--abc--=1 → magic method (AUM JVM)
--abc=100 → strong

→ where are using the JAVA & python program
1) Text PAD, Text PAD++, edit+
2) IDE
3) pycharm → python, JAVA → www.jetbrains.com
4) vs code - c, c++
5) Anaconda
6) Jupiter
7) Google Colabs

download pycharm
community → free → to exocta pgm
professional → license → to develop project

Reserved words
→ the words which are reserved by python interpedia to & pvm
→ these are pre defined mean
→ In python we have 33 reserved words -

pgm
import keywords
print (keywords.kwlist)

Datatypes
→ Datatype specifies type of data & size of data
→ python supports following datatypes.
1) int
2) float
3) complex
4) bool
5) str
6) bytes → concrit data type
7) list
8) tuple
9) set
10) dict
11) frozenset
12) None.

→ small set of characters in a pgm is called tokens
these are 5 tokens
1) operators
2) keywords
3) Identifiers
4) special symbols -) ;,
5) constants → #define

1) keywords
2) identifiers
3) literals
4) operators
5) punctuators → ;,

type( ).

→ This function is useful to find out datatype of a given variable.

Eg : i = "python"
print (type(i))

o/p <<class 'str'>>

→ In python str can be enclosed in either in " " or in ' '

→ In python we don't have char datatype

Output function in python

⇒ print function is output function in python

syntax :- print ("welcome to python")

o/p :- welcome to python

pgm :-
```
j = 10
i = 20
k = 30
print (i, j, k, sep = ", ")
print (i, end = ".")
print (j)
```

* In python "*" performs two operations
i) multiplication ii) seperator after 1 argument is number & other is string

pgm = print (10*"Hello")

⇒ Customize pgms

```
s = "ira"
age = 10
print ("Hello", s, "my age is", age, "years")
```

→ print with formated strings

%d = integer
%i = integer
%f = float
%s = string

syntax :- print ("formated string", % (variable(s)))

pgm =
```
a = 10
print ("the value f a = %d" %a)
b = 20
c = 30
print ("b = %d and c = %d", (a, b, c))
```

pgm :
```
s1 = "Ira"
L1 = [10, 20, 30, 40]
print ("%s no items are %s %s" (s1, L1))
```

→ print with replacement operator : { }

syntax :

```
print ("{}".format(variable));
```

pgm :

```
name = "Ira"
salary = 10000
age = 10
print("Hello {} your salary is {} and age is {}".format
(name, salary, age));
print("Hello {0} your salary is {1} and age is {2}".format
(name, salary, age));
print("Hello {0} your salary is {1} and age is {2}".format
(name, salary, age))
print("Hello {a} your salary is {b} and age is {c}".format
(a=name, b=age, c=salary))
```

Input functions

→ By using input function we read the data from user dynamically at the run time

write a pgm to perform addition of two no's

syntax : input (prompt)

```
i = input("enter 1st no")
j = input("enter 2nd no")
sum = i+j
print(sum)
```

→ from the console whatever the value we read pvm will consider as string datatype

Datatype Type casting

→ converting one datatype to another datatype.

→ In python we have built in datatypes to convert one datatype to another datatype

1) int() → It is used to convert another datatype into integer datatype

2) float(), list(), set(), str(), dict(), tuple().

Add of 2 no's

```
i = eval(input("enter 1st no"))
j = eval(input("enter 2nd no"))
sum = i+j
print(sum)
```

eval function

→ Based on the type of value automatically it will be converted to the required datatype.

Control flow statements

→ By using control flow statements we control the execution of python pgm. It specifies the order in which statements should be execute.

→ python support 3 types of control flow stmts

1) Conditional stmts (if)
   a) if
   b) if else
   c) if else - elif
   d) nested if

2) Iterative stmts (or) loops

→ If a stmt (or) block of stmts repeat for multiple times go with loops concepts.

→ python supports 2 types of loops

1) for loop → it is used when we know the fixed no of iterations
   ↓
   i) simple for loop
   ii) Enhanced for loop (for each)
   iii) Infinite for loop
   iv) labelled for loop.

2) while loop

→ It is called as entry control loop

→ It is used when we don't know the fixed no. of iterations.

- python supports 2 types of while loop
  1) simple while loop 2) Infinite while loop

3) Jump strmts

→ these strmts are used to jump from one step to another step.

1) Break
2) Continue
3) Return
4) del
5) pass

Conditional strmts

→ For different conditions we perform different actions
→ python support different types of conditional strmts

1) simple if

Syntax: if condition:
statement 1 → if is inside the if
if is outside the if. statement

2) if else

syntax: if condition:
statement 1
else:
statement 2

3) if - elif - else

syntax: if condition1:
statement 1
elif condition2:
statement 2
else:
statement 3

write a pgm to find greatest among 3 nos.

```
i = eval(input("enter 1st no"))
j = eval(input("enter 2nd no"))
k = eval(input("enter 3rd no"))
if i>j and i>k:
    print(i," is greater")
elif j>k:
    print(j," is greater")
else:
    print(k," is greater")
```

o/p : enter 1st no 11
enter 2nd no 22
enter 3rd no. 33
33 is greater.

Iterative stmts

1) for loop :

syntax : for variable in sequence
statements.

→ sequence means collection like string, array, list, tuple, set, dict, object etc.

Pgm

```
s = "python"
for i in s:
    print(i, end=" ")
```

python not support ++ & --

o/p :
python

```
s=input("enter the string")
i=0
for x in s:
    print("character at", i, "index is", x)
    i=i+1
```

o/p: enter the string: python

character at 0 index is p
character at 1 index is y
so on.

Range functions

syntax: this function is used to displays the range of values

syntax: range(starting index, ending index, step)

Ex: range (0, 10, 2) default index=0
default step=0

range(10)

o/p= 0 1 2 3 4 5 6 7 8 9

ex pgm:

```
for i in range(11):
    print("hello")
for j in range(0, 11, 2):
    print(j)
```

write a pgm to display numbers from 0-10 descending order

```
for i in range(10, 0, -1)
    print(i)
```

o/p: 10 9 8 7 6 5 4 3 2 1 0

Adding list of elements.

```
li=eval(input("enter the list"))
sum=0
for i in li:
    sum=sum+i
print(sum)
```

o/p : enter the list {10, 20, 30, 40}
100.

while loop

→ It is used when we don't know the fixed no. of iterations

Syntax : Initilization
while condition:
statements
increment/decrement

pgm → i=0
while i<=10:
print(i)
i=i+1

o/p : 0

write a pgm to find sum of the n natural no's

```
n=eval(input("enter the value of n"))
sum=0
i=1
while i<=n:
    sum=sum+i
    i=i+1
print(sum)
```

```
name = " "
while name != "ida":
    name = input("enter the name")
print (TQV)
```

Infinite loops

→ the loops which don't have any conditions is called Infinite loops.

```
pgm i=0
    while true:
        print(i)
        i=i+1
```

Nested loops

→ the loop which resides inside the another loop is called nested loop.

syntax: for i in range(5)
    for j in range(5)
        print(i,j)

```
o/p: i=0  j=0      i 0   j 0
     i=0  j=1      i 1   j 1
     i=0  j=2      i 2   j 2
     i=0  j=3      i 3   j 3
     i=0  j=4      i 4   j 4.
```

write a pgm to print stars in right angled triangle

```
n=int(input("enter the no. of rows"))
for i in range(1, n+1):
    print("*" x i)
```

o/p: enter the no. of rows: 5

```
*
* *
* * *
* * * *
* * * * *
```

jump stmts

→ these stmts are used to control the loops

1) break

→ we can use break stmts in loops, switch

pgm for i in range(10):
    if i==7:
        break
    print(i)

o/p: 0
1
2
3
4
5
6

continue statements

→ we can use continue statements to skip current iteration & continue next iteration.

pgm: li = [10, 20, 500, 30, 40]
for i in range(li):
    if i>500:
        continue
    print(i)

o/p: 10
20
30
40

pgm: numbers = [10, 20, 0, 5, 0, 30]
for i in numbers:
    if i==0:
        print("number can't div by zero")
        continue
    print("100/{}={}", format(i, 100/i))

division operator (/) → the output always floating point
→ In this both arguments should be integer i.e 100/10 = 10.0

Float division operator (//) → the output will be in float if any of the argument is different. i.e
100//10.5 = float value

loop else block

→ If break stmt is not executed then else block is executed.

del :

By using del keyword we deallocate the memory for the unused variables

means that variable is eligible for garbage collector

pgm
```
i=10
del i
print(i)
```
o/p : error.

pass :-

By using pass keyword we don't do anything. It is a null or empty state

pgm
```
i=10
while true : pass
```
o/p :-

list datatype

→ list is a collection of Heterogeneous elements
→ Duplicate values are allowed.
→ Insertion order is preserved.

0	1	2	+ve
10	11	12	
-3	-2	-1	-ve

→ list support both -ve & +ve index.
→ we can access the elements from the list
→ we can access the element from the list by using index (or) slice (:) operator
→ In python list is created in 2 ways:
1) list(sequence)
2) Li = [ ]

→ listname[index]

write a pgm accept the values into lists dynamically

```
li=eval(input("enter the elements"))
print(li)
```

o/p: enter the elements: [10,20,30]
[10, 20, 30]

```
s="python"
li=list(s)
print(li)
```

o/p: ['p','y','t','h','o','n']

```
li=list(range(0,21,2))
print(li)
```

o/p: 0 2 4 6 8 . . . 20

```
s="python is very easy"
li=split(" ")
print(li)
```

o/p: ['python','is','very','easy']

```
print(type(li))
```

→ list are mutable (modifyable)

```
li=[10,11,12,20]
li[0]='python'
print(li)
```

o/p: ['python',11,12,20]

→ print("hi")
print("10/10")
print("welcome")

→ n=[0,1,2,3,4,5,6,7,8,9,10]
print(n[2:7:2])

→ we can access the elements by using slice operator

syntax: listname [start : stop : step]

→ Default starting value is "zero"
→ Default stop value length of the list
→ Default step value is "one"

pgm
```
n = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print (n[4:100])
print (n[2:7:2])
print (n[4::2])
print (n[3:7])
print (n[8:2:-2])
```

op: [5, 6, 7, 8, 9, 10]
[3, 5, 7]
[5, 7, 9]
[4, 5, 6, 7]
[9, 7, 5, 3]

write a pgm display the element both +ve & -ve index
write a pgm to display even no's from the list without using step

1)
```
n = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in n
    if i % 2 == 0:
        print(i)
```

2)

```
li=['A','B','C']
x=len(li)
for i in range(x):
    print("character", li[i], "positiveindex", i, "-ve
    index", -len(li)+i)
```

o/p:

character A positiveindex 0 -ve index -3
character B positiveindex 1 -ve index -2
character C positiveindex 2 -ve index -1

Important functions of list

1) len(r) → It returns no. of elements in a given list
2) count() → It returns no. of occurence of specific element in a list
3) index() → It returns index of specific element if the element is not available we get expection
4) Apperd() → Adding element to the list at the last
5) remove() → It removes specific element from the list
6) pop() → It will remove last element from the list & it will print removed element

```
Li=[]
Li=apperd(11)
Li=apperd(33)
Li=insert(4,777)
Li=insert(100,99)
Li=insert(-100,88)
print(Li)
```

0	1			
11	33			
0	1	2		
11	777	33		
11	777	99	33	
88	777	99	33	

o/p: 88 777 99 33

write a pgm add values into the list up to 100. which are divisible by 10

```
Li=[]
for i in range(0,101):
    if i%10==0:
        Li.append(i)
print(Li)
```

o/p: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

```
Li=[10, 20, 10, 40, 50]
print(Li.index(20))
print(Li.index(10))
print(Li.index[48]) → o/p: expection
```

o/p: 1
0 → first occurence

```
print(Li.remove(10)  o/p: 20 10 40 50
print(Li.pop()) → o/p: 20 10 40,
```
to display.

Reverse()

→ this function is used to print elements in reverse order.

o/p: Li=[10, 20, 30

sort():

→

```
Li=[10, 11, 44, 1, 12, 13, 22]
Li.reverse()
print(Li)
Li.sort() → ascending
print(Li)
Li.reverse() → descending
print(Li)
```

o/p: [22, 13, 12, 1, 44, 11, 10]

[1, 10, 11, 12, 13, 22]

[22, 13, 12, 11, 10, 1]

```
Li=[10, 'i', 'c', 'py']
Li.sort()
print(Li)
```

o/p: expection

```
li=[10,11,44,1,12,13,22]
li.sort(reverse=True)
print(li)
```

o/p: 44, 22, 13, 12, 11, 10, 1

+ By default reverse property is false in sort function

→ Aliasing & cloning:

Alias() → Giving alternate name for existing list

→ In Aliasing what are the changes you made in one list it will be effected in another list

Alternate solution to this is "cloning"

cloning:- Creating exactly duplicate copy of the original one

→ cloning is possible in python by using copy() & slice operator.

Alias pgm:

```
listname=[10,11,44,1,12,13,22]
li=listname
print(li)
print(listname)
li[0]=999
print(listname)
```

o/p: 10 11 44 12 13 22
10 11 44 12 13 22
999 11 44 12 13 22

cloning:

```
liold=[10,11,44,1,12,13,22]
linew=liold[::] or linew=liold.copy()
print(linew)
linew[0]=999
print(liold)
```

o/p:

Membership operators

In → exist → True

not in → not exist → False

pgm: li = [10, 20, 30, 40]
print (li in 30)
print (li not in 40)

o/p: True
False

Comparing the list (=, ≠)

→ comparing the list by using equal & equal to operators

→ 1) no. of elements
2) order of the elements
3) content (Case sensitive)

li1 = ['Dog', 'cat', 'Rat']
li2 = ['Dog', 'cat', 'Rat']
li3 = ['DOG', 'CAT', 'RAT']
li1 == li2 is true
li1 == li3 is false

→ In case of Relational operators only the first element will be compared.

ex: li1 = [10, 20, 30]
li2 = [9, 20, 30]
li1 > li2 = True

Nested list

→ The list inside the another list is called nested list

ex: li = [10, 20, 30, [40, 50]]
(0, 1, 2, 3; 0, 1)

li[3][0] = 40
li[0] = 10

```
n = [[10,20,30], [40,50,60], [70,80,90]]
for i in range(len(n))
    for j in range(len(n[i]))
        print(n[i][j], end=" ")

    print()
```

o/p: 10 20 30
40 50 60
70 80 90

List Comprehension

→ creating the list with the help of iterable objects like range, list, string, tuple, set, array, dict

syntax: list = [expression for item in iterable if condition]

Ex: list = [i+i for i in range(1,11)]

pgm: print(list)

o/p: [1, 4, 9, 100].

ex: list = [i for i in range(1,11) if i%2==0]
print(list)
o/p: [2, 4, 6, 8]

li = ['python', 'java', 'AI']
Lirew = [i[0] for i in li]
print(Lirew)
o/p: ['p', 'j', 'A']

Tuple datatype

→ It is same as list datatype.
→ Tuple are immutable (not modifyable)
→ It is read only version of list
→ It is represented by using parenthesis ( ) → are mandatory

t = 10, → int datatype

t = (10,) tuple

t = (10) → int t = (10,) tuple

-) when we have only one element place comma (,) after element. so, that it represent tuple datatype

```
t = (10,)
print(type(t))

t = (10, 20, 30, 40)
t1 = list(t)
t1[0] = 100
~~print(t1)~~ t3 = tuple(t1)
print(t3)
```

Set datatype

-) set is a collection of unique elements
-) set doesn't allow duplicate values
-) Insertion order is not preserved
-) Indexing & slicing are not supported
-) set is a collection of heterogeneous elements
-) set objects are mutable
-) we can represent the set by using open & close curly braces { }
-) empty set = set()
-) empty dict = s = { }, print(type(s)), o/p: dict
-) empty curly braces indicates empty dictionary
-) we represent empty set by using set function
-) By using set function we create a set

Syntax: set(sequence)

-) Important functions of set

1) add(x) -) this function is used to add elements inside the list
2) update(1-) it is used to add multiple elements inside the set

it should not be individual elements. it should be iterable objects like list, range, tuple, set etc

pgm:

```
s=set()
s.add(10)
s.add(20)
s.li=[11,22,33]
s.update(li,range(5))
print(s)
```

o/p: 11 20 10 33 22 0 1 2 3 4

pop(): this function is used to remove the random element from the set

```
s={10,20,30}
s.pop()
print(s)
```

o/p: 20,30

```
s={10,20,30}
s.remove(40)
print(s)
```

replace with discard() to not get error

o/p: KeyError

Mathematical operations on set

1) union() → this function returns all the elements from both the sets

```
s1={10,20,30,40}
s2={10,20,50,60}
print(s1|s2)
print(s1.union(s2))
print(s1&s2)
print(s1.intersection(s2))
print(s1-s2)
print(s1.difference(s2))
print(s1^s2)
print(s1.symmetric_difference(s2))
print(s1<s2)
```

write a pgm eleminate duplicate values from lists

```
Li1=[10,20,30,10,20]
Li2=[]
for i in Li1:
    if i not in Li2:
        Li2.append(i)
print(Li2)
```

o/p: [10, 20, 30]

write a pgm to find unique vowels in a given word

```
s="python"
s1={'a','e','i','o','u'}
w=input("enter the word")
s1=set(w)
s2={'a','e','i','o','u'}
print(s1.intersection(s2))
```

o/p: python
pytho

Dictionary datatype

→ Dictionary is a collection of key value pairs
→ It is a collection of Heterogereous values
→ key can't be duplicate & values can be duplicate
→ Insertion order is not preserved
→ Indexing & slicing are not support
→ Dictionaries are mutable
→ Dictionaries are dynamic
→ we can create a dict by using empty curly braces (or) dict()
→ d = {key:value, key:value}
→ d[key] = value
→ d = {}
→ we can access value from the dictionary with the help of key
→ d={ empno: 93, address: 'hyd'}

→ d = {100:'aa', 200:'bb', 300:'cc'}
print(d[300]) → o/p: cc
# print(d[400]) → error: keyerror
if 400 in d:
    print(d[400])
else:
    print("not there")
] o/p: not there

→ update the value in dictionary
d = {100:'aa', 200:'bb', 300:'cc'}
d[100] = 'iva'
print(d)] → o/p: {100:'iva', 200:'bb', 300:'cc'}
d[400] = "Python"
print(d)] → o/p: {100:'iva', 200:'bb', 300:'cc', 400:"Python"}

→ If the key is available value will be updated
→ If the key is not available key value pair will be added to the dictionary

write a pgm add student name & percentage of marks into dictionary & display

```
d = { }
n = input("enter no. of records")
i = 1
while i <= n:
    name = input("enter name of student")
    marks = eval(input("enter % marks"))
    d[name] = marks
    i = i+1
print("name", "% of marks")
for i in d:
    print(x, "\t", d[x])
```

o/p: enter no. of records. 2
enter name of student vishal
enter % marks 99.9999999
name % of marks
vishal 99.9999999

Deleting pair from the dictionary

d={100:'aa', 200:'bb', 300:'cc'}

del d[200]
print(d) → o/p: {100:'aa', 300:'cc'}

d.clear() → o/p: {}
print(d)

del d
print(d) → o/p: error

get(key)

→ It returns the value based on the key
→ If the key is not available we get default value

d={100:'aa'}. get(key)

syntax: get(key, default msg)

pgm d={100:'aa', 200:'bb', 300:'cc'}
print(d.get(400, "not there"))
o/p: not there

pop()

→ It will remove key value associated with key & returns corresponding value.
→ If it is not available we get key error expection

d={100:'aa', 200:'bb', 300:'cc'}
print(d.pop(100)) → o/p: aa
print(d) → o/p: {200:'bb', 300:'cc'}

popitem()

→ It will remove key value pair from the dictionary & returns it.

d={100:'aa', 200:'bb', 300:'cc'}
print(d.popitem()) → o/p: 300:'cc'
print(d) → o/p: {100:'aa', 200:'bb'}

keys()

→ It is used to display all the keys from dictionary

values()

→ It is used to display all the values from dictionary

```
d={100:'aa', 200:'bb', 300:'cc'}
print(d.keys()) → o/p :- 100 200 300
print(d.values()) → o/p :- aa bb cc
for i in d:
    print(i,d[i]) → o/p :- 100 aa
                           200 bb
                           300 cc
```

items()

→ It is a collection of list of tuples representing key value pairs.

```
d={100:'aa', 200:'bb', 300:'cc'}
for k,v in d.items():
    print(k,-----,v)
```

o/p :- 100 ----- aa
200 ----- bb
300 ----- cc

write a pgm to print sum of value in a given dictionary

```
d=dict(input("enter values"))
s=sum(d.values())
print(s)
```

o/p : enter values: {"aa"=100, "bb"=200}
300

write a pgm find no. of occurence of a word in a given string

```
s=input("enter name")
d={}
for x in s:
    d[x]=d.get(x,0)+1
for k,v in d.items():
    print(k,"occurs",v,"times")
```

o/p :-

```
word = input("enter name")
vowels = ['a', 'e', 'i', 'o', 'u']
d = {}
for x in word:
    if x in vowels:
        d[x] = d.get(x, 0) + 1
for k, v in d.items():
    print(k, "occurred", v, "times")
```

write a pgm accept name & % of marks into dictionary & display marks based on name

```
n = int(input("enter no. of records"))
d = {}
for i in range(n)
    name = input("enter name")
    marks = eval(input("enter marks"))
    d[name] = marks
while True
    name = input("enter the name")
    marks = d.get(name, -1)
    if marks == -1:
        print("not found")
    else:
        print("marks of", name, "are", marks)
option = input("Do you want to continue [Yes/No]")
if option = "No":
    print("not found")
    break
print("Bye")
```

functions

-) A group of stmts under a name is called function
-) the main advantage of function is "Code Reusability."
-) python support 2 types of functions
  1) Built In functions 2) p
-) the functions which comes automatically from python is called Built In functions
2) User Defined functions
-) the functions which are created by the users

Syntax: def function_name (parameters)
        stmts
        return values

def, function_name are mandatory
stmts, return values, parameters are optional

Ex: def f1( ):
        print ("welcome to function")

f1() → o/p: welcome to function

-) parameters are the input to the function while calling the function we have to pass value to the parameter
-) No. of parameters & no. of values should be balance. otherwise expection

def f1 (name)
    print("welcome to ....", name)

f1("isa") → welcome to..isa
f1(100) → welcome to.... 100
f1(11.23) → 11.23

function with return stmt

→ In c, c++, java function will return only one value

→ In python function will return multiple values

→ Return keyword should be last stmt inside the function. After Return stmt - if we write any stmt then those stmts will not be considered

write a function to perform add of 2 no's

```
def sum(a,b):
    return a+b

add=sum(10,20)
print(add) (or)
print(sum(10,20))
```

o/p : 30

→
```
def f1():
    print("hello")

f1()
print(f1())
```

o/p : hello
hello
none

* for function returning multiple values

```
def sum_sub_mul_div(a,b):
    sum=a+b
    sub=a-b
    mul=a*b
    div=a/b
    return sum,sub,mul,div
```

```
sum,sub,mul,div = sum_sub_mul_div(20,2)
print(sum)
print(sub) (or)
```

→ o/p 22
18

```
t=sum_sub_mul_div(20,2)
print(type(t))
for i in t:
    print(i)
```

→ o/p 22
18

→ The function which doesn't have any name is called Anonymous function (or) lamda function. These are the Instantaneous functions

Types of Arguments

def f1(a,b)
↳ a,b are formal parameters

f1(a,b)
↳ a,b are Actual parameters

→ python support 4 types of arguments
1) positional arguments 2) keyword arguments
3) Default args 4) variable length args

1) positional arguments:
→ these are the arguments passed to the function in Correct positional order

ex: def sub(a,b):
print(a-b)

sub(100,200)
sub(200,100)

→ If we change the order we don't get ~~order~~ error we get different output

2) Keyword Arg:
→ we can pass the values to the args by using keyword

def wish(name,msg):
print("Hello",name,msg)

wish(name="ira", msg="Gm")
wish(msg="Gm", name="ira")

o/p: Hello ira Gm
Hello ira Gm

→ we can mix positional args with keyword arg but first arg should be positional. ex: wish("ira", msg="gm")

Default args

→ we can pass default ~~arg~~ values to the args

```
def wish (name="Nun")
    print("Hello", name)
```

wish( ) → o/p :- Hello Nun
wish(two) → o/p :- Hello two

→ After default parameters we should not take non default arg.

```
def wish (name, msg="Hello") → correct order
def wish (msg="Hello", name) → error
```

Variable length args

→ we can pass "n" number of args to a function it is called as variable length args

→ we can declare variable length args with "*" symbol

syntax : def fun_name (*n):

Write a pgm to find sum of the n numbers

```
def f1 (*n)
    sum=0
    for i in n:
        sum=sum+i

    print(sum)
```

f1( ) → o/p = 0
f1 (1,2,3) → 6
f1 (2,5,6) → 13

→ we can pass positional arg with variable length arg

```
def f1 (s, *n):
    print(s)
    for i in n:
        print(i)
```

f1(10, 20, 30, 40) → o/p : 10
20
30
40

→ After variable length arg if your taking any arg that arg should be keyword arg

```
def f1(*n, s)
    print(s)
```

f1(11, 12, 13, 14, s=14) → o/p: 14

Types of variables

1) local variable:

→ The variable which is declared inside the function (or) method (or) block (or) constructor is called local variable.

→ The scope of local variable is with in the function

```
i=100
def f1():
    i=10
    print(i) → o/p = 10
def f2():
    print(i) → error
    print(i) → o/p = 100
f1()
f2()
```

→ The variable which is declared above all the functions is called global variable

→ If local variable & global variable is same by using globals() we can access global variable value

```
a=10
def f1():
    a=777
    print(a)
    print(globals()['a'])
```

f1() → o/p: 777
10

→

```
def f1():
    global a
    a=777
    print(a)
def f2():
    print(a)
```

f1() → 777
f2() → 777

→ we can make local variable as global variable by using global keyword

lambda functions

→ By using lambda keyword we create lambda functions

lambda arglist : expression

write a pgm to find square of a given no. by using lambda functions

```
s = lambda n : n*n
print(s(6)) → 36
```

write a pgm to find sum of 2 no's

```
s = lambda a,b : a+b
print(s(2,3)) → o/p : 5
```

?: → ternary operator

write a pgm to find greatest among 2 no's

```
s = lambda a,b : a if a<b else b
print(s(2,3)) → o/p : 3
```

```
import time
def f(c):
    for i in range(1,11):
        print(i)
        time.sleep(2)
```

f(c) → o/p 1 after 2 seconds
2
...

Recursive functions

→ the function which call by itself is called Recursive functions

→ By using Recursive functions we can reduce lengthy of coding

modules

→ module is a collection of functions, variables, classes & methods. Every .py file is considered as a module.

→ To access members of the module we have to import the module

Syntax:- import modulename

→ To access members of the module we write modulename.functionname, modulename.methodname

firstex.py

```
x = 999
def add(a,b)
    print(a+b)
def sub(a,b)
    print(a-b)
```

secondex.py

```
import firstex
print(firstex.x)
print(firstex.add(30,20))
print(firstex.sub(30,20))
```

→ Renaming the module at the time of importing by using "as" keyword.

```
import firstex as fe
print(fe.x)
```

→ import firstex

print(dir(firstex)) → it gives only list of names

print(help(firstex)) → it gives complete details

→ from firstex import * → all functions / methods / variables

print(x) → 999

add(10,10) → 30

Random Module

→ It is used to generate random numbers. used for security purpose

Random function

→ It generates floating point value b/w 0 & 1

from random import *

for i in range (5):

print(random()) → o/p : 0.4446056289 so on to 5 rows

randint()

→ It generates random integer values b/w two given numbers

from random import *

print(randint(0,9)) → o/p = 7

print(randint(10,99)) → o/p : 22

print(randint(0,9), randint(0,9), randint(0,9)) →

o/p : 7 8 6

(or) print(chr(randint(65,97)), randint(0,9))

o/p : A 7

uniform()

→ It generates floating point values b/w two given numbers

from random import *

print(uniform(1,10))

o/p : 2.5628205628

choice()

→ It won't return random number. It will return random object from the given list (or) tuple

from random import *

li = ['c', 'c++', 'java']

print(choice(li))

o/p : c

(or) c++

OOPS

→ If any programming language follow the concepts like class, object, abstraction, encapsulation, Inheritance & polymorphism then it is oop language

class :

→ class is a collection of data variables & methods
data members member functions

→ class is a logical representation of data

→ class doesn't hold any value

→ we don't allocate any memory for a class

Syntax : class classname:
variables
methods()

object :

→ It is an instance of the class

→ It is the physical representation of the data

→ object holds the value

→ with the help of the object we can access variables & methods of a class

syntax for creating object of a class :

reference variable = className()

objectname.variablename

objectname.methodname()

Types of variables

1) local variables → method level
2) Instance variables → object level
3) static variables → class level

Methods

→ In python class methods are used to perform operations

Types of Methods

→ python support 3 types of methods :

1) Instance Method
2) static Method
3) class Method

Abstraction

→ Hiding the implementation & showing the functionality is called Abstraction
→ Hiding unwanted info. from the user is called abstraction
→ By using abstract class abstraction is possible

Encapsulation

→ wrapping (or) Combining the related info. into single unit is called encapsulation
→ Encapsulation is possible by using classes

polymorphism

→ poly means many, morphism means forms
→ one name different functionalities is called polymorphism
→ python support 2 types of polymorphism
1) Compile time polymorphism
→ It is achieved by using overloading concept
→ python support 3 types of overloading
1) operator overloading
2) method overloading
3) Constructor overloading
2) Run time polymorphism
→ It is achieved by using overriding concept
→ python support 2 types of overriding
1) method overriding
2) constructor overriding

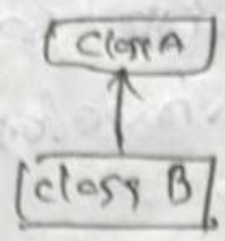
Inheritance

→ Extracting the properties from one class to another class is called Inheritance

→ python support 5 types of Inheritance

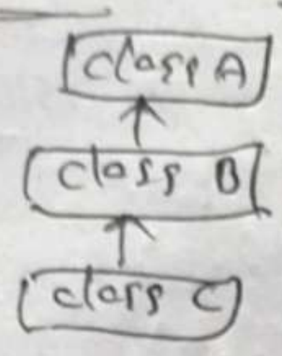
1) single level Inheritance

Ex:

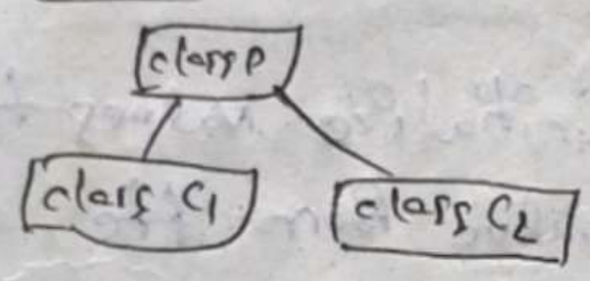


2) Multilevel Inheritance

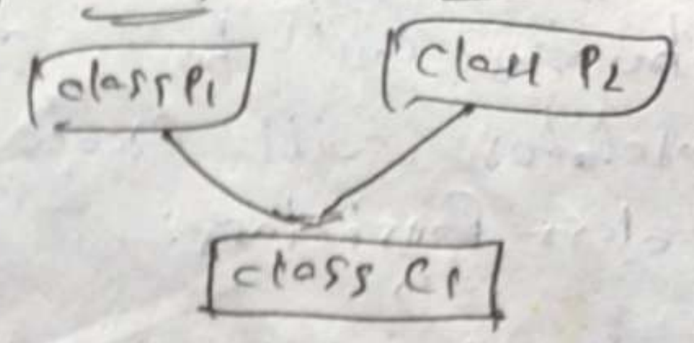
Ex:



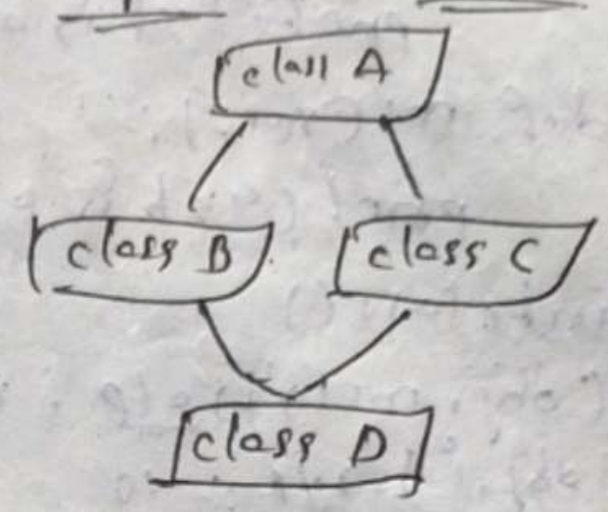
3) Hierarchical Inheritance



4) multiple Inheritance



5) Hybrid Inheritance



→ In python multiple Inheritance is possible by ~~Inheritance~~ ~~Interfaces~~ using classes.

Syntax for creating method

```
def methodname(self, args):
        stmts
        return value
```

→ def, methodname are mandatory & others are optional

self

→ self is the default variable which refers to the default object

→ it is like this keyword in java

→ In method & constructor self should be first argument

constructor

→ constructor is a type of method which is used for initializing values to the variable based on the datatype.

→ Integer default value : 0

→ Float default value : 0.0

→ String default value : NULL

→ boolean default value : false

→ Name of the constructor should be (__init__(self))

→ once we create object for a class automatically constructor will be created otherwise python interpretator will create default constructor

```
class firstclass:
    i=10
    def __init__(self):
        print("It is a constructor")
    def m1(self):
        print("It is a method")
obj = firstclass()
print(obj.m1())  o/p : It is a c.
print(obj.i)  o/p : 10
```

calling at time

→ with the help of the self we can initialize values to the variables & we can access value from the variable class values:

eg :-

```
def __init__(self):
    self.name = "Ira"
    self.number = "99"
    self.address = "Hyde"
def display(self):
    print(self.name)  o/p : Ira
    print(self.number)  99
    print(self.address)  Hyde
obj = value()
obj.display()
```

→

```
class Example:
    def __init__(self, x, y, z):
        self.name = x
        self.no = y
        self.add = z
    def display(self):
        print(self.name)
        print(self.no)
        print(self.add)
```

```
obj = Example("Ira", 99, "Hyd")
obj.display()
obj1 = Example("Vishal", 98, "Cola")
obj1.display()
```

Types of variables

1) Instance variable

→ If the value of the variable is different from each one object to another object then it is instance variable

→ for every object a separate copy of instance variable is created

→ If we change the value in one object the value will not reflect in another object

→ places to declare instance variable

1) Inside the constructor by using self

2) Inside the method by using self

3) outside the class by using object reference variable

2) static variable

→ If the value of the variable is not different from one object to another object then it is static variable

→ per one class static variable is created only once.

→ static variable is declare above all methods & below the class

→ places to declare static variable

1) Inside the constructor by using classname

2) Inside the instance method by using classname

3) Inside the classmethod(cls) by using classname (or) cls variable

4) Inside the staticmethod by using classname

5) outside the class by using classname (or) object reference variable

→ If we make changes in one object it will reflect in another object

```
class example:
    def m(self,ename,eadd,ero):
        self.ename = "kristal"
        self.eadd = "HNK"
        self.ero = 99
    def display(self):
        print(self.ename)
        print(self.eadd)
        print(self.ero)
obj = example()
obj.m()
obj.display()
```

```
class Test
    def __init__(self):
        self.ename='Ra'
        self.cro=99
        self.add='HYD'

T=Test()
print(T.__dict__)
```

(or)

```
def display(self):
    print(self.ename)
    print(self.cro)
    print(self.add)
T.display()
```

---

case

```
class Example:
    def __init__(self):
        self.a=10
        print(self.a)

    def m1(self):
        self.b=20
        print(self.b)

c=30
obj=Example()
obj.m1()
print(c)
```

```
def display(self):
    print(self.a)
    print(self.b)
    pri
```

```
class Test
    def __init__(self):
        self.a=10
    def m1(self):
        self.b=20

obj=Test()
obj.c=30
print(obj.__dict__)
```

→ we can modify the instance variable

→ we can delete the instance variable by using "del" keyword

→ class Test:

```
    def __init__(self):
        self.a=10
        self.b=20

T1=Test()
T2=Test()

T1.a=90 888
T2.b=40 999

print(T1.a,T1.b)=~~30,20~~ 888 999
print(T2.a,T2.b)=~~10,40~~ 10 20
```

```
class Test:
a=10
  def m1(self):
    self.a=888
obj = Test()
obj.m1()
print(obj.a) -> o/p=888
print(Test.a) -> o/p=10
```

→ we can't change the value of the static variable by using self. (or) object reference variable

```
class Test
    a=10
  def __init__(self):
        self.b=20

t1 = Test()
t2 = Test()
Test.a=888
t1.b=999
print(t1.a, t1.b) -> 888, 999
print(t2.a, t2.b) -> 888 20
```

```
class Test:
    a=10
  def __init__(self):
        self.b=20
  def m1(self):
        self.a=888
        self.b=999

t1=Test()
t2 = Test()
t1.m1()
print(t1.a, t1.b) -> 888, 999
print(t2.a, t2.b) -> 10, 20
```

import sys

```
class Customer:
    bankname = "AxisBank"
    def __init__(self, name, balance=0.0):
        self.name = name
        self.balance = balance
    def deposit(self, amt):
        self.balance = self.balance + amt
        print("After deposit the balance is :", self.balance)
    def withdraw(self, amt):
        if amt > self.balance:
            print("insufficient balance")
            sys.exit()
        self.balance = self.balance - amt
        print("After withdraw balance is :", self.balance)
print("welcome to ", customer.bankname)
name = input("enter the name")
c = customer(name)
while True:
    print("D = deposit\n W = withdraw\n E = exit")
    option = input("enter the option")
    if option = 'D' or option = 'd':
        amt = float(input("enter amount"))
        c.deposit(amt)
    elif option = 'w' or option = 'w':
        amt = float(input("enter the amount"))
        c.withdraw(amt)
    else option = 'e' or option == 'E':
        print("thank you for banking")
        sys.exit()
    else:
        print("invalid option")
```

o/p: welcome to Axis Bank
enter the name vishal
D = deposit
W = withdraw
E = exit
enter the option

Types of methods

→ Python support 3 types of methods

1) Instance method → Inside method implementation if we are using instance variables then such type of method is called Instance method.

→ Instance method we have to pass atleast one parameter i.e., self

→ Instance method is called with the help of object

→ within the class we can call the instance method by using "self" variable.

→ outside the class by using object reference variable

```
class std:
    def __init__(self,name,marks):
        self.name=name
        self.marks=marks
    def display(self):
        print("hi", self.name)
        print("ur marks are", self.marks)
    def grade(self):
        if self.marks>=65:
            print("first class")
        elif self.marks>=55:
            print("second class")
        elif self.marks>=35:
            print("third class")
        else:
            print("fail")

n=int(input("enter no. of records"))
for i in range(n):
    name=input("enter name")
    marks=eval(input("enter marks"))
    s=std(name,marks)
    s.display()
    s.grade()
```

print()→ it gives the space after each record

class std:
~~def __init__(self, name, marks):~~

2) class Method

→ In the Method implementation if we are using "cls" variables then such type of method is called class method

→ It is declared with @classmethod decorator

→ Inside the class method we can call the variables by using cls variable or object reference variable

```
class Animal:
    legs=4
    @classmethod
    def walk(cls):
        print(cls.legs)
a=Animal()
a.Walk()→ o/p=4
```

→ passing members from one class to another class

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

class test:
    def modify(emp):
        self.esalary=emp.esalary+500
        emp.display()
e=Employee(100,"nani",5000)
Test.modify(e)
```

o/p: 100
nani
5500

Polymorphism

→ overloading

→ Method overloading (1 class)

→ In a class we can have same method names but different no. of parameters

→ Different datatype of the parameter

→ order of the parameter (signature)

→ python doesn't support method overloading & constructor overloading

→ If we write forcefully only the last method (or) last constructor will be called

→ operator overloading

→ one symbol can perform multiple operations

ex: +, *

→ In python operator overloading is possible by using magic methods

→ __add__() → magic methods

__var = 100 → private modifier

__var = 100 → protected modifier

var = 100 → default public

```
class example
    def m1(self):
        print("zero person")
    def m1(self,a):
        print("one person",a)
    def m1(self,a,b):
        print("two person",a,b)
obj = example()
obj.(11,11) → o/p : 22
```

→ Every operator in python will take magical methods

→ By using magical methods operator overloading is possible

→ __add__(self,other)

→ __sub__(self,other)

→ __gt__(self,other)

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

b1 = book(100)
b2 = book(200)
b3 = b1 + b2
print(b3) → error.
```

syntax:

```
def __add__(self, other):
    return self.var + other.var
```

Ex: class

```
    def __add__(self, other):
        return self.pages + other.pages
b1 = book(100)
b2 = book(200)
b3 = b1 + b2
print(b3) → 300
```

Ex: class student:

```
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    def __gt__(self, other):
        return self.marks > other.marks
    def __le__(self, other):
        return self.marks <= other.marks
obj1 = student("isa", 99)
obj2 = student("rani", 97)
print(obj1 > obj2) → True
```

Calendar Module

→ By using the calendar module we can perform actions on the calendar like month, day, year

```
import calendar
YY = int(input("enter year"))
mon = int(input("enter month"))
print(calendar.month(YY, mon))
```

→ import calendar
print(calendar.calendar(2020))

** (exponential operator)
R→L

Inheritance

→ Extracting the properties from one class to the another class (CC, DC, SC)
(PC, BC, SC)

syntax
class Pclass class.
P.var
P.const
P.method
class childclass (parent class)

Ex:
```
class Parentclass:
    i=100
    def __init__(self):
        print("pclass const")
    def m1(self):
        print("pclass method")
class childclass (parentclass):
    j=200
    def __init__(self):
        print("child class const")
    def m2(self):
        print("childclass method")
obj=childclass()
print(obj.i) → 100
print(obj.j) → 200
obj.m1() → pclass method
obj.m2() → child class method
```

```
class Animal:
    def eat (self):
        print("dog is eating")
class dog (Animal):
    def bark (self):
        print("dog is barking")
class babydog (Animal):
    def sleep (self):
        print("dog is sleeping")
obj=Animal()
obj1 = dog ()
obj1.bark()
obj1.eat()
obj2 = babydog()
obj2.sleep()
```

Multiple Inheritance.

→ In python multiple Inheritance is possible by using classes.

```
class Animal
    def eat (self):
        print("dog is eating")
class dog (self):
    def bark (self):
        print("dog is barking")
class babydog (Animal, dog):
    def sleep (self):
        print("dog is sleeping")
obj = babydog()
obj.sleep()
obj.bark()
obj.eat()
```

Runtime polymorphism

→ It is achieved by using overriding concept (2 class)

→ there are 2 types of overriding's

1) Method overriding 2) Constructor overriding

Method overriding

→ Re-defining the functionality of parent class method with the help of child class method

```
class Pclass
    def m1(self):
        print("pclass method")
class chclass(Pclass):
    def m1(self):
        print("childclass method")
obj = chclass()
obj.m1() → o/p: childclass method
```

super keyword

→ with the help of super we can access super class methods/constructors/variables

```
class Pclass:
    def __init__(self):
        print("pclass constructor")
class cclass(Pclass):
    def __init__(self):
        super().__init__()
        print("cclass constructor")
obj = cclass() → o/p: pclass constructor
                      cclass constructor
```

```
class palads:
    def __init__(self,name,age):
        self.name=name pistalname
        self.age=age 28 age

class classclass(pelass):
    def display(self):
        print(self.name

class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
class employee(person):
    def __init__(self,name,age,no,salary):
        super().__init__(name,age)
        self.no=no
        self.salary=salary
    def display(self):
        print(self.name)
        print(self.age)
        print(self.no)
        print(self.salary)
obj=employee("rani",23,190,5000)
obj.display().
```

File Handling

→ Files are used to save the data permanently in the local hard disk

→ Before we perform any operations on the file we have to open the file

→ python support built in functions "open" to open the file

syntax: f=open("filename", mode)

Ex: f=open("abc.py", "r")

Types of files

- python support 2 types of files
1) text file 2) Binary file

Text files

→ text files are used to read character data like
→ Binary files are used to read binary data
images, audio & video files, raw data

Modes

→ r mode

→ It reads the data if already exits.
→ If file doesn't exist we get "FileNotFound Error"

→ w mode

→ It opens the file for write operation. If the file doesn't exist it will create a file. If the file exists old data will be replaced with the new data

→ a mode

→ It opens the file for append operation. If the file doesn't exist it will create a file. If the file exist new data will be added to existing data
(last)

→ r+ mode (read & write mode)
→ w+ mode (write & read mode)
→ a+ mode (append & read mode)
→ x → open the file in exclusive mode

→ By using close() we are closing the file

properties of the file

→ Name
→ It returns name of the file

Mode

→ It returns mode of the file

closed

→ It returns whether file is closed or not

Readable

→ It returns whether file is readable or not

writable

→ It return whether file is writable or not

ex: f=open("abc.txt","w")
print(f.name)→o/p: abc.txt
print(f.mode)→o/p: w
print(f.closed)→o/p: False
f.close()→o/p: file closed
print(f.closed)→o/p: True
print(f.readable())→o/p: False
print(f.writable())→o/p: False (error)

writing the data to the file

→ By using "write" function we write character data to the text file

write(str)
writelines(list of lines)

ex: f=open("abc.txt","w")
f.write("learning python is easy\n")
f.write("welcome\n")
f.write("ise ins")
print("successfully created")
f.close()

Reading the data from the file

1) Read() → It reads total data from the file
2) Read(n): It reads n characters
3) Readline(): It reads only one line
4) Readlines(): It reads all lines into the list

```
data f=open("abc.txt","r")
print(f.read()) -> o/p: All data
    f.read(7)) -> o/p: learn
    f.readline() -> o/p: learning python is easy
print(data)
f.close()
```

use read lines are read the data

```
f=open("abc.txt","r")
data = f.readlines()
Li=list(data)
for i in Li:
    print(i, end=" ") -> o/p: learning python is easy
                              welcome
                              iva inst
f.close()
```

Seek & Tell methods

Seek() syntax:-

→ seek(offset, from where)

→ this function is used to move the cursor to the specific location.

→ from where values: 0 → from begining

→ 1 → from the current position

→ 2 → from the end

→ python 3 support only one value i.e "zero"

tell()

→ It returns where is your current cursor location

→ with statement is used by opening the file.

→ we can use this to group multiple options once we core with block automatically file will be closed

```
with open("abc.txt", "w") as f:
    f.write("welcome to iva")
print(f.closed)→True.
```

```
data = " All students are clevers"
f.open("abc.txt", "w")
f.write(data)
with open("abc.txt", "r+") as f:
    text = f.read()
    print(text)→o/p: welcome to All students are clevers
    print("current position is", f.tell())
    f.seek(17)
    print("current position is", f.tell())
    f.write(Cerms)
    f.seek(0)
    text = f.read()
    print(text)→o/p: All students are Cerms
```

→By using "os" module we can get complete info about the file. It has path sub module.

→we use isfile() to check whether file exist or not

```
import os
fn = input("enter the file name")
if os.path.isfile(fn):
    print("file is exist")
    f=open("fn", "r").
    i=f.read()
    print(i)
else:
    print("file doesn't exist")
```

o/p: abc.txt
file is exist
All students are Cerms

write a pgm to count no. of lines, characters, words in a given file.

```
import os
filename = input("enter filename")
if os.path.isfile(filename):
    print("file is exist")
    f.open("filename", "r")
else:
    print("file doesn't exist")

lcount = ccount = wcount = 0
for line in frame:
        ccount = ccount + len(frame)
        words = line.split(" ")
        wcount = wcount + len(words)
        lcount = lcount + 1
print(ccount)
print(wcount)
print(lcount)
```

Exception handling

→ while developing we consider 2 types of errors
1) Syntax errors 2) Run time errors.

Syntax errors (compile time error)

→ If the pgm is syntactically wrong we get syntax errors.

→ syntax errors are identified by language compilers & interpreters.

→ If we get syntax errors source code will not be converted into byte code.

Runtime errors

→ the errors which are occured at run time are called Runtime errors

→ PVM will identify the run time errors. If we don't handle the run-time errors, it leads to abnormal termination

→ Exception handling got 5 keywords
1) try 2) except 3) finally 4) raise 5) throws

try block:

→ In the try block we write functionality. (or) expection code.

Except block

→ From the try block we get exception that exception should be handled in except block

syntax:

```
except exceptionclassName as referencevariable:
    expectorcode
```

→ Always except block should follow the try block it can't be alone.

pgm p try:
```
    print(10/0)
except ZeroDivisionError as e:
    print(e) → default msg of class o/p: division by zero
    print("second no can't be zero")
print("welcome")
```

→ Every exception in a python is a class
→ All the exception class names are derived from base exceptions

finally block

→ finally block is executed before closing the pgm
→ In finally block we write memory related code, closing files, closing DB connections
→ for one try block we should have one finally block
→ finally block can follow after try block and except block

write a pgm to find division of 2 no's by using exception handling except

```
try:
    a=input(e
    a=eval(input("enter 1st no"))
    b=eval(input("enter 2nd no"))
print(c=a/b)

except zeroDivisionError:
    print("2nd no can't be zero")
```

Try with except block

```
try:
a=int(input("enter 1st no"))
b=int(input("enter 2nd no"))
div=a/b
print(div)
except zeroDivisionError as e:
    print(e)
except valueError as e:
    print("enter int values")
```

→ order of the exception is very important

→ single except block can handle multiple exceptions

→ except (zeroDivisionError, valueError) as e:
    print(e)

Default exception

→ Default exception can handle any type of exception.

→ It should be placed at last of the pgm

```
except:
    print("enter correct value:")
```

• except Exception as e:
    print(e) handle all exceptions

finally block

→ finally block is executed if we get expection or not

```
try:
  print("try block")
  print(10/0)
except:
   print("except block")
finally:
   print("finally block
```

o/p : try block
e b
f b

→ finally block is executed if we handle the expection or not

→ In all the cases finally block is executed. In only one case it is not executed if we manually exit from the pgm

```
try:
  print("try block")
  print(10/0)
except ZeroDivisionError as e:
    print("except block")
finally:
  print("finally block")
```

```
def fun(a):
    if a<4:
        b=a/a-3
        print(b)

try:
  fun(5)
except ZeroDivisionError as e:
    print(e)
```

→ python supports 2 types of expections

1) predefined exceptions

→ the exceptions which are raised automatically by the pvm are called builtin exceptions

2) usedefined exceptions

→ the exceptions which are raised manually by the user by using "raise" keyword are called usedefined exceptions.

Syntax :-

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

Ex:-

```
class Tooyoungexception(Exception):
    def __init__(self, arg):
        self.msg = arg
class Toooldexception(Exception):
    def __init__(self, arg):
        self.msg = arg
age = int(input("enter age"))
if age > 60:
    raise Toooldexception("u r too old")
elif age < 18:
    raise Tooyoungexception("u r too young")
else:
    print("your middle age")
```

Multithreading

→ thread is a small unit of code which has its own execution part.

→ thread is a light-weight device i.e, it consumes less memory

→ one thread is independent of other thread

→ there are 2 types of multitasking:

1) process based 2) thread based

→ In python Thread is a class
→ under the thread class we have following the methods
1) start() 2) stop() 3) resume() 4) setpriority()
5) getpriority()

syntax:
→ int getpriority() → str getName()
→ setpriority(int)
→ setName(string)

6) sleep(sec) 7) join (millisec) 8) Yield (milli) 9) wait() 10) run()

10) Daemon

Every python pgm will have one thread i.e., main thread

```
import threading
print(threading.Current_thread.getName())
```

o/p :- Main Thread

→ there are 3 ways to create a thread in python:-
1) creating the thread without using any class.
2) creating the thread by extending the Thread class
3) creating the thread without extending the thread class

creating the thread without using any class

```
from threading import *
    def display():
        print("welcome to thread")
t = thread(target = display())
t.start()
```

creating the thread by extending the thread class

```
from threading import *
class mythread(thread):
    def run(self):
        print("wel to thread")
obj = mythread()
obj.start()
```

Creating the thread without extending thread class

```
from threading import *
class mythread:
    def display(self):
        print("we to thread")
obj=mythread()
t=thread(target=obj.display())
t.start()
```

import time

-> from ~~import~~ threading import *

```
class mythread1(thread):
    def run(self):
        for i in range(4):
            print(i)
            time.sleep(2)
obj1=mythread1()
obj2=mythread2()
obj1.start(), obj2.join()
obj2.start()
```

~~obj=mythread()~~ time.sleep(2)

~~obj.start(),obj~~

```
class mythread2(thread):
    def run(self):
        for i in range(4):
            print(i)
            time.sleep(2)
```

-> we can't start the same thread twice. for the 1st time it will execute successfully, for the 2nd time we get runtime error

setName() & getName()

```
from threading import *
print(current_thread().getName())
print(current_thread().setName("ira"))
print(current_thread().getName())
```

op:- MainThread
ira

Thread class

setpriority() & getpriority()

→ Thread scheduler schedule the thread based on the priority

→ priority value ranges from "1 to 10"

→ "1" is min priority, "10" is the max priority, "5" is the default priority, Normal priority.

→ If we give max priority

Enumerate()

→ It displays list of all the active threads.

→ isAlive() → It returns whether thread is alive or not.

It returns boolean value

→ Count() → It counts how many threads are available.

return type is "int"

Daemon threads

→ The threads which are running in the background are called Daemon threads

ex: garbage collector.

isDaemon() → It is used check whether the thread is daemon or not

```
from threading import *
print(current_thread().isDaemon()) → False
current_thread().setDaemon(True) → error
```

→ Main thread is always non-daemon thread & we can't change its nature.

```
from threading import *
class mythread(Thread):
    def run(self):
        print("welcome to thread")
print(mythread().isDaemon())
print(mythread().setDaemon(True))
```

```
obj = mythread()
obj.setDaemon(True)
print(obj.isDaemon()) -> True.
```

Synchronization

→ The concept of synchronization is at a time one thread is allowed.

→ By using lock(), Rlock(), semaphore() synchronization is possible in python

Inter thread Communication

→ One thread is communicating with another thread is called Inter thread communication

Regular expressions----

→ If the group of characters represent in the customized format go with regular expressions

1) finditer() → It returns iterative objects which match with required pattern

Syntax: pattern finditer("abaabab")

Match object has following methods

1) start() → starting index of the match
2) end() → returns end+1 index of the match
3) group() → returns matched string

```
import re
pattern = re.finditer("ab", "abaababa")
count = 0
for match in pattern:
        count = count + 1
        print(match.start(), "....", match.end(), "....",
              match.group())
print("no of occurence =", count)
```

o/p: 0..... 2..... ab
3 5 ab
5 7 ab
no of occurence = 3

character classes

→ [abc] → either "a" or "b" or "c"
or
[a-c]

→ [^abc] → except a & b & c we can use rather characters

→ [a-z] → any lower case from "a to z"

→ [A-Z] → any upper case from "A to Z"

→ [a-zA-Z] → any alphabet either any upper or lower case

→ [0-9] → any digit from "0 to 9"

→ [a-zA-Z0-9] → any alpha numeric

→ [^a-zA-Z0-9] → any alpha numeric i.e., symbols

```
import re
pattern = re.finditer("[abc]", "akfbz")
for match in pattern:
    print(match.start(), "....", match.group())
```

o/p :- 0 .... a
3 .... b

special character classes

1) \s → space character
2) \S → any character except space character
3) \d → any digit from "0 to 9"
4) \D → any character except digit
5) \w → any word character → \w [a-zA-Z0-9]
6) \W → any character except word character
7) "." → any character including special character

```
pattern = re.finditer("\s", "abcd efg")
```
or
"\S"

o/p :- 4 .....

o/p :- 0 .... a
1 .... b
2 c
3 .... d
5 .... e
6 .... f
7 .... g

write a pgm to check the phone number whether valid or not

```
import re
phno = input("enter your phno")
m = re.fullmatch("[6-9]\d{9}", phno)
if m != None:
    print("Valid phno")
else:
    print("Invalid phno")
```

Quantifiers

→ These are used to find no. of occurence of a given match

a → Exactly one "a"

a+ → Atleast one "a"

a* → any no. of "a"'s including "0" zero number

a? → atmost "0" or "1" number "a"

a{m,n} ? = 0 or more occurence

m → minimum numbers

n → max no's

Important functions of re module

→ match() :- It is used to check whether given pattern at the beginning of the target string

→
```
import re
match = re.finditer("a{2,3}", "abaabaaab a")
for i in match:
    print(i.start(), "...", i.end(), "...", i.group())
```

o/p: 2 ... 4 ... aa

5 ... 8 ... aaa

match()

```
import re
s = input("enter pattern")
m = re.match("s", "python")
if m != None:
    print("match")
else:
    print("not match")
```

fullmatch()

→ this function is used to match the pattern to the target string (complete string)

search()

this function is used to search given pattern in the given string.

findall():

→ It is used to find all the occurence of the match. It returns list of objects.

sub()

→ This function replaces the target string with every matched pattern

subn()

→ It is exactly same as sub. this returns no. of replacement

startswith(): It is used to check whether given target string starts with out pattern not.

dollar($): It is used to check whether target string ends with given pattern or not.

write a regular expression "a-z, A-Z, 0-9, #"

1) Allowed characters are
2) 1st character should be lower case alphabet from 'a to k'
3) 2nd character should be digit divisible by 3
4) length of identifier should be atleast '2'

~~expression~~ [a-k][0369][a-zA-Z0-9#]*

```
import re
s=input("enter pattern")
m=re.fullmatch("s",[a-k][0369][a-zA-Z0-9#]*")
if m!=None:
    print("match")
else:
    print("not match")
```

output: valid
not match

write a pgm to check whether gmail id is valid or not

```
[w[a-zA-Z0-9]* @gmail[.]com
  import re
  s=input("enter pattern")
  m=re.fullmatch(s,"[w[a-zA-Z0-9]*@gmail[[.]com"]
  if m!=None:
      print("match")
  else:
    print("not match")
```

tkinter module --

By using this module we can develop GUI application in python

→ It is a builtin module

How to install external module into python

By using package manager (pip)

Go to terminal and write \ pip install modulename

ex: pip install pandas

→ By using Tk() class we can create a window component (or) container

→ In tkinter widgets are used to create the controls like label, button, textbox etc.

→ By using mainloop() method it ensures window is visible to the user

```
import tkinter as tk
root = tk.Tk()
root.title("isa")
root.geometry("300x400")
root.mainloop()
```

steps to develop offline applications

1) Import the module (tkinter)
2) Create the window by using Tk() class
3) Create the widgets (components)
4) place the widgets on the window
⇒ By using pack(), grid(), place() methods make components visible to the user

6) write the events for the components

```
from tkinter import *
root = Tk()
root.geometry("400x400")
root.title("ida education")
lbl = Label(root, text = "enter the name", bg = "red",
fg = "yellow", cursor = "hand2", underline = 0, font =
("Arial", 18, "bold")).pack()
pack(pady = 120).
root.mainloop()
```

→ By using text variable we are going to associate the control to object ref var

```
from tkinter import *
root = Tk()
text_var = StringVar()
text_var = set("hello")
lbl = Label (root, textvariable = text_var).pack()
root.mainloop()
```

```
from tkinter import *
root = Tk()
root.geometry("400x500")
def f():
    print("you clicked me")
btn = (root, text = "click me", command = f(), height = 5, width = 10)
.pack()
btn_exit = (root, text = "exit", ~~command root~~ command = root.
destroy, height = 5, width = 10).pack()
root.mainloop()
```

click me

exit

e) username [ ]
password [ ]
[login] [clear]

```
from tkinter import *
root = Tk()  from tkinter import messagebox
root.geometry("400x250")
name_var = StringVar()
passw_var = StringVar()
def submit():
    name = name_var.get()
    password = passw_var.get()
    print("at name", name)
    print("at passw", password)
    name_var.set("")
    passw_var.set("")
def validate_login():
    nameid = name_var.get()
    passid = passw_var.get()
    if nameid = "admin" and passid = "iva":
        messagebox.showinfo("title", "valid user
        -name")
    else:
        messagebox.showinfo("title", "invalid__")
name_label = Label(root, text = "username", textvariable =
    Entry_var)
name_entry = Entry(root, text "password", textvariable =
    name_var)
passw_label = Label(root, text = "password", textvariable
    passw_var)
passw_entry = Entry(root, textvariable = passw_var)
sub_btn = Button(root, text = "clear", command = submit)
login_btn = Button(root, text = "login", command = validate
    login)
name_label.grid(row = 0, column = 0)
name_entry.grid(row = 0, column = 1)
pass_label.grid(row = 1, column = 0)
```

```
pass-entry.grid(row=1, column=1)
sub_btn.grid(row=2, column=0)
login_btn.grid(row=2, column=2)
root.mainloop()
```

Canvas : the Canvas widget is used to draw the Canvas on window

Entry : the entry widget is used to display the single-line text field to the user

Frame : It Can be defined as a Container to which another widget Can be added & organized

Label : A label is a text used to display some message or info. about the other widgets

Listbox : the listbox widget is used to display a list of options to the Customer.

Menubutton : the menubutton is used to display the menu items to the user

Menu : It is used to add menu items to user

Message : the message widget is used to display the message box to the user

Text : It is diff from entry bcz it provides a multi-line text field to the user

→ the pack() widget is used to organize widget in a block

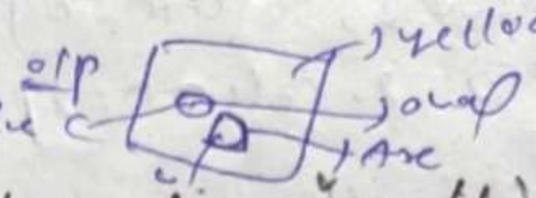
→ the grid() geometry manager organize widget in tabular form

→ the place() geometry manager organize the widget to the specific x & y Co-ordinates

Canvas

```
from tkinter import *
win = Tk()
win.geometry("400x250")
win.title("Example")
c = Canvas(win, bg="yellow", height=250, width=300)
```

```
ovl = c.create_oval(88,30,140,150, fill="blue")
arc = c.create_arc(180,150,88 200, fill="white")
c.pack()
win.mainloop()
```



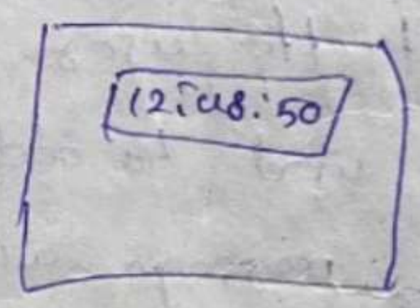
Crate a digital clock by using tkinter

```
from tkinter import *
from time import strftime
win=Tk()
win.geometry("400x250")
win.title("example")

lbl=Label(win,font=("Arial",18,"bold"),bg="blue",
    fg="white")

def time():
    string = strftime('%H,%M,%S %p')
    lbl.config(text=string)
    lbl.after(1000,time)

lbl.pack()
time()
win.mainloop()
```



<u>checkbox</u>

checkbox is used to select multiple options from the user

```
from tkinter import *
win=Tk()
win.geometry("400x250")
chk=Checkbutton(win,text="python",onvalue=1,
    offvalue=0).pack()

chk2=Checkbutton(win,text="AI",onvalue=1,offvalue=0):
    pack()
if onvalue==1
    print("checkbox is selected")

elif onvalue==0
    print("checkbox is not selected")
else:
    print("")
```

```
chk1_var = IntVar()
def chck_clicked():
    if chk1_var.get()==1:
        print("checkbox is selected")
    else:
        print("It is not selected")

chk = Checkbox(win, text="python", command= check_
 clicked(), variable = chk1_var, onvalue=1, offvalue=0).
 pack()
win.mainloop()
```

listbox → It contains list of items.

→ User can select multiple items from the list box

```
from tkinter import *
win = Tk()
win.geometry("400x250")
listbox = Listbox(win, selectmode=MULTIPLE)
listbox.insert(1, "AI")
listbox.insert(2, "DL")
listbox.insert(3, "DS")
listbox.insert(4, "Java")
listbox.insert(5, "python")
def select_item():
    for i in listbox.curselection():
        print(listbox.get(i))
btn = Button(win, text="click me", command = select_item())
    .pack(side="bottom")
listbox.pack()
win.mainloop()
```

Pillow module

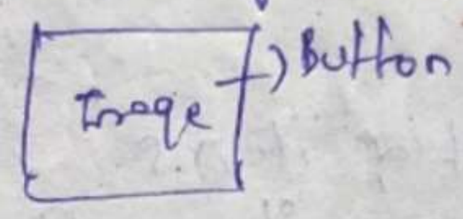
→ By using pillow module we can handle the images

→ It is an external module: It can be downloaded by using pacakgemanage

Prog 66

→ load the image & extract the image on window by using PhotoImage

```
from tkinter import *
from PIL import. Image, ImageTk
win = TK()
win.geometry("400x250")
bg = ImageTk.PhotoImage(file="dove.jpg")
btn = Button(win, image=bg, height=200).pack(side=
"Bottom")
win.mainloop.
```



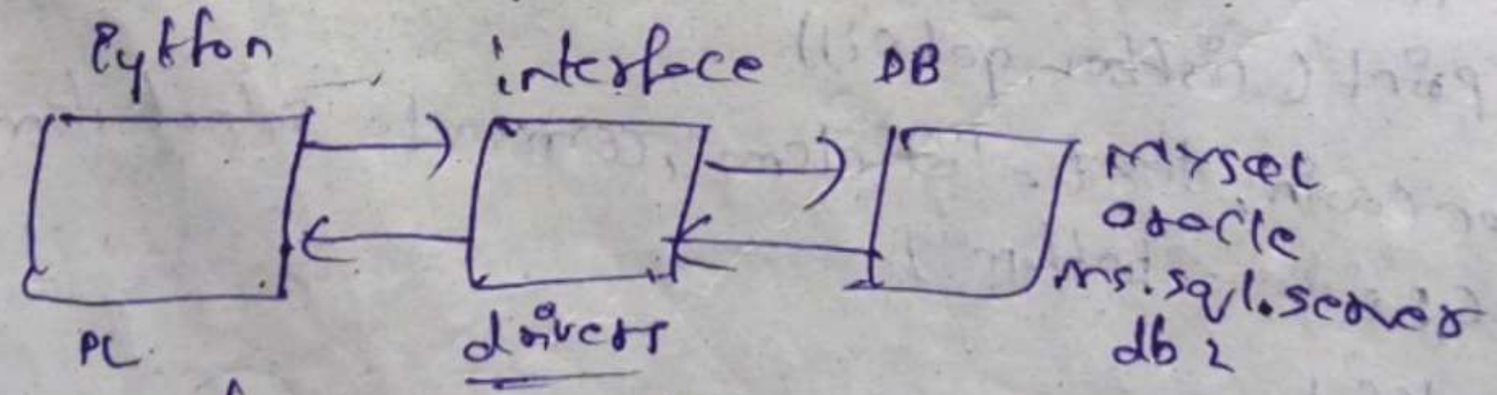
Python DB Connectivity

There are 2 types of storage units:

1) Temporary storage unit (list, variables, arrays, etc)
2) permanent storage unit (files, DB, Bigdata, cloud, Computing)

→ In the file system we are having the following drawbacks:

1) we can't store huge amount of data
2) Data will be stored in the format of unstructured data
3) we can't provide security in the file system
4) Retrieving of info. is difficult
5) we can't backup & restore



oracle ← cx_Oracle
MYSQL ← mysql.connector } → modules

→ every DB has a separate module to interact with DB

steps to Connect to the DB:-

1) import the module

ex: import mysql.connector

2) Establish the connection to the DB by using connect() function

ex: con = mysql.connector.connect(host='localhost', user='root', password='root', database='mydb')

3) Execute sql queries & hold the result by using cursor object → cursor = con.cursor()

```
import mysql.connector
con = mysql.connector.connect(host='localhost', user='root', password='root', database='mydb')
if con != None:
    print("success")
else:
    print("not success")
```

4) By using execute() method we can execute sql queries by using cursor object

1) Execute("sql query") → it is used to execute single

2) execute script("sql queries") → it is used to execute a string of sql queries separated by semicolon

3) execute many() → it is used to execute parameterized queries.

ex: cursor.execute("select * from emp")

5) Commit() and Rollback()

→ In case of DML operations

Commit() → save the changes to DB

rollback() → rollback the last transaction

6) fetch the records from cursor object in case of select queries

→ fetchone() → to fetch one record

→ fetchall() → to read all records / rows & returns list of objects

→ fetchmany(n) → to fetch first "n" rows

7) close the Resources like cursor & connection

```
import mysql.connector
try:
con = mysql.connector.connect(host='localhost'
user='root', password='root', Database='trydb')
cursor = con.cursor()
cursor.execute("Create table employee (empno
int(5), ename varchar(10))")
print("Table Created")
except mysql.connector.DatabaseError as e:
    print(e)
finally:
  con.close()
```

~~con~~

write a pgm insert values into emp DB from python pgm

```
cursor.execute("insert into emp values (195,
'Vishwa')")
con.commit()
```

write a pgm to insert multiple records into emp table by using execute many()

```
sql = "insert into emp values (%s,%s)"
records = [(10,'ira'),(20,'ira2'), (30,'ira3')]
cursor.executemany(sql, records)
```

write a pgm to insert multiple records into employee dynamically by using i/p function

```
import mysql.connector
try:
con=mysql.connector.Connect(host='localhost',user=
'root',password='root',database='mydb')
cursor=con.cursor()
while True:
    empno=int(input("enter empno"))
    empname=input("enter empname")
    sql="insert into employee Values(%d,'%s')"
    cursor.execute(sql %(empno,empname))
    print("Inserted Success")
    option=input("enter do you want to continue(Yes/No)")
    if option=="No":
        break
except mysql.connector.DatabaseError ape:
    print(e)
finally:
    con.close()
```

write a pgm to fetch all the records from the emp table by using fetchone() method

```
import mysql.connector
try:
con=mysql.connector.Connect(host='localhost',user='root',
password='root',Database='mydb')
cursor=con.cursor().
cursor.execute("select * from employee")
row=cursor.fetchone()
while row in not None:
        print(row)
        row=cursor.fetchone()
except mysql.connector.DatabaseError as e:
    print(e)
finally:
    con.close().
```

write a pgm to display all ~~pro~~ records from emp table by using fetchall() method.

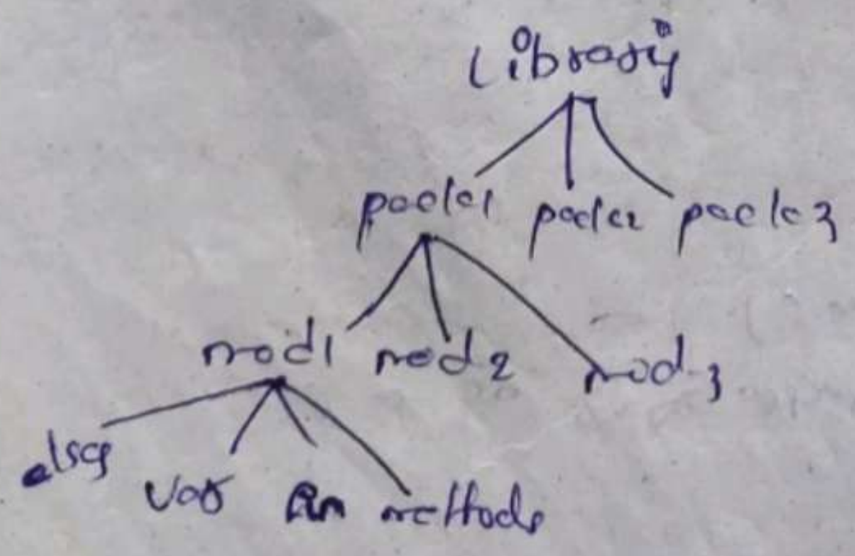
```
import mysql.connector
con = mysql.connector.connect(host = 'localhost',
  user = 'root', password = 'root', Database = 'mydb')
cursor = con.cursor()
cursor.execute("select * from employee")
row = cursor.fetchall()
for data in row:
    print("employee no=", data[0])
    print("employee name =", data[1])
    print()
    print()
except mysql.connector.DBError as e :
```

O/P: employee no = 190
employee name = ira

write a pgm to fetch 1st 2 rows from emp table

```
cursor.execute("select * from employee")
n = int(input("enter no. of rows"))
row = cursor.fetchmany(n)
for data in row:
    print(data)
```

Packages



emp table

st',
rydb')

→ It is an encapsulation mechanism which is group the similar modules into single unit any folder (or) directory which contains __init__.py it is considered as a package

→ A package can contains sub package also

Pack1
- __init__.py → empty file
- module1.py
- text.py

module2.py

```
def f1():
    print("praram walcum")
```

~~text.py~~

log files

→ we will store all the application flow
→ log files are used for debugging ~~error~~ purpose
→ these are the transaction files

logging levels

→ Based on the type of data we are passing logging data is divided into "6" levels :-

1) critical 2) error 3) warning 4) info 5) debug 6) Notset

1) Critical → Represents very serial problem
2) Error : Represents serious error
3) warning : Represents a warning given to the programmer
4) info : Represents a msg with some important info
5) debug : Represents a msg with debugging information
6) Notset :

→ python support builtin module i.e., logging module to perform logging operations
→ we have to store all the info. in a file & set the level by using basicconfig() function.

syntax :

```
logging.basicConfig(filename='abc.txt', level=logging.
                    WARNING)
```

```
import logging
logging.basicconfig(filename='abc.txt', level=logging.
WARNING)
logging.critical("It is critical")
logging.error("It is error")
logging.debug("It is debug")
logging.warning("It is warning")
```

write a pgm save exception info. inside log file

```
import logging
logging.basicconfig(filename='abc.txt', level=logging.INFO)
logging.info("A new request come")
try:
    i=int(input("enter 1st no"))
    j=int(input("enter 2nd no"))
    div=i/j
    print(div)
except ZeroDivisionError as e:
        print("2nd no can't be zero")
        logging.exception(e)
except ValueError as e:
        print("pass correct int value")
        logging.exception(e)
logging.info("process completed")
```

<u>Zipping</u> & <u>Unzipping</u> files

→ python support inbuilt module i.e., zipfile to perform zipping & unzipping operations.
→ It improves memory utilization
→ It decreases transportation time
→ It improves performance
→ By using ~~zipfile class object~~ zipfile() class object we create zipfile, mode, constant (ZIP_DEFLATED)

→ Once we created zipfile object write the files by using write method

```
from zipfile import *
f = zipfile("files.zip", "w", ZIP_DEFLATED)
f.write("a.jpg")
f.write("abc.txt")
f.write("Dove.jpg")
f.close()
print("success created")
```

Unzipping

→ ZIP_STORED Represents unzip operations

→ we can get all file names by using namelist() method

```
from zipfile import *
f = zipfile("files.zip", "r", ZIP_STORED)
names = f.namelist()
for i in names:
    print("file name = ", i)
    f1 = open(i, "r")
    data = f1.read()
    print(data)
    print()
    print()
```

~~f1 = open(i, "r")~~ ~~f1.read()~~

reading data of all files & displaying

Synchronization

→ To avoid deadlocks we use synchronization

Lock

→ lock object is acquired by only one thread at a time even owner thread can't acquire the lock multiple times

→ Recursive functions & Nested accessible are not suitable

Rlock

→ Rlock object is acquired by one thread at a time but owner thread can acquire the lock multiple times

→ It is best suitable for recursive functions

Semaphore :

→ By using semaphore we can execute multiple threads at a time

- syntax

s = semaphore (counter)