class m1 method

Polymorphism:

---------------

poly means many and morphs means forms

Polymorphism is nothing but many forms

operator over loading:

-----------------------

+ operator:

------------

print(10+20)-------->30

print("hello"+"world")-------->"helloworld"

\* operator:

------------

print(10\*2)--------->20

print(2\*"Hello")------->HelloHello

Method over riding:

---------------------

eg:

class P:

def property(self):

print("Bicycle")

class C(P):

def property(self):

print("Bike")

c=C()

c.property()

output:

Bike

1.over loading:

----------------

1>operator over loading:

------------------------

Same operator we can use multiple purpose is nothing but operator over loading

+ operator:

------------

print(10+20)-------->30

print("hello"+"world")-------->"helloworld"

\* operator:

------------

print(10\*2)--------->20

print(2\*"Hello")------->HelloHello

eg:

class Book:

def \_\_init\_\_(self,pages):

self.pages=pages

b1=Book(100)

b2=Book(200)

print(b1+b2)

output:

Traceback (most recent call last):

File "test.py", line 7, in <module>

print(b1+b2)

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

Magical methods:

---------------

Every operator internally having one magical method .When ever operator is excuited internally

coresponding magical is exicuted.

magical method for + operator is \_\_add\_\_()

print(10+20)

magical method for - operator is \_\_sub\_\_()

print(10-20)

eg:

class Book:

def \_\_init\_\_ (self,pages):

self.pages=pages

def \_\_add\_\_(self,other):

totalpages=self.pages+other.pages

return totalpages

b1=Book(200)

b2=Book(100)

print(b1+b2)

output:

300

eg:

class Book:

def \_\_init\_\_ (self,pages):

self.pages=pages

def \_\_sub\_\_ (self,other):

difference=self.pages-other.pages

return difference

b1=Book(200)

b2=Book(100)

print(b1-b2)

output:

100

magical methods :

----------------

+ ----------> \_\_add\_\_(self,other)

- ----------> \_\_sub\_\_(self,other)

\* ----------> \_\_mul\_\_(self,other)

/ ----------> \_\_div\_\_(self,other)

// ----------> \_\_floordiv\_\_(self,other)

% ----------> \_\_mod\_\_(self,other)

\*\* ----------> \_\_pow\_\_(self,other)

< ----------> \_\_lt\_\_(self,other)

<= ----------> \_\_le\_\_(self,other)

> ----------> \_\_gt\_\_(self,other)

>= ----------> \_\_ge\_\_(self,other)

== ----------> \_\_eq\_\_(self,other)

!= ----------> \_\_ne\_\_(self,other)

eg:

class Student:

def \_\_init\_\_(self,name,marks):

self.name=name

self.marks=marks

def \_\_gt\_\_(self,other):

return self.marks > other.marks

def \_\_lt\_\_(self,other):

return self.marks < other.marks

s1=Student("jagan",80)

s2=Student("Mohan",90)

print(s1>s2)

print(s1<s2)

output:

False

True

method over loading:

---------------------

Method over loading nothing but same method name with multiple number of arguments

Method over loading not supported in python

eg:

class Test:

def m1(self):

print("No args method")

def m1(self,x):

print("one args method")

def m1(self,x,y):

print("Two args method")

t=Test()

t.m1()

outoput:

Traceback (most recent call last):

File "test.py", line 9, in <module>

t.m1()

TypeError: m1() missing 2 required positional arguments: 'x' and 'y'

eg:

class Test:

def m1(self):

print("No args method")

def m1(self,x):

print("one args method")

def m1(self,x,y):

print("Two args method")

t=Test()

t.m1(1,2)

output:

Two args method

eg:

class Test:

def sum\_1(self,\*args):

print(args)

total=0

for x in args:

total=total+x

print("Total:",total)

t=Test()

t.sum\_1()

t.sum\_1(4)

t.sum\_1(1,2,3)

output:

()

Total: 0

(4,)

Total: 4

(1, 2, 3)

Total: 6

over riding:

-----------

What ever methods present in parent class are by default available to child class through

inhertance.

some times child class may not satisfy with the parent class methods then child

class is allowed to define their own methods.

eg:

class P:

def property(self):

print("Land+Cash")

def vehicle(self):

print("TVS Bike")

class C(P):

pass

c=C()

c.property()

c.vehicle()

output:

Land+Cash

TVS Bike

eg:

class P:

def property(self):

print("Land+Cash")

def vehicle(self):

print("TVS Bike")

class C(P):

def vehicle(self):

print("Royla Enfield Bike")

c=C()

c.property()

c.vehicle()

output:

Land+Cash

Royla Enfield Bike

e:

class P:

def \_\_init\_\_(self):

print("Parent constructor")

class C(P):

def \_\_init\_\_(self):

print("Child constructor")

c=C()

output:

Child constructor

Data hiding:

-------------

Outside person can't access our internal data and our data can't go out side directly.

eg:

class Account:

def \_\_init\_\_(self,balance):

self.balance=balance

a=Account(1000)

print(a.balance)

output:

1000

eg:

class Account:

def \_\_init\_\_(self,balance):

self.\_\_balance=balance

a=Account(1000)

print(a.\_\_balance)

output:

Traceback (most recent call last):

File "test.py", line 5, in <module>

print(a.\_\_balance)

AttributeError: 'Account' object has no attribute '\_\_balance'

eg:

class Account:

def \_\_init\_\_(self,balance):

self.\_\_balance=balance

def getbalance(self):

#validations

return self.\_\_balance

a=Account(1000)

print(a.getbalance())

Encapuslation:

----------------

The process of grouping or binding a data into a single unit is known as Encapuslation

Encapuslation=Data hiding+Abstraction

Three pillars of oops:

----------------------

1.Inhertance -------------------->code reusablity

2.Polymorphism-------------------->Flexibilty

3.Encapuslation -------------------->Security

Exception Handling:

--------------------

In any programing language two types of errors are possible

1.syntaxerror

2.Runtime errror or logical error

1.syntaxerror:

--------------

If doing any mistake at coding level such type of errors are known as suntax error

eg:

x=10

if x==10

print(x)

output:

File "test.py", line 2

if x==10

^

SyntaxError: invalid syntax

programer is responsible to correct these syntax errors .once all the syntax errors corrected

then only program is exgiuted.

2.Runtime errror or logical error:

---------------------------------

Exception hanlding concept talks about Runtime errors .

If exciguting a program something goes wrong because of end user input or programming logicc or

memory errors e.t.c that situations errors raised .These errors are known as Runtime errors

or exceptions.

eg:

print(10/0)

output:

Traceback (most recent call last):

File "test.py", line 1, in <module>

print(10/0)

ZeroDivisionError: division by zero

eg:

print(10/"ten")

output:

Traceback (most recent call last):

File "test.py", line 1, in <module>

print(10/"ten")

TypeError: unsupported operand type(s) for /: 'int' and 'str'

what is exception:

------------------

While exiguting the program something goes wrong which is disturbs the normal flow of the program

is nothing bit a exception

eg:

ZeroDivisionError

TypeError

ValueError

file not error

what is need of exception handling:

-----------------------------------

Need of exception handling is normal or gracefull termination of the program

what is the meaning of exception handling:

-------------------------------------------

It means that defining alternative way to continue the rest of the program normally.

try, except , finally

syntax:

try:

---------

-----------

except:

--------------

-------------

finally:

---------------

----------------

try:

Risky code---> the code may be or may not be raise the error

except Name\_of\_exception:

Alternative code

finally:

cleanupcode -------->This block always excuited .It is exicuted error raised or error not raised

or error are not handled all the cases these except block excicute.

eg:

try:

print("Hello")

print("Hello")

print("Hello")

except:

print("Exception Handled")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Hello

Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(10/0)

print("Hello")

except:

print("Exception Handled")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Exception Handled

Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(10/0)

print("Hello")

except ZeroDivisionError:

print("Exception Handled")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Exception Handled

# Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(10/"ten")

print("Hello")

except ZeroDivisionError:

print("Exception Handled")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Finally block exicuted

Traceback (most recent call last):

File "test.py", line 4, in <module>

print(10/"ten")

TypeError: unsupported operand type(s) for /: 'int' and 'str'

eg:

try:

print("Hello")

print("Hello")

print(10/"ten")

print("Hello")

except ZeroDivisionError:

print("Exception Handled ZeroDivisionError")

except TypeError:

print("Exception Handled TypeError")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Exception Handled TypeError

Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(a)

print("Hello")

except ZeroDivisionError:

print("Exception Handled ZeroDivisionError")

except TypeError:

print("Exception Handled TypeError")

except:

print("All the exceptions handled in default except ")

finally:

print("Finally block exicuted")

output:

Hello

Hello

All the exceptions handled in default except

Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(10/0)

print("Hello")

except ZeroDivisionError:

print("Exception Handled ZeroDivisionError")

except TypeError:

print("Exception Handled TypeError")

except:

print("All the exceptions handled in default except ")

finally:

print("Finally block exicuted")

output:

Hello

Hello

Exception Handled ZeroDivisionError

Finally block exicuted

eg:

try:

print("Hello")

print("Hello")

print(10/0)

print("Hello")

except:

print("All the exceptions handled in default except ")

except ZeroDivisionError:

print("Exception Handled ZeroDivisionError")

except TypeError:

print("Exception Handled TypeError")

finally:

print("Finally block exicuted")

output:

File "test.py", line 5

print("Hello")

^

SyntaxError: default 'except:' must be last

eg:

try:

print("Hello")

print("Hello")

print(10/0)

print("Hello")

except:

print("All the exceptions handled in default except ")

else:

print("ELSE Block")

finally:

print("Finally block")

output:

Hello

Hello

All the exceptions handled in default except

Finally block

eg:

try:

print("Hello")

print("Hello")

print("Hello")

except:

print("All the exceptions handled in default except ")

else:

print("ELSE Block")

finally:

print("Finally block")

output:

Hello

Hello

Hello

ELSE Block

Finally block