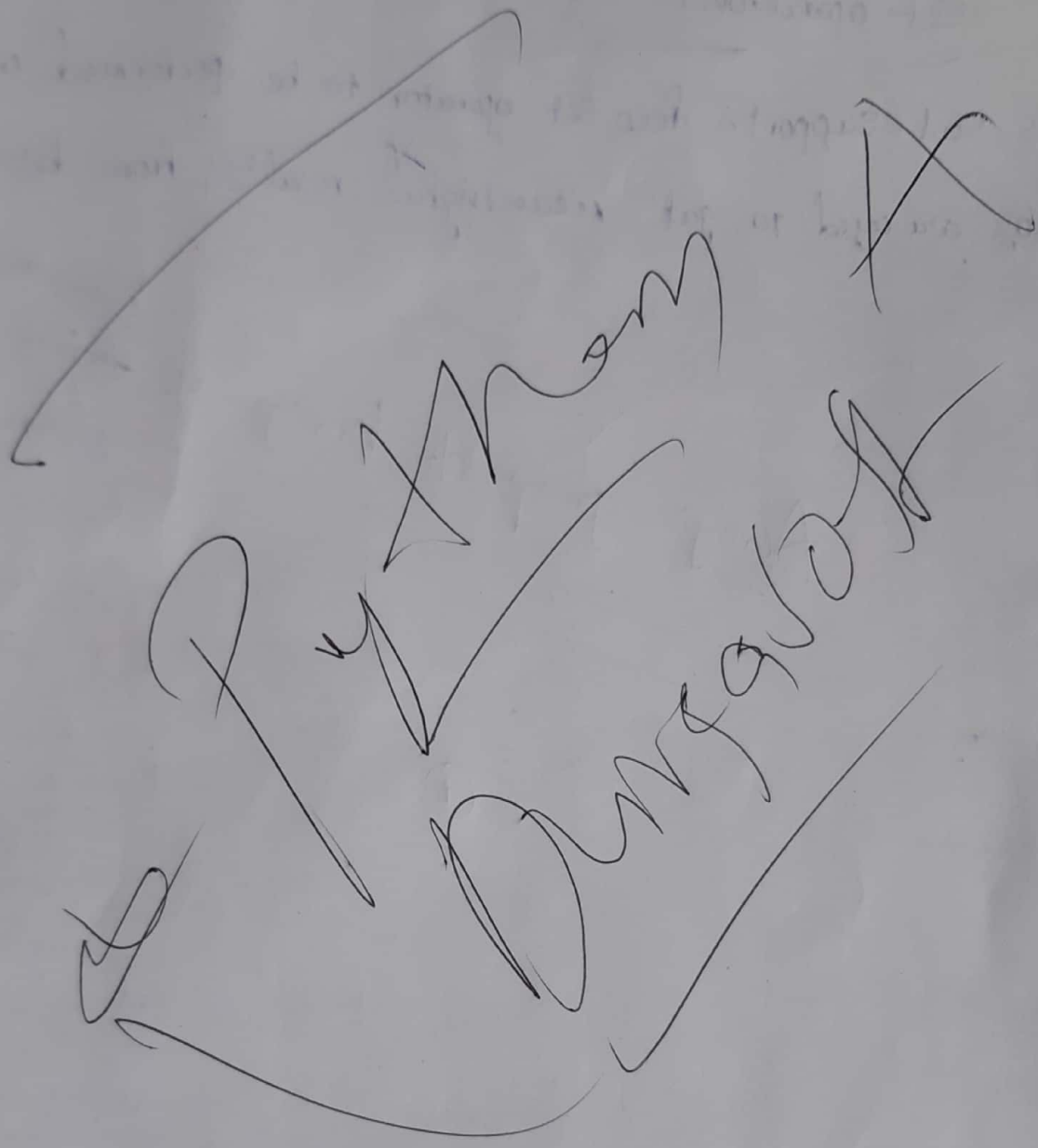
d) what are the set operators? Explain with Example union, intersection and minus.

Ans:- Set operation in sql :-

→ Sql supports few set operator to be performed on table. Those are used to get meaningful results from data,

Array :- group of homogenous ~~Variable~~ elements



Predefined modules

i) math module:

This module is used to do mathematical operations, this module contains several built-in functions

Ex: from math import *

```
Print (sqrt(16)) = 4
Print (ceil(45.6)) = 46
Print (log(4)) =
Print (tan(7)) =
Print (flor(49.9)) = 49
```

(2)

ii) Random module:

This module is used to generate random numbers on fly for authentication

→ This module will have several built-in functions

i) random()
ii) randint()
iii) uniform()
iv) randrange()
v) choice()

i) random()

It will generate random float values blw (0 to 1)(not include).

Ex: from random import *

```
for i in range(5)
    Print(random)..
```

(ii) randint(): It will generate random int values b/w two given numbers (include)

Ex: ⇒ from random import *

```
for i in range(10):
    Print(randint(2,20))
for i in range(20):
    Print(randint(1000,3000))
```

(3)

(iii) uniform(): It will generate random float values b/w two given numbers (not include)

Ex: ⇒ from random import *

```
for i in range(10):
    Print(uniform(2,20))
```

iv) randrange(): It will generate random range values based on start, stop, step values

Ex: ⇒ from random import *

```
for i in range(8):
    Print(randrange(14,30,2))
```

o/p:- 12, 14, 16, 18, 20
22, 24, 26

v) choice(): It will not generate any number, it will random object

Ex: ⇒ lst = ['apple', 'mango', 'grape', 'pineapple']

```
Print(choice(list))
```

** datetime(): This module is used to work with date related task.

Eg: Current system date and time

(1) import datetime
```
x = datetime.datetime.now()
print(x)
```

(2) from datetime import **
```
x = datetime.now(x)
print(x)
```

Eg: from datetime import **
```
x = datetime.now()
print(x)
print(x.year)
print(x.month)
print(x.day)
print(x.hour)
print(x.minute)
print(x.second)
print(x.microsecond)
```

** We can create our own date objects

→ To create date objects we should provide three parameters i.e (year, month, day)

→ All three parameters must be integers and valid

Eg: from datetime import **
```
x = datetime(2020, 2, 3)
print(x)
```

Error

#x = datetime(2019, 2, 29) : day is out of range

#x = datetime(2020, 13, 20) : month is out of range

#x = datetime(2020, 2, 30) : date is out of range.

→ strftime(): This function is used to display date parts in string readable format

Ex: from datetime import *

```
x = datetime.now()
Print(x)
Print(x.strftime("%a"))  :- weekday short version = Thur
Print(x.strftime("%A"))  :- weekday full version = Thursday
Print(x.strftime("%b"))  :- month short version = feb
Print(x.strftime("%B"))  :- month full version = February
Print(x.strftime("%y"))  :- year short version = 20
Print(x.strftime("%Y"))  :- year full version = 2020.
```

* WAP to find difference b/w no of days b/w two given dates *

```
from datetime import date
def f1(date1, date2):
    return(date2-date1).days
date1 = date(2019, 2, 13)
date2 = date(2020, 2, 13)
Print(f1(date1, date2), "days")
```

o/p: 365 days

interval of months & years.

* Calender: This module is used to display the specified month calender (or) whole year calender

Ex: import calender

(6)

m = 2

y = 2020

Print(calender.month(y, m)) -> february 2020 month Calender displayed

Print(calender.month(1947, 8)) - August month Calender will displayed

* whole year calender:

import calender

Print(calender.calender(2020, 2, 1, 4)) (#(year, w, l, c)

w = width of characters

l = no of lines per week

c = Colowmn spacing.

* from calender import *

Print(leapdays(1980, 2020)) -> 10

Print(isleap(2015) = false

Print(isleap(2020)) = True.

* Arrays: It is a collection of homogenous elements is called array

Arrays

→ Array contains homogenous elements

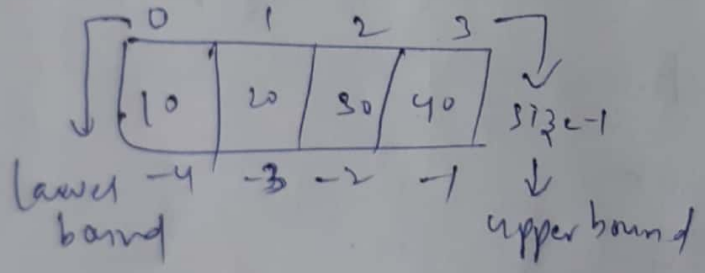
→ In general Arrays can be of two types (7)

i) Single dimension array

ii) Two dimension array.

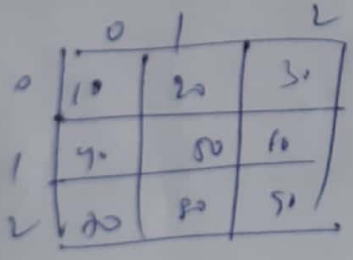
→ An array which contain only one row (or one column is called "single dimension array"

→ Always array index starts with '0' and ends with ('size-1')



→ The array can also has -ve index also

→ An array which contains more than one row and more than one column is called 'two dimensional array'



→ To create arrays in python we use "array" module but using array module we can create only "single dimensional array's"

→ To create single dimensional and two dimension arrays we use "numpy" module

(8)

→ "numpy" stands for numerical python

→ "numpy" is a multi dimensional array processing package

→ "numpy" is not available by default, we need to install explicitly

→ Steps to install numpy in Command prompt :

↳ Go to Command prompt type> pip3 _ install _ numpy

→ steps to install numpy in pycharm

click on file menu, click on 'settings', expand python @ 3pm project, select project interpreter, click on '+' symbol from RHS, type "numpy" in search, select "numpy" module

Then click on install package.

* import array

A = array.array('i', [2, 3, 4, 5, 6, 7])

print (A)

print (A.typecode)

(9)

→ In above example 'i' is a type code which is compulsory

i = integer

f = float

U = unicode character

d = double

I = integer

* Basic methods of array:

from array import *

A = array('i', [23, 45, 67, 89])
indexes: 0, 1, 2, 3 / -4, -3, -2, -1

print (A)

print (A[0]) = 23

print (A[-2]) = 67

print (A[9]) # Array index out of range

A.append (33) [23, 45, 67, 89, 33]

print (A) index

A.insert (2, 44)

index

A.remove (3) => index cannot used number should be placed

A.remove (89)

print (A)

A.reverse (A)

print (A).

* Copying elements from one array to another:-

(1)

```
from array import *
A = array('i', [10, 20, 30])
print (A)
B = A
print (B).
```

(2)

```
B = array(A.typecode, (i*2 for i in A))
Print (B)
```

(10)

o/p: [40, 20, 60]

* print array elements using for loop:

```
from array import *
A = array('i', [4, 5, 6, 7, 8])
print (A)
for i in range (len (A)):
    print (A[i])
```

- (1)

```
for i in A:
    print (i)
```

- (2)

o/p:
4
5
6
7
8

* print character array:

```
from array import *
A = array('u', ['a', 'b', 'c', 'd'])
print (A) = ['a', 'b', 'c', 'd']
for i in range (len (A)):
    print (A[i])
```

o/p:
a
b
c
d.

Program to create array by accepting lengths and elements at run-time

from array import

A = array('i', [])

n = int(input("Enter length of array"))

for i in range(n):

    x = int(input("Enter value"))

    A.append(x)

print(A)

s = int(input("Enter element to search"))

print("Array element present at", A.index(s))

"Numpy" arrays

→ To create numpy arrays we use different methods

i) Array()

ii) linspace()

iii) logspace()

iv) arange()

v) zeros()

vi) ones()

* import numpy
A = numpy.array([2,3,4,5,6,7])
Print(A)
Print(A.ndim) = 1
Print(A.dtype) = int
Print(A.size) = 6 => no. of elements
Print(A.shape) = (6,0)

* import numpy as np
A = np.array([2,3,4,5,6,7,8,9], int)
Print(A)
Print(A.dtype)
A = np.array([3.5, 9.4, 5.6])
Print(A)
Print(A.dtype)

-) Any one of the element is float, all elements turns into floats, Though one of the element is float still to display integer elements we use 'int' datatype

```
from numpy import *
# linspace (start, stop, no. of parts) #
A = linspace (2, 20, 7)
Print(A)
# logspace (start, stop, no. of parts) #
A = logspace (2, 10, 4)
# arange (start, stop, step) #
A = arange (2, 30, 4)
# zeros #
A = zeros (7, int)
# ones #
A = ones (1, ones)
print (A).
```

Two-dimension array :

```
from numpy import *
A = array ([[2, 3, 5],
            [4, 7, 8],
            [7, 1, 3]])
print (A)
```

```
print (A.ndim)
print (A.size)
print (A.shape).
```

* Basic operations on ~~two~~ dimension array to two

→ In two-dimension array dimensions are called "axis".

→ "axis = 0" means Columns

→ "axis = 1" means rows.

* from numpy import *

```
A = array([[-2, 3, 5],
           [4, -9, 8],
           [-9, 1, 3]])
```

axis = 1 = row elements

axis = 0 = Column elements

```
print(A[2, 1])
print(A[2])
print(A[0, 2], A[1, 2], A[2, 2])
print("max element", A.max())
print("min element", A.min())
print("Sum of elements", A.sum())
print("Row wise max element", A.max(axis=1))
print("Row wise min element", A.min(axis=1))
print("Column wise max element", A.max(axis=0))
print("Column wise min element", A.min(axis=0)).
```

* flatten() *

This method is used to collapse all the rows of the two dimension array into single row. (15)

```
from numpy import *
A = array([[-2, 3, 5],
           [4, -7, 8],
           [-7, 1, 3]])
print(A)
print(A.ndim)
B = A.flatten()
print(B)
print(B.ndim)
```

→ It can also perform the arithmatic operations like

(A+B)
(A-B)
(A*B)
(A/B).

A = one array
B = another array

A = array 1
B = array 2.

object oriented programming (OOP's)

→ If a programming language is object oriented, the main advantage is

1. Security
2. Reusability
3. App enhancement

(16)

→ In object oriented programming mainly we need to know about two terms. i) class, ii) object.

i) class: It is a Collection of member Variables and member methods

ii) object: It is an instance of class (or) it is used to represent a class.

Ex: bird is a class

Ex: parrot is an object for bird class

→ Apart from class and object any programming language wants to become as object oriented, that programming language must should satisfied the following features

i) Encapsulation:

ii) Abstraction:

iii) polymorphism:

iv) Inheritance:

i) Encapsulation:- It is the process of providing restrictions to access variables and methods, why we need encapsulation means to prevent the data from modifications. (17)

ii) Abstraction:- hiding the implementation but providing service (or) result.

iii) polymorphism:- poly means "many" and morph means "behaviours".

→ In polymorphism an operation (or) method will shows different behaviour, when we change datatypes of arguments (or) no. of arguments

→ This is of two types

i) static ii) dynamic

→ static polymorphism can achieve with the help of overloading

→ dynamic " " " " " "

overriding.

iv) Inheritance:- It is the process of creating new class from existing class

→ In inheritance process existing class is treated as

Base class, parent class, super class

→ newly created class is treated as Derived, child, subclass

→ In the Inheritance process child class will get all features from parent class

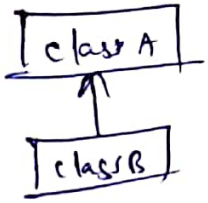
(19)

→ The main purpose of Inheritance is reusability and Extensability.

→ Types of Inheritance:

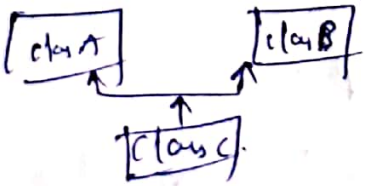
i) single, ii) multiple, iii) multi-level, iv) Hybrid

→ creating new class from single base class is called "Single Inheritance"



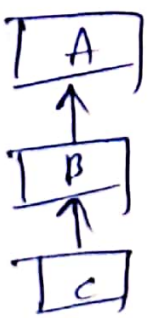
→ class 'B' will get with the help of class-A

→ Creating new class from two (or more base classes is called "multiple Inheritance"



→ class A, B will get with the help of class-C

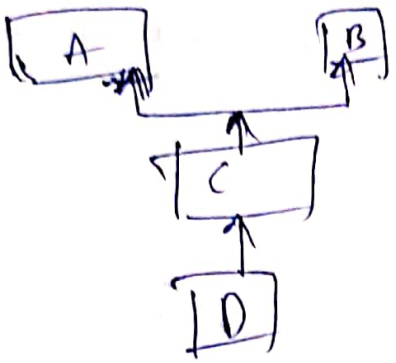
→ Creating new class from already derived class is called "multi-level"



class 'B' gets by class A

class 'C' gets by class B.

→ Hybrid inheritance is the combination of multiple & multi-level.



* How Create class:

class classname:

Ex: class student:

* How to Create object for class:

objectreferencevariable = classname()

Ex: s = student()

Ex: s = student().

** class student: — (P-1)

This * class is creating for display student details

to get descriptions of class

help (student)

Print (student.__doc__).

W.A.P to create student class and display details

```
class student :
    def __init__(self):                    # default argument
        self.sid = 1234
        self.sname = "ram"
        self.saddress = "hyd"
    def display(self):
        print("student id is", self.sid)
        print("student name is", self.sname)
        print("student address is", self.saddress)
s = student()
s1 = student()
s.display()
s1.display()
```

declaring → Constructor

method → printing

o/p displaying forms

→ In the above program '__init__' is a "Constructor".

→ "Constructor" is used to initialize and declare variables.

→ "self" is the default variable and it should be first variable of inside Constructor and method.

→ Using "self" we can access variables and methods inside the class.

→ Constructor will execute automatically, when we create object to class (21)

→ we can create any no of objects to class.

→ for every object Constructor will execute ones.

→ In above program 's' and 's1' are objects.

→ In above programs for 's' and 's1' objects same Constructor is going to execute so, that we get two times same student details.

→ To avoid this, for every object to display different student details, then we use "parameterized Constructor".

~~class~~ class student: (P-2)

```
    def __init__(self, sid, sname, saddress):
        self.sid = sid
        self.sname = sname
        self.saddress = saddress
    def display(self):
        print(self.sid, self.sname, self.saddress)
s = student(101, "sai", "hyd")
s1 = student(102, "hari", "sr nagar")
s.display()
s1.display()
```

→ (p-3) ✱

→ we can use the diffent Variables instead of sid, sname, saddress as arguments

✱✱✱ class student :

```
    def __init__(self, x, y, z):
        self.sid = x
        self.sname = y
        self.saddress = z
    def (display)(self):
        print (sid=self, self.sname, self.saddress)
s = student (101, 'Sai', 'Hyd')
s1 = student (102, 'Anaj', 'Gnt')
s.display()
s1.display().
```

→ Example to get the details without method ✱

Ex: class student :

```
    def __init__(self, sid, sname, saddress):
        self.sid = sid
        self.sname = sname
        self.saddress = saddress
s = student (101, 'Sai', 'Hyd')
s1 = student ('102', 'Anaj', 'Gnt')
```

display.

```
print (s.__dict__)
print (s1.__dict__).
```

→ without method display the values by help of Constructor only


```
class student :
    def __init__(self, x, y, z):
        self.sid = x
        self.sname = y
        self.saddress = z
        print(self.sid, self.sname, self.saddress)
s = student(101, 'Arun', 'Gut'):
```

→ Using runtime and without any Constructor

```
class student :
    def getdata(self):
        self.sid = int(input("Enter sid"))
        self.sname = input("Enter sname")
        self.saddress = input("Enter saddress")
    def displaydata(self):
        print("sid is {}, sname is {}, saddress is {}".
              format(self.sid, self.sname, self.saddress))
s = student()
s.getdata()
s.displaydata()
```

→ difference b/w Constructor and method :-

Constructor	method
→ Constructor name should be __init__	→ method name can be of any name.
→ Constructor will Execute automatically when we Create object to class	→ It Execute only when we Call that.
→ for every object Constructor Executes ~~one~~ ones	→ for every object method can Execute any no of time.
→ Inside Constructor we can declare Variables and initialize the data to the Variables	→ Inside method we can write business logic Code.

~~Types~~ Types of Variables : ~~A to Z~~

→ Three types of Variables are allowed in python class

i) instance Variable (object level)
ii) static Variable (class level)
iii) Local Variables (method level).

1. Instance Variables (object level) :-

If the value of variable is varied from object to object, then such type of variables called instance variables

→ Where we can declare instance variables:-

1. inside the constructor by using "self."
2. inside the method by using "self."
3. outside the class by using "object reference variable"

Eg:

```
class Test:
    # inside the Constructor by using self #
    def __init__(self):
        self.a = 10
        self.b = 20
    # inside the method by using self #
    def m1(self):
        self.c = 30
t = Test()  → class calling
t.m1()  → method calling.
# outside the class by using object reference #
t.d = 40
print(t.__dict__).
```

→ How to access instance Variable :

1. we can access instance Variable by using 'self' with in the class

2. " " " " " outside of class by using 'object reference'.

declaring = initialize the Variable

accessing = printing the Variable o/p.

Ex:

```
class Test:
    def __init__(self):
        self.a = 10
        self.b = 20
        self.c = 30
    # accessing the Variable within class #
    def m1(self):
        print(self.a, self.b, self.c).
t = Test() → class calling
t.m1() → method calling.
# accessing the Variables outside by using object reference #
print(t.a, t.b, t.c).
```

→ How to delete instance Variable :

→ we can delete instance Variable "with in class" as follows

(27)

* del self.Variablename

→ we can delete instance Variable "outside of the class" as follows

* del objectreference.Variablename.

Ex:

```
class Test:
    def __init__(self):
        self.a = 10
        self.b = 20
        self.c = 30
        self.d = 40
    # delete with in class #
    def m1(self):
        del self.a
t = Test()  → class calling
print(t.__dict__)
t.m1()  → method calling.
print(t.__dict__)
    # delete outside the class #
del t.b
print(t.__dict__).
```

Note: For every object a seperate copy of instance Variables will be Create, if we delete (or) modify one copy of instance Variables that will not effect other copy of instance Variables.

Eg:
```
class Test:
    def __init__(self):
        self.a = 10
        self.b = 20
        self.c = 30
        self.d = 40

t1 = Test()
t2 = Test()
```

Print(t1.__dict__) = (a: 10, b: 20, c: 30, d: 40) = t1

Print(t2.__dict__) = (a: 10, b: 20, c: 30, d: 40) = t2

del t1.a = (b: 20, c: 30, d: 40) = t1

t2.b = 99 = (a: 10, b: 99, c: 30, d: 40) = t2

Print(t1.__dict__) = (a: 20, b: 30, c: 40) = t1

Print(t2.__dict__) = (a: 10, b: 99, c: 30, d: 40) = t2.

2. Static Variable (Class Variable):

If the value of Variable is not varied from object to object then such type of variables we should declare directly inside the class but outside of the methods, these types of variables are called static variables.

(99)

→ for all objects only one copy of static variables will be create, Then that copy is shared to all objects

→ we can access static variable by using "classname" (or) by using "objectreference"; but recommend to use is "classname"

(static = global variable).

```
class Test:
    a = 10  # static Variable #
    def __init__(self):
        self.b = 20  # instance Variable #.

t1 = Test()
t2 = Test()                    (o/p)
print(t1.a, t1.b) = (10, 20)
print(t2.a, t2.b) = (10, 20)
Test.a = 99 (# changing static Variables from a=10, to a=99#)
t1.b = 88
print(t1.a, t1.b) = (99, 88)
print(t2.a, t2.b) = (99, 20).
```

3. Local Variables (method level):

The Variables which are declared directly inside the method and that Variables are available to only that method is called local Variables.

→ local Variables can't be access outside of method

→ local Variables also called as temporary Variables, because local Variables are available, when the method Execute, one's method Execution Completes, local Variable will been destroyed:

Ex:

```
class Test:
    def m1(self):
        a=10
        print(a)
        print(b) # 'b' is not defined #
    def m2(self):
        b=20
        print(b)
        print(a) # 'a' is not defined #

t = Test()
t.m1()
t.m2()
```

Types of methods

→ Three types of methods are allowed to python They are:

i) Instance method
ii) class method
iii) static method.

i) Instance methods :

→ Inside method implementation, when we use instance Variables, Then such type of methods are Called instance methods.

→ while declaring instance methods we should pass "self".

→ Using "self" we can access instance Variables with in the instance methods

→ We can access instance methods within the class by using "self".

We can access " " outside the class by " "object reference".

Eg: class Student :

```
    def __init__(self, m1, m2, m3):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
    # instance method #
    def avg(self):
        return (self.m1 + self.m2 + self.m3)/3
s1 = Student (67, 52, 43)
s2 = Student (58, 49, 43)
print(s1.avg())
print(s2.avg())
```

(32)

o/p:
63.333
72.443

— (1)

(or)

```
def avg(self):
    print((self.m1 + self.m2 + self.m3)/3)
s1 = Student (67, 52, 43)
s2 = Student (58, 49, 43)
s1.avg()
s2.avg()
```

— (2)

ii) class methods :-

→ Inside method implementation when we use only static (or) class level Variables then such type of methods are called class methods.

→ while declaring class methods we should pass class Variable i.e "cls"

→ Using "cls" we can access static Variables with in class

→ We can access class methods by using "classname" (or) "object reference".

→ To make a method as class method we use "@classmethod" decorator.

Note :- class methods are Very rarely used methods in Python language.

Ex :-
```
class Student:
    inst = "durgasoft"  # static Variables #
    def __init__(self, m1, m2, m3):
        self.m1 = m1
        self.m2 = m2
        self.m3 = m3
    # instance method #
    def avg(self):
```

return (self.m1 + self.m2 + self.m3) / 3

class method

@classmethod
def m1(cls):
    return cls.inst

s1 = student(67, 89, 46)
s2 = student(75, 90, 84)

Print(s1.avg()) = 63.33
Print(s2.avg()) = 78.44
Print(student.m1()) : durgasoft

iii) static methods :

These methods are general utility methods.

→ while declaring static methods we should not pass any self, cls Variables.

→ To make a method as static you will use "@staticmethod" decorator.

→ we can access static method either by using "classname" (or) "object reference."

Ex: :

```
class Test:
    @staticmethod
    def add(a,b):
        Print("Sum is", a+b).
```

```
@staticmethod
def sub(a,b)
    print("sub is", a-b).
@staticmethod
def fi():
    print("hello world")
```

```
t=Test()                    o/p:
t.add(20,30)    = 50
t.sub(10,5)     = 5
t.fi()          = helloworld.
```

Task:-1 In which places we can accessing static Variables

Task:2 How to delete static Variables

Inner class

without existing one type of object, ther is no chance of existing another type of object, then we should go for inner class

Syntax:

```
class Car:
    class Engine:
```

```
class Head:
    class Brain:
```

Ex:-1

```
class outer:
    def __init__(self):
        print("outer class Constructor")

    class Inner:
        def __init__(self):
            print("Inner class Constructor")

        def m1(self):
            print("Inner class method")

o = outer()  — Cons-①
i = o.Inner()  — Cons-②
i.m1().  → 'method'
```

(36)

(or)

```
o = outer()
i = o.Inner()
t = i.m1()
```

|

```
i = outer().Inner()
i.m1()
```

(or)

```
outer().Inner().m1()
```

o/p:

```
outer class Constructor
Inner class Constructor
Inner class method.
```

All three methods in one program

Instance + class method + static method.


```
class Test:
    profile = "amarooska"   # class method # declaration
    def __init__(self):
        self.a = 20
        self.b = 30
        self.c = 40
    def avg(self):   # instance method #
        print((self.a + self.b + self.c)/3)
    @classmethod
    def m1(cls):
        return cls.profile
    @staticmethod   # static method #
    def add(a,b):
        print("sum is", a+b)
    @staticmethod
    def sub(a,b):
        print("sub is", a-b)
```

Instance method

class method

static method

```
t = Test()
t.avg()
print(t.avg())
print(Test.m1())
```

```
t.add(50, 70)
t.sub(50, 30)
```

o/p:

```
30.00= avg
amarooska = classname
sum is 120
sub is 20.
```

* Task-1: where we use and access static Variables :-

→ we can access static Variables inside the Constructor by using self (or) classname.

→ Inside instance method by using self (or) classname.

→ Inside class method by using cls (or) classname.

→ outside of the class by using classname.

* Task-2: How to delete static Variables :-

we can delete static Variable in any place by using the following syntax:

"del classname.Variablename"

→ within the class method like

"del cls.Variablename."

```
class Test:
    a=10
    @classmethod
    def m1(cls):
        del cls.a
Test.m1()
print(Test.__dict__)
```

* * How to access one class members into other class * *

```
class Employee:
    def __init__(self,eno,ename,
                 esal):
        self.eno = eno
        self.ename= ename
        self.esal = esal
    def display(self):
        print(self.eno,self.ename,
              self.esal)
class Test:
    def modify(self):
```

```
        self.esal = self.esal + 4000
        self.display()
e = Employee(101, 'sai', 10000)
Test.modify(e).
```

o/p:
=> 101, sai, 14000.

(39)

→ In the above example 'esal' is a member of employee class, 'esal' is available to the "test class" for required modification.

** Gc (Garbage Collector) :- **

→ Gc it is used to provide automatic memory management for application objects

→ automatic memory management is the process of allocate the memory for useful objects and deallocate the memory for unused objects

→ By default in python 'Gc' is enabled

→ Gc will call internally destructor for cleanup activities

→ 'Destructor' is a special method and the name is ["__del__"]

→ Before destroying unused object garbage Collector will call destructor to perform Cleanup activities.

→ After Completion of destructor Execution, then Gc will destroy that objects.

→ So, that destructor is not for destroy the objects, it is only for cleanup activity.

(10)

i) isenabled() → if enabled return True else False

ii) disable() → to disable GC explicitly

iii) enable() → to enable " ".

Ex: import gc

o/p

print(gc.isenabled()) - True

gc.disable()

print(gc.isenabled()) - False

gc.enable()

print(gc.isenabled()) = True.

Ex: import time

class Test:

def __init__(self): → "Constructor"

print("object initialization")

def __del__(self): → "Destructor".

print("cleanup activities")

t1 = Test()

time.sleep(5)

t1 = None

print("end of application")

Access specifiers :-

→ In python their is no access specifiers, like private, public, protected, we can only identify them with "__" underscore

X = 10 = public

-Y = 20 = protected

--Z = 30 = private.

→ public Variables don't have restrictions, we can access anywhere of program

→ protected Variables can assess with in the same class and also in derived class

→ private Variables can access only within the same class

Ex:

```
class Test:
    X = 10  → public
    -Y = 20 → protected
    --Z = 30 → private
    def __init__(self):
        print(self.X)
        print(self._Y)
        print(self.__Z)

t = Test()
t.m1()

#outside the class#
print(t.X)
print(t._Y)
print(t.__Z) #not access because private Variable
```

Oops methods ~~to do~~ Important Concept begins

i) Encapsulation :-

It is the process of providing restrictions to access (42)
Variables and methods

→ Why we need encapsulation to prevent the data from the modifications (or) access.

Ex: with private method:

```
class Car:
    def __init__(self):
        self.__updatesoftware()
    def __updatesoftware(self):
        print("updating software")

c = Car()
```

Variable name # ⇒ instance with private

Ex: No Constructor:

```
class Car:
    def __updatesoftware():  # private method #
        print("updating software")

c = Car()
c.__updatesoftware()  # attribute error because
```

private method cannot access outside of class #

SM: with private Variables:


```
class Car:
    --name = " "
    --maxspeed = "0"
    def __init__(self):
        self.__name = "Ford Figo"
        self.__maxspeed = "100"
        Print(self.__name, self.__maxspeed)
    def drive(self):
        Print(self.__name, self.__maxspeed)
c = Car()
c.drive()
c.__maxspeed = 200  # private Variable Cannot change
                    outside of the class #)
c.drive()
```

⇒ maxspeed = private Variable; that Cannot be accessed modify outside of the class.

Ex:- updating the maxspeed :-

Encapsulation


```
class Car:
    __name = ""
    __maxspeed = 0
    def __init__(self):
        self.__name = "ford figo"
        self.__maxspeed = 100
        print(self.__name, self.__maxspeed)
    def setspeed(self, speed):
        self.__maxspeed = speed
        print(self.__maxspeed)

c = Car()
c.setspeed(200)
```

=> In the above example, though maxspeed is a private variable, we can modify maxspeed through public method [i.e setspeed()] outside of the class.

ii) polymorphism

→ poly = many and morph means behaviour

→ In polymorphism, an operator (or method will shows different behaviour, when we change datatypes of the arguments (or) no. of arguments.

→ It is of two types :-

i) static

ii) Dynamic

→ By static polymorphism we can achieve "overloading"

→ By Dynamic " " " " "overriding".

i) overloading :-	ii) overriding :-
1. operator overloading - Yes	method overriding - Yes
2. method overloading - No	Constructor overriding - Yes
3. Constructor overloading - No	

i) operator overloading :-

we can use '+' operator for arithmatic additions and Concatenations

→ we can also use '+' operator for our class-objects

Ex: class Book: — (1)

```
    def __init__(self, pages):
        self.pages = pages

b1 = Book()
b2 = Book()
print(b1+b2)  # type error #
```

→ when we execute the above program it shows "type error".

→ To overload '+' operator for 'Book'-objects, we have to override "magic method" into our class.

→ In python every operator is having magic methods :-

+ → __add__
- → __sub__
* → __mul__
/ → __div__

Ex: class Book: — (2)

```
    def __init__(self, pages):
        self.pages = pages
    def __add__(self, other):
        return self.pages + other.pages

b1 = Book(10)
b2 = Book(20)
print(b1+b2)
```

o/p: 30

→ Here 'other' is default variable like 'self'.

ii) method overloading :

If we have multiple methods with same name with different types of parameters, then that method is said to be overload method

→ Python dont support method overloading, because when we have multiple methods with same name the last method will only will be considered.

Ex:
```
class Test:
    def m1(self):
        print("no arg method")
    def m1(self,a):
        print("one arg method")
    def m1(self,a,b):
        print("two arg method")
t=Test()
# t.m1() → missing positional arguments a,b
# t.m1(10) → missing positional argument b
t.m1(10,20)
```
o/p: two arg method

iii) Constructor overloading:

→ python Cannot Support Constructor overloading because, when we have multiple Constructor with different arguments, last Constructor only will been Considered (48)

Ex!
```
class Test:
    def __init__(self):
        print("no arg constructor")
    def __init__(self,a):
        print("one arg Constructor")
    def __init__(self,a,b):
        print("two arg Constructor")
```

errors
```
#t = Test()
#t = Test(a)
```
} because positional arguments are missing.

t = Test(a,b) → o/p: two arg Constructor

2) (i) Method overriding:

→ what ever the members are available in parent class, that are by default available to the child class through inheritance

→ If the child class not satisfy with the parent class implementation, child class can Redefine the parent class methods with in it., this process is called overriding.

Ex!

```
class Parent:
    def property(self):
        print("cash+gold+power+lands")
    def car(self):
        print("alto")
class Child(Parent):
    def car(self):
        super().car()
        print("benz")
c = Child()
c.property()
c.car()
```

(19)

o/p:

```
cash+gold+power+land
alto
benz
```

→ If child class wants to access parent class method also, then we use "super()" method in the child class functions.

ii) Constructor overriding:-

```
class Parent:
    def __init__(self):
        print("Parent Constructor")
class Child(Parent):
    def __init__(self):
        super().__init__()
```

Print ("child class Constructor")

c = Child()

o/p: parent class Constructor
child class Constructor

(50)

→ Because we use the "super()" method.

iii) Inheritance :-

→ Inheritence is the process of creating new class from exisisting class

→ The advantage of inheritence is reusability and extensability.

Syntax:

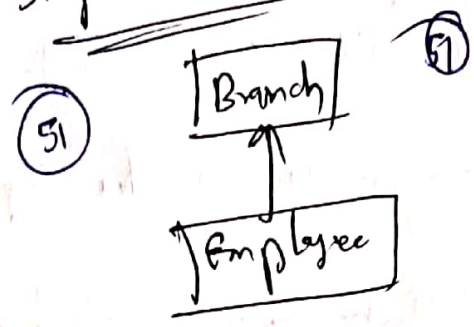
```
class Baseclass :
    Body of baseclass
class Derivedclass(Baseclass):
    Body of derivedclass
```

→ Assume that if a Company is asking to Computerized their branch details, then we create branch class and implement Then we will get branch details

→ If another Company is asking to Computerized their employee details, Then we create employee class and implement, so that we will get employee details.

→ Assume that if Company is asking to know which employee belongs to which branch, then we should do inheritance.

Single Inheritance:



Program for single inheritance:

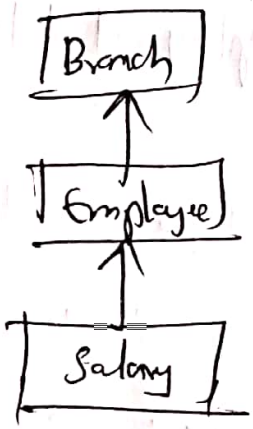
```
class Branch:
    def getbdata(self):
        self.bcode = int(input("Enter branch code"))
        self.bname = input("Enter branch name")
        self.baddress = (input("Enter branch address")
    def displaybdata(self):
        print("branch details are")
        print(self.bcode, self.bname, self.baddress).
class Employee(Branch):
    def getempdata(self):
        self.empid = int(input("Enter empid"))
        self.ename = input("Enter ename")
        self.eaddress = input("Enter eaddress")
    def displayempdata(self):
        print("emp details are")
```

print(self.empid, self.ename, self.eaddress)

e = Employee()
e.getbdata()
e.getempdata()
e.getdisplaybdata()
e.displayempdata().

→ later Company is asking to Computerized their employee salary details and they want to know which employee belongs to which branch, then we should Extend Salary class from employee class.

multi-level inheritance



* program for multilevel inheritance :-

```
class Branch:
    ==========
class Employee(Branch):
    ==========
class Salary(Employee):
    def getsal(self):
        self.basic = int(input("Enter basic salary"))
    def calculate(self):
        self.DA = self.basic * 0.06
        self.HRA = self.basic * 0.7
```

self.Gross = self.basic + self.DA + self.HRA

def displaysal(self):

Print("Emp salary details")

Print(self.basic, self.DA, self.HRA, self.Gross)

s = Salary()

s.getbdata()

s.getempdata()

s.getsaldata() → s.calculate()

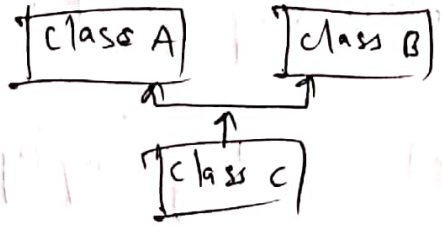
s.displaybdata()

s.displayempdata()

s.displaysaldata().

iii) multiple Inheritance:

It is the process of creating new class from two (or) more base class



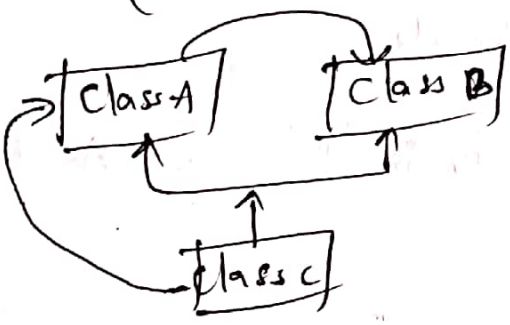
Ex:

```
class A:
    def f1(self):
        print("f1 function of class A")

class B:
    def f2(self):
        print("f2 function of class B")
```

```
class C (A, B):
    def f3(self):
        Print("f3 is a function of class C")

c = C()
c.f1()
c.f2()
c.f3().
```


→ If base classes are having same function i.e (f1), then according to MRO (Method Resolution order) rule, only one f1() function will be called.

MRO (Method Resolution order)



→ clockwise direction

Ex:

```
class A:
    def f1(self):
        Print("f1 function of class A")

class B:
    def f1(self):
        Print("f1 function of class B")

class C(A, B):
    def f2(self):
        # B.f1(self)
        Print("f2 function of class C")
```

```
c = C()
c.f1()
c.f2()
```

i)) Constructor rule in single inheritance :-

```
class A:
    def __init__(self):
        Print("A class Constructor")

class B(A):
    def __init__(self):
        Print("B class Constructor")

b = B()
```

o/p:- B class Constructor

→ In the above example we created object for class-B, so that 'B' class Constructor will execute

→ If their is no Constructor in "B", Then A-class Constructor will execute.

ii) Constructor rule in multi-level Inheritance :-

```
class A:
    def __init__(self):
        Print("A class Constructor")

class B(A):
    def __init__(self):
        Print("B class Constructor")

Class C(B):
    def __init__(self):
        Print("C class Constructor")

c = C()
```

o/p:-

"C class Constructor"

→ In the above example, we created object for class-C, class-C Constructor will Execute

→ If their is no Constructor in class-C, then class-B will been Execute

→ If their is no Constructor in class-B, then class-A Constructor will been Executed

iii) Constructor role in multiple inheritance

```
class A:
    def __init__(self):
        print("A-class Constructor")
class B:
    def __init__(self):
        print("B-class Constructor")
class C(A,B):
    def __init__(self):
        print("C-class Constructor")
c = C()
```

→ In the above example, if their is no Constructor in class-C acc to MRO rule, class-A Constructer will Execute, if their is no constructer in "A", then class-B executed.

iv) Abstraction :-

1) Abstract method :-

1. A method which doesn't Contain any implementation or body, which contain only declaration is Called Abstract method
2. In python to make a method as abstract we use an @abstractmethod decorator
3. To leave abstract method without implementation we use 'pass' as keyword.
4. @abstractmethod is present "abc module"
5. abc module is abstract base class module
6. abstract methods are Compulsory to implement into derived class.

2) Abstract class :-

1. A class which Contains one (or) more abstract methods is Called abstract class
2. abstract class is also Contain non-abstract methods
3. to make a class as abstract, our class should extend from ABC predefined abstract class
4. ABC is present in "abcmodule"
5. Abstract class cannot instantiated directly, so that we need to create (or) derive new class from abstract

class to provide functionality to its abstract methods.
& we cannot ~~reject~~ create object for abstract class, because it is partially implemented.

Ex:-
```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def Wheels(self):
        Pass (or) None
    @abstractmethod
    def Colour(self):
        Pass (or) None
    def engine(self): → # Non-abstract method
        Print("BS-4 engine")
class Car(Vehicle):
    def wheels(self):
        Print("Car: 4 wheels")
    def Colour(self):
        Print("Car Colour is Red")
class Bike(Vehicle):
    def wheels(self):
        Print("Bike: 2 wheels")
```

```
    def Colour (self):
        Print ("Bike Colour is Black")
class Auto (Vehicle):
    def wheels (self):
        Print ("Auto: 3 wheels")
    def Colour (self):
        Print ("Auto Colour is Yellow")
```



```
c = Car()
c.wheels()
c.engine()
c.Colour()
```
} object for Car

```
b = Bike()
b.wheels()
b.engine()
b.Colour()
```
} object for Bike

```
a = Auto()
a.wheels()
a.engine()
a.Colour()
```
} Object for Auto

→ An abstract-class how many no-of abstract methods are their, that all methods Compulsory to implement in every derived class of abstract class, otherwise

derived class will become abstract-class

→ abstract class means partial implementation of class

→ A-class which is having fully implementation methods is called "Concrete class"

→ A class which is having all abstract methods then it is called "Interface class"

60

```
from abc import *
class collegeautomation(ABC):
    @abstractmethod
    def m1(self): pass
    @abstractmethod
    def m2(self): pass
    @abstractmethod
    def m3(self): pass
class sample(collegeautomation):
    def m1(self):
        print("m1 implementation")
    def m2(self):
        print("m2 implementation")
class sample1(sample):
    def m3(self):
        print("m3 implementation")
s=sample1
s.m1()
s.m2()
s.m3()
```

interface class

Concrete class.

* accessing the private variable outside of class :-


```
class Test:
    def __init__(self):
        self.__x = 10   # private variable
t = Test()
print(t._Test__x) = 10
```

=

* * * Exception Handling :- * * * *

i) Syntax errors :- If syntax is wrong

ii) Runtime error :- while executing time

* Exceptions :- An unwanted (or unexpected) reason, that will disturbs normal flow of execution is called "exception".

Ex:- ZeroDivisionError
TypeError
ValueError
FileNotFound
EndOfTheFileError
NameError.

→ It is not repairing the error it is an alternate way to execute the program that is called "Exception handling".

→ There are two Exception-handling models:

i) logical implementation

ii) try except implementation.

i) logical implementation :-

```
a = int(input("Enter num1"))
b = int(input("Enter num2"))
if b==0:
    Print("second num Can't be zero")
else:
    c = a/b
    Print(c)
```

o/p: Enter num1 = 20
Enter num2 = 0
Second num Can't be zero

ii) try Except implementation

Syntax:

```
try:
    statements
except:
    statements
```

→ First Control jumps to try block if there is an error then it moves to except block (or) try block will Execute

Ex1:

```
try:
    a = int(input("Enter num1"))
    b = int(input("Enter num2"))
    c = a/b
    Print(c)
except:
    Print("Error is occured").
```

① i/p: Enter num1 = 20
Enter num2 = 0
Error is occured

② Enter num1 = 10
Enter num2 = 5
= 2

③ Enter num1 = Ynamar (string)
error is occured

→ we can use two (or) more except blocks in one single try block

Ex1:

```
try:
    a = int(input("Enter num1"))
    b = int(input("Enter num2"))
    c = a/b
    Print(c)
Except: ZeroDivisionError as msg:
    Print(msg)
Except ValueError as msg:
    Print(msg)
Except NameError as msg:
    Print(msg).
```

Single try block

multiple Except block

Ex:

```
try:
    a = int(input("Enter num1"))
    b = int(input("Enter num2"))
    c = a/b
    Print(c)
except (ZeroDivisionError, ValueError, NameError) as msg:
    Print(msg)
```


→ Except zerodivisionerror and nameerror error is occured in given program:

Ex:

```
try:
    a = int(input("Enter num1"))
    b = int(input("Enter num2"))
    c = a/b
    Print(c)
except ZeroDivisionError as msg:
    Print(msg)
except:
    Print("Error is occured")
```

} default except block

→ Always default except block should be last of the Program.

* Control flow in try-except + * *

```
try :
    statement-1
    statement-2
    statement-3          } Risky Code
except msg :
    statement-4          } handling Code
statement-5 → finally Code
```


i) Case (i): if there is no Error in try block then 1, 2, 3, 5 statements are Executed

ii) Case (ii): if error raised at statement-2 and Corresponding except block matched, then 1, 4, 5, are Executed

iii) Case (iii): if Error raised at statement-2, and Corresponding except block are not matched then statement-4 will only Executed (or) abnormal termination

iv) Case (iv): if Error is raised in statement (4) (or) (5) then it is always abnormal termination.

finally :-

If Their is an Exception raised (or not but user should want some type of output then finally block will been executed → It is an optional, if you included it must be executed.

Ex: if Their is no Exceptions :

```
try:
    print("try")
except:
    print("except")
finally:
    print("finally")
```

o/p: try
finally

Ex: if Their is an Exception raised but handled :-

```
try:
    Print("try")
    Print("1/0") → error
except:
    print(except)
finally:
    Print("finally")
```

o/p: except
finally

```
try:
    Risky code
except:
    handling code
finally:
    mandatory msg.
```

syntax of finally block.

*) Their is only one situation to stop finally block Executing i.e Can we use "os._exit(0)"

→ when we use this function (pvm) itself shutdown

Ex:- import os

```
try:
    print("try block")
    os._exit(0)
except:
    print("except block")
finally
    print("finally block")
```

o/p:- try → after try it will exit the loop.

**) Nested try, except, finally blocks:-

) If a try (or) except (or) finally is having one more try, except, finally blocks is called nested try, except, finally.

Syntax:

```
try:
    statements
    try:
        statements
    except:
        statements
    finally:
        statements
except:
    statements
finally:
    statements
```

(nested try block)

Ex:

```
try:
    # print(1/0) #
    print("outer try")
    try:
        # print(10/0) #
        print("inner try")
    except ZeroDivisionError:
        print("Inner except")
    finally:
        print("Inner finally")
except:
    print("outer except")
finally:
    print("outer finally")
```


Note :-

→ If exception raised inside the inner try, inner except block should take responsible handled, if inner except block is not able to handled, then outer except block should take responsible.

Types of Exceptions :-

i) Userdefined Exceptions
ii) pre-defined Exceptions

→ Predefined Exceptions are by default available
→ The Exceptions which are created by programmer, which is used to raise according to their requirement is called, userdefined Exceptions (or) Customized Exceptions
→ To create Customized Exception, our class should extend from predefined Exceptions class.

Syntax :-

```
class classname (predefined Exceptions class):
    def __init__(self, arg):
        self.msg = arg
```

Ex:

```
class tooOldException(Exception):
    def __init(self, arg):
        self.msg = arg
```

Ex-1 Raising of Error

```
class toOldException(Exception):
    def __init__(self,arg):
        self.msg = arg
class tooYoungException(Exception):
    def __init__(self, arg):
        self.msg = arg
```

```
Age = int(input("Enter your age"))
if Age >= 60:
    raise toOldException("You are old")
elif Age > 16:
    raise tooyoungException("you are young")
else:
    Print("you are elisible to take policy").
```

Ex:- handling of Errors:-

```
try:
  Age=int(input("Enter your age"))
  if Age >= 60:
      raise toooldException("You are old")
  elif Age <=16:
      raise tooyoung exception("You are young")
  else:
      Print("You are elisible to tulee policy")
except(tooyangexception, toold exception, ValueError) as msg:
      Print(msg)
```

File handling in python

→ Files are very commonly to store the data permanently in our disk locations

→ In general their are two types of files (71)

i) Text files

ii) Binary files

→ Text files are used to store character data, extention is the ".txt"

→ Binary files are used to store images, audio and video etc

→ To Create a file we use built-in method i.e "open()"

→ while creating file we have to give "filename, file mode"

Syntax:

file object = Open("filename", "filemode")

f = Open("Sample.txt", "w")

→ In general files can been used for write, read operations

→ "write", means to store the data into file

→ "Read", means to retrive the data from file

→ After Completion of read & write operations, we should close file by using "close()" method,

"f.close()"

i)

ii)

iii)

iv)

v)

Ex

File properties:

i) name: name of the opened file

ii) mode: mode in which file it is opened

iii) Readable(): True/False

iv) Writable(): True/False

v) Closed() : True/False

Ex:- f = open("sample.txt", "w")

print("file name is:", f.name) = sample.txt

print("file mode is:", f.mode) = "w"

print("is file readable:", f.readable()) = False

print("Is file writable:", f.writable()) = True

print("Is file closed:", f.closed) = False

f.close()

print("Is file closed", f.close) = True

→ To write the data into file we use the following methods

i) write(): to write string data

ii) writelines(): to write list of lines

Ex:-1

```
f= open("xyz.txt", 'w')
f.write("Hello\n")
f.write("Durgasoft\n")
f.write("hyderabad")
print("Data written to the file")
f.close()
```

(23)

Ex:-2

```
f= open("sample.txt","w")
list=["hai\n","hello\n","how are you"]
f.writelines(list)
print("List data written to file")
f.close()
```