PYTHON

An Introduction to PYTHON

- Python is a simple, easy to learn, powerful, high level and object-oriented programming language.
- Python is an interpreted scripting language also.
- Python was developed by Guido Van Rossum and Released in 1991.
- **Python** is a general purpose, dynamic, high level and interpreted programming language.
- It supports Object Oriented programming approach to develop applications.
- It is simple and easy to learn and provides lots of high-level data structures.
- Python is *easy to learn* yet powerful and versatile scripting language which makes it attractive for Application Development.
- Python supports *multiple programming pattern*, including object oriented, and functional or procedural programming styles.
- We don't need to use data types to declare variable because it is *dynamically typed* so we can write a=10 to assign an integer value in an integer variable.

Features of PYTHON

1. Easy to Learn and Use

Python is easy to learn and use. It is developer-friendly and high level programming language.

2. Expressive Language

Python language is more expressive means that it is more understandable and readable.

3. Interpreted Language

Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and its suitable for beginners.

4. Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.

5 .Free and Open Source

Python language is freely available at offical web address.www.python.org/downloads.

The source-code is also available. Therefore it is open source.

6. Object-Oriented Language

Python supports object oriented language and concepts of classes and objects come into existence.

7. Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code.

8. Large Standard Library

Python has a large and broad library and provides rich set of module and functions for rapid application development.

9. GUI Programming Support

Graphical user interfaces can be developed using Python.

10. Integrated

It can be easily integrated with languages like C, C++, JAVA etc.

Versions of PYTHON

- **Python 0.9.0 - February 20, 1991**
  - Python 0.9.1 - February, 1991
  - Python 0.9.2 - Autumn, 1991
  - Python 0.9.4 - December 24, 1991
  - Python 0.9.5 - January 2, 1992
  - Python 0.9.6 - April 6, 1992
  - Python 0.9.8 - January 9, 1993
  - Python 0.9.9 - July 29, 1993
- **Python 1.0 - January 1994**
  - Python 1.2 - April 10, 1995
  - Python 1.3 - October 12, 1995
  - Python 1.4 - October 25, 1996
  - Python 1.5 - December 31, 1997
  - Python 1.6 - September 5, 2000
- **Python 2.0 - October 16, 2000**
  - Python 2.1 - April 15, 2001
  - Python 2.2 - December 21, 2001
  - Python 2.3 - July 29, 2003
  - Python 2.4 - November 30, 2004
  - Python 2.5 - September 19, 2006
  - Python 2.6 - October 1, 2008
  - Python 2.7 - July 4, 2010
- **Python 3.0 - December 3, 2008**
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012
  - Python 3.4 - March 16, 2014
  - Python 3.5 - September 13, 2015
  - Python 3.6 - December 23, 2016
  - Python 3.7 - June 27, 2018

History of PYTHON

- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at National Research Institute in Netherland.
- In February 1991, van Rossum published the code
- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
  - ABC language.
  - Modula-3

➢ Using Python we can Develop the Following Applications:

- ❖ Web Applications
- ❖ Desktop GUI Applications
- ❖ Network Programming
- ❖ Gaming Applications
- ❖ Data Analysis Applications
- ❖ Console Based Applications
- ❖ Business Applications
- ❖ Audio and Video Based Applications

❖ Steps To Install Python For Windows:

Go to Google

|

Type python download for windows

|

Click on Python.org

|

Click on Download python 3.7.0

|

It will be download python 3.7.0.Exe

|

Open Python 3.7.0.Exe

|

Click on Install Now

|

Click On Customize Installation

|

Click on Next

|

Activate the Checkbox Add Python to Environment variables

|

Click on Install

Different Ways to Execute Python Code:

1. **Using Interactive Mode**
2. **Using Script Mode**
3. **Using Python IDLE**
4. **Using Pycharm(IDE)**

Interpreter vs Compiler:

Interpreter	**Compiler**
1. It will check line by line and executes	1. It will check Whole program at a time
2. It gives the result line by line	2. It gives whole output at a time
3. If any error occurs interpreter stops Hence it shows only one error	3.It Checks all statements in the program and show all errors in program
4. It will not generate executable file	4. If no errors in the program then it generates executable file
5. It always executes only source code Ex: Html,Perl,Javascript,Python	5. It executes exe file Ex: C,C++,C#,Java

Python Indentation:

- Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.
- A code block (body of a function, loop etc.) starts with indentation and ends with the first un indented line. The amount of indentation is up to you, but it must be consistent throughout that block.
- Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
a=10
if a==10:
    print("welcome to durgasoft")
```

- The indentation in Python makes the code look neat and clean.
- Indentation can be ignored in line continuation.
- But it makes the code more readable.

Python Comments:

- Comments are very important while writing a program.
- Python Interpreter ignores comment.
- Python supports two types of comments:

Single line comment:

- In case user wants to specify a single line comment, then comment must start with (#)

Multi line comment:

- If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line.
- Multi lined comment can be given inside triple quotes.

Identifiers:

- A Name in python program is called identifier.
- It can be a class name or function name or variable name.
- **Rules to define identifier:**
- Alphabet symbols(either lowercase or uppercase)
- Digits(0 to 9)
- Underscore symbol(_)
- Identifier should not starts with digit.
- Identifiers are case sensitive
- We can not use keywords as Identifier.
- If Identifier starts with Underscore then it is private.

Python Variables:

- Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value.
-
- In Python, we don't need to specify the type of variable because Python is a dynamically typed to get variable type.
- Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.
- It is recommended to use lowercase letters for variable name.

Multiple Assignment:

- Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment
- We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables.

➢ **Assigning single value to multiple variables**

```
a=b=c=10

print (a)

print (b)

print (c)
```

➢ **Assigning Multiple values to multiple variables**

```
a,b,c=10,20,30

print (a)

print (b)

print (c)
```

- The values will be assigned in the order in which variables appears.

Python Keywords:

- Python Keywords are special reserved words which convey a special meaning to the interpreter.
- Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

True	False	None	and	as
Asset	def	class	continue	break
Else	finally	elif	del	except
Global	for	If	from	import
Raise	try	Or	return	pass
Nonlocal	in	not	is	lambda
Yield	while	with		

Python Data Types:

- Data type represent the type of data present inside a variable.
- Every value in Python has a data type.
- In python not required to specify the type explicitly .
- Based on value, the type will be assigned automatically.
- Python is Dynamically Typed Language

  Python Contain the following Data types:

  1. None

2. Numeric
3. List
4. Tuple
5. Set
6. String
7. Range
8. Dictionary or Mapping

- **None:** When we have a variable which is not assigned any value is called None.
- Normally in any language the keyword can be use null , but in python we use None.
- **Numeric:** It classified Four Types
    1. Int
    2. Float
    3. Complex
    4. Bool

    Ex:

```
>>> num=3.4
>>> type(num)
<class 'float'>
>>> num=5
>>> type(num)
<class 'int'>
>>> num=3+4j
>>> type(num)
<class 'complex'>
```

To Convert From Float to int:

```
>>> a=5.8
>>> b=int(a)
>>> type(b)
<class 'int'>
>>> b
5
>>>
```

To Convert from int to float:

```
>>> a=6
>>> b=float(a)
>>> type(b)
<class 'float'>
>>> b
6.0
>>>
```

To Create complex number:

```
>>> a=3
>>> b=5
>>> c=complex(a,b)
>>> c
(3+5j)
>>>
```

Bool:

```
>>> a=10;

>>> b=20;

>>> a<b

True

>>> bool=a<b

>>> bool

True

>>>


>>> int(True)

1

>>> int(False)

0

>>>
```

List,Tuple,Set,string,Range are Sequence.

List: List is an ordered sequence of items.

It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

```
>>> lst=[34,56,78,99]

>>> type(lst)

<class 'list'>

>>> lst

[34, 56, 78, 99]
```

Tuple:

Tuple is an ordered sequence of items same as list.

The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

```
>>> t=(23,45,23,55)

>>> type(t)

<class 'tuple'>

>>> t

(23, 45, 23, 55)
```

SET:

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
>>> s={1,2,3,4,5}

>>> type(s)

<class 'set'>

>>> s

{1, 2, 3, 4, 5}
```

String:

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, `'''` or `"""`

```
>>> str="durga"
>>> type(str)
<class 'str'>
>>> str
'durga'
```

String Slicing:

Slice means a piece.

[] Operator is called slice operator,which can be used to retrieve part of string

In Python string follows o based index.

The index can be either +ve or –ve

+ve index means forward direction i.e from left to right

-ve index means backward direction i.e right to left

-5	-4	-3	-2	-1
M	O	H	A	N
0	1	2	3	4

Ex:

```
>>> s="mohan"
>>> s[0]
'm'
>>> s[1]
'o'
>>> s[-3]
'h'
>>> s[5]
```

```
  s[5]

IndexError: string index out of range

>>> s[0:3]

'moh'

>>> s[:3]

'moh'

>>> s*4

'mohanmohanmohanmohan'

>>> len(s)

5

>>>
```

Range:

Range Data type represent sequence of numbers.

The elements are present in Range data type is not modify, i.e Range data type is immutable.

```
>>> range(0,20)

range(0, 20)

>>> list(range(20))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

>>>
```

Dictionary or Maping:

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
>>> d={'mihan':'nokia','manoj':'samsung','kiran':'oneplusone'}

>>> type(d)

<class 'dict'>

>>> d

{'mihan': 'nokia', 'manoj': 'samsung', 'kiran': 'oneplusone'}

>>> d.keys()

dict_keys(['mihan', 'manoj', 'kiran'])

>>> d.values()

dict_values(['nokia', 'samsung', 'oneplusone'])

>>> d['mihan']

'nokia'

>>> d.get('kiran')

'oneplusone'

>>>
```

Python Operators:

Operator is a symbol that perform certain operation.

EX: 2+3=5 ----- + ,= is operator and 2,3 are Operands.

Python supports the following operators

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.

7. Bitwise Operators.

Arithmetic Operators:

Operator	**Description**
//	It perform floor division
+	It perform Addition
-	It Perform subtraction
*	It Perform Multiplication
/	It Perform Division
%	Return remainder after division
**	Perform Exponent or Power Operator

EX:

```
>>> 10+2

12

>>> 10-2

8

>>> 10*2

20

>>> 10/2

5.0

>>> 10%2
```

```
0
>>> 10**2
100
>>> 10//2
5
>>>
```

Relational Operators:

These operators will check relation

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not Equal to

Ex:

```
>>> a=10
>>> b=20
>>> a<b
True
```

```
>>> a>b

False

>>> a<=b

True

>>> a>=b

False

>>> a!=b

True

>>> a=10

>>> b=10

>>> a<=b

True

>>> a>=b

True

>>> a=b

>>> a==b

True

>>>
```

Assignment Operators:

assignment operators that are used to assign values to the variables.

Operator	Description
=	Assignment
/=	Divide and Assign
+=	Add and Assign
-=	Subtract and Assign
*=	Multiply and Assign
%=	Modulus and Assign
**=	Exponent and Assign
//=	Floor division and Assign

Ex:

```
>>> a=10

>>> a+=5

>>> a

15

>>> a-=5

>>> a

10

>>> a*=5

>>> a
```

```
50
>>> a/=2
>>> a
25.0
>>> a%=2
>>> a
1.0
>>> a//=2
>>> a
0.0
>>> a=5
>>> a
5
>>> a**=2
>>> a
25
>>> a//=2
>>> a
12
>>>
```

Logical Operators:

Perform Logical Operations

Operator	**Description**
And	Logical AND(When both conditions are true output will be true)
Or	Logical or(if any one condition is true output will be true).
Not	i.e Reverse , complement the condition

```
EX:
>>> a=6>3 and 5<3
>>> print(a)
False
>>> a=6>3 or 5<3
>>> print(a)
True
>>> a=not(6>4)
>>> print(a)
False
>>>
```

Membership Operators:

Operator	**Description**
In	Return true if variable is available in sequence , else false.
Not in	Return true if variable is not available in sequence , else false.

Ex:

```
a=10
b=20
```

```
list=[10,20,30,40,50];
if (a in list):
    print("a is available in list" )
else:
    print ("a is not available in list"  )
if(b not in list):
    print ("b is not available in list" )
else:
    print ("b is available in list"  )
```

Identity Operators:

Operator	Description
Is	Return true if identity of two operands are same,else false.
Is not	Return true if identity of two operands are not same, else false.

Ex:

```
1.  a=20
2.  b=20
3.  if( a is b):
4.      print("a,b have same identity")
5.  else:
6.      print("a, b are different ")
7.  b=10
8.  if( a is not b):
9.      print (" a,b have different identity")
10. else:
11.     print("a,b have same identity")
```

```
x=input("Enter first num:")
a=int(x)
y=input("Enter Second Num:")
b=int(y)
z=a+b
print(z)



////////

x=int(input("Enter first num:"))
y=int(input("Enter Second Num:"))
z=x+y
print(z)

////

import sys
x=int(sys.argv[1])
y=int(sys.argv[2])
z=x+y
print(z)

////4
result=eval(input("enter your expression:"))
```

```
print(result)
```

Id():

The id() built-in function returns a unique integer which is the identifier of an object in the memory.

We can use id() to check if two variables are using the same object.

```
>>> a=2

>>> id(a)

1361656080


>>> b=3

>>> id(b)

1361656096

>>> id(a)==id(b)

False

>>> id(a)!=id(b)

True

>>> c=2

>>> c

2

>>> id(c)

1361656080

>>> id(a)==id(c)

True
```

```
>>> d="mohan"
>>> d
'mohan'
>>> id(d)
55031680
>>> e="Mohan"
>>> e
'Mohan'
>>> id(e)
55032064
>>> id(d)==id(e)
False
>>>
```

Control Statements:

The if statement executes only when specified condition is **true**. We can pass any valid expression into the if parentheses.

There are various types of if statements in Python.

- if statement
- if-else statement
- nested if statement

Syntax:

```
if(condition):
  statements
```

Ex:

a=10

```
if a==10:
        print  "Welcome to Durgasoft"
```

If Else:

The If statement is used to test specified condition and if the condition is true, if block executes, otherwise else block executes.

The else statement executes when the if statement is false.

```
if(condition):
   statements
else:
  statements
```

Ex:

```
year=2000
if year%4==0:
    print  "Year is Leap"
else:
    print "Year is not Leap"
```

Program To Find Even or Odd:

```
x=int(input(print("enter any number:")))
r=x%2
if r==0:
    print("Even Number")
else:
    print("Odd Number")
```

Nested If :

In python, we can use nested If to check multiple conditions.

Python provides **elif** keyword to make nested If statement.

This statement is like executing a if statement inside a else statement.

```
If (condition):
    statements
else :
    statements
     if(condition):
     else:
       statements..
```

EX:

```
a=10
if a>=20:
    print "Condition is True"
else:
    if a>=15:
        print "Checking second value"
    else:
        print "All Conditions are false"
```

```
x=int(input(print("Enter Any Num:")))
if x==1:
    print("One")
elif x == 2:
    print("Two")
elif x == 3:
    print("Three")
elif x == 4:
    print("Four")
elif x == 5:
    print("Five")
else:
    print("Invalid Number")
```

Loops in python:

For Loop:

Python **for loop** is used to iterate the elements of a collection in the order that they appear. This collection can be a sequence(list or string).

for <variable> **in** <sequence>:

```
x=["mohan",20,34.5]
for i in x:
    print(i)
```

To Find Sum of 10 Numbers:

```
sum=0
for n in range(1,11):
    sum+=n
print (sum)
```

```
num=2
for a in range (1,6):
   print  num * a
```

Python Nested For Loops

Loops defined within another Loop are called Nested Loops. Nested loops are used to iterate matrix elements or to perform complex computation.

When an outer loop contains an inner loop in its body it is called Nested Looping.

```
num_list = [1, 2, 3]
```

```
char_list = ['a', 'b', 'c']

for n in num_list:

    print(n)

    for l in char_list:

        print(l)
```

Ex:

```
list_of_list = [['coke', 'sprite', 'pepsi'],[1, 2, 3],[1.3, 1.4,1.5]]

for list in list_of_list:

    for item in list:

        print(item)
```

Python While Loop

In Python, while loop is used to execute number of statements or body till the specified condition is true. Once the condition is false, the control will come out of the loop.

Syntax:

```
while <expression>:
        Body
```

```
i=1
while  i<=5:
    print("Hello")
    i=i+1
```

With the `while` loop we can execute a set of statements as long as a condition is true.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
drinks = ["coke", "pepsi", "sprite"]
for x in drinks:
  if x == "pepsi":
    break
  print(x)
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

String DataStructure:

What is string?

A string is a sequence of characters.

Different Ways to Create string:

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

Ex:

```
str = 'Hello'

print(str)

str = "Hello"

print(str)

str = '''Hello'''

print(str)

str = """Hello, welcome to

          durgasoft"""

print(str)
```

String Indexing and String Slicing:

```
str = 'durgasoft'

print(str)

#first character

print(str[0])

#last character

print(str[-1])
```

#slicing 2nd to 5th character

print(str[1:5])

#slicing 6th to 2nd last character

print(str[5:-2])

String Concatenation and String Multiplication:

Joining of two or more strings into a single one is called concatenation.

+ Sign is used to concatenate.

The ***** operator can be used to repeat the string for a given number of times.

EX:

str1 = 'Durga'

str2 ='Soft'

# using +

print(str1 + str2)

# using *

print(str1 * 3)

String Split():

the split() method takes maximum of 2 parameters:

syntax:

str.split([separator [,maxsplit]])

EX:

```
alphabet = "a b c d e f g"

data = alphabet.split() #split string into a list

for temp in data:

    print (temp)
```

String Split+Max Split:

First Two White Spaces Only Split:

```
alphabet = "a b c d e f g"

data = alphabet.split(" ",2) #maxsplit

for temp in data:

    print (temp)
```

String Capitalise():

The capitalize() function doesn't take any parameter.

The capitalize() function returns a string with first letter capitalized. It doesn't modify the old string.

If the first letter of a string is an uppercase letter or a non-alphabetic character, it returns the original string.

EX:

```
string = "durgasoft"

cap = string.capitalize()

print('Old String: ', string)

print('Capitalized string:', cap)
```

String Title():

Title function in python is used to convert the first character in each word to Uppercase and remaining characters to Lowercase in string and returns new string.

```
# Python Title() Method Example


str1 = 'duRga soft is beSt
inStitute'
str2 = str1.title()
print('Output after Title()
method is = ', str2)

# observe the original string
print('Converted String is = ',
str2)
print('Original String is = ',
str1)

# Performing title() function
directly
str3 = 'mOHAN rEDDy'.title()
print('Output after Title()
method is = ', str3)
```

```
str5 = 'a6568'.title()
print('Output after Title()
method is = ', str5)
```

String Count():

count() method returns the number of occurrences of the substring in the given string.

string = "Python is popular and it is good,it is a oop,it is scripting lang"

substring = "is"

count = string.count(substring)

print("The count is:", count)

String Replace():

oldstring = 'I like durgasoft'

newstring = oldstring.replace('like', 'love')

print(newstring)

Upper():

string="python at durgasoft"

print(string.upper())

Lower():

```
string="PYTHON AT DURGASOFT"

print(string.lower())
```

Using "join" function for the string

The join function is a more flexible way for concatenating string. With join function, you can add any character into the string.

For example, if you want to add a colon (:) after every character in the string "Python" you can use the following code.

Ex:

```
print(":".join("Python"))
```

Reversing String

By using the reverse function, you can reverse the string. For example, if we have string "12345" and then if you apply the code for the reverse function as shown below.

```
#string="12345"

string="SAI"

print(''.join(reversed(string)))
```

In Python, Strings are immutable.

```
x = "Durga"

x.replace("Durga","Mohan")

print(x)
```

String Sort():

```
str = "Durgasot is one of the best institute in hyderabad"

words = str.split()

words.sort()

print("The sorted words are:")

for word in words:

   print(word)
```

String Swapcase():

The swapcase() method returns the string where all uppercase characters are converted to lowercase, and lowercase characters are converted to uppercase.

```
string = "DURGASOFTWARE SOLUTIONS,HYDERABAD."
print(string.swapcase())

string = "durgasoftware solutions,hyderabad."
print(string.swapcase())

string = "dUrGasofTwAre sOlUtioNs."
print(string.swapcase())
```

List DataStructure:

What is List:

List is a collection which is in ordered and it can changeable. It Allows duplicate members.

List contain different data type Items.

Nested List:

A List can have another list within it as Item is called Nested List.

Creating a List:

list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

Ex:

```
#List with Integer elements
lst=[10,20,30]
print(lst)
#List with float elements
lst=[10.5,20.5,30.5]
print(lst)
#List with diff datatype
elements
lst=["mohan",20,33.3,'S']
print(lst)
```

```
#Nested List
nlst=["mohan",[3,4,5],['A','B',
'C']]
print(nlst)
```

Access elements from a List:

We can use the index operator [] to access an item in a list.

Index starts from 0. So, a list having 4 elements will have index from 0 to 3.

Trying to access an element out of index this will raise an `IndexError`.

The index must be an integer.

We can't use float or other types, this will result into `TypeError`.

Nested list are accessed using nested indexing.

Ex:

```
lst = ['D','u','r','g','a']
print(lst[0])
print(lst[2])
print(lst[4])
# Nested List
nlst = ["Mohan", [2,0,1,5]]
# Nested indexing
print(nlst[0][1])
print(nlst[1][3])
```

```
#negative index
print(lst[-2])
```

List Slicing:

```
lst =
['d','u','r','g','a','s','o','f
','t']
# elements 3rd to 5th
print(lst[2:5])
# elements beginning to 4th
print(lst[:-5])
# elements 6th to end
print(lst[5:])
# elements beginning to end
print(lst[:])
```

Changing or Adding elements to List:

List are mutable,that means elements can be changed in List,
unlike string or tuple

We can use assignment operator (=) to change an item or a range of items.

Ex:

```
lst = [3, 6, 9, 12]
print(lst)
# change the 1st item
lst[0] = 1
print(lst)
# change 2nd to 4th items
lst[1:4] = [2, 4, 8]
print(lst)
```

Append and Extend Method in List:

We can add one item to a list using `append()` method or add several items using `extend()`method.

Ex:

```
lst = [1, 2, 3]
lst.append(4)
print(lst)
lst.extend([5, 6, 7])
print(lst)
```

List Concatenation and List Multiplication :

We can use + operator to combine two lists. This is also called concatenation.

The * operator repeats a list for the given number of times.

```
lst = [1, 2, 3]
print(lst + [7, 8, 9])
print(["AB"] * 4)
print(lst*3)
```

Insert Method in List:

we can insert one item at a desired location by using the method `insert()`

```
EX:
lst = [1, 9]
lst.insert(1,3)
print(lst)
lst[2:2] = [5, 7]
print(lst)
```

Delete or Remove Elements from List:

We can delete one or more items from a list using the keyword `del`. It can even delete the list entirely.

```
lst =
['d','u','r','g','a','s','o','f
','t']
# delete one item
del lst[2]
print(lst)
# delete multiple items
del lst[1:5]
print(lst)
# delete entire list
del lst
```

```
# Error: List not defined
print(lst)
```

We can use `remove()` method to remove the given item or

`pop()` method to remove an item at the given index.

The `pop()` method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the `clear()` method to empty a list.

```
lst =
['d','u','r','g','a','s','o','f
','t']
lst.remove('d')
print(lst)
print(lst.pop(1))
print(lst)
print(lst.pop())
print(lst)
lst.clear()
print(lst)
```

List Sort():

The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

Ex:

```
# vowels list
vowels = ['e', 'a', 'u', 'o',
'i']
# sort the vowels
vowels.sort()
# print vowels
print('Sorted list:', vowels)
```

sort() method accepts a `reverse` parameter as an optional argument.

```
list.sort(reverse=True)
```

Alternately for sorted(), you can use the following code.

```
sorted(list, reverse=True)
```

```
# vowels list
vowels = ['e', 'a', 'u', 'o',
'i']

# sort the vowels
vowels.sort(reverse=True)

# print vowels
```

```
print('Sorted list (in
Descending):', vowels)


List Copy():

old_list = [1, 2, 3]
new_list = old_list

print('Old List:', old_list )
# add element to list
new_list.append('a')

print('New List:', new_list )

List Count():
# vowels list
vowels = ['a', 'e', 'i', 'o',
'i', 'u']

# count element 'i'
count = vowels.count('i')

# print count
```

```
print('The count of i is:',
count)

# count element 'p'
count = vowels.count('p')

# print count
print('The count of p is:',
count)
```

List Index():

```
# vowels list
vowels = ['a', 'e', 'i', 'o',
'i', 'u']

# element 'e' is searched
index = vowels.index('e')

# index is printed
print('The index of e:', index)

# element 'i' is searched
index = vowels.index('i')
```

```
# only the first index of the
element is printed
print('The index of i:', index)
```

Creating List from User Values:

```
our_list = []  # create empty
list

first_num = int(input('Enter
first number: '))
second_num = int(input('Enter
second number: '))
third_num = int(input('Enter
third number: '))

our_list.append(first_num)
our_list.append(second_num)
our_list.append(third_num)

print(our_list)
```

Creating List Using Range Function:

```
# empty range
print(list(range(0)))

# using range(stop)
print(list(range(10)))

# using range(start, stop)
print(list(range(1, 10)))
```

What is tuple?

In Python programming, a tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

Creating a Tuple

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. The parentheses are optional but is a good practice to write it.

A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

Ex:

```
# empty tuple
tpl = ()
print(tpl)
# tuple with integers
tpl = (1, 2, 3)
print(tpl)
# tuple with different
datatypes
tpl = (9, "mohan", 3.4)
print(tpl)
# nested tuple
tpl = ("mohan", (8, 4, 60), (1,
2, 3))
print(tpl)
# tuple creating without
brakets is called tuple packing
tpl = 5, 7.8, "hai"
print(tpl)
```

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple.

```
# only parentheses is not
enough
tpl = ("hello")
print(type(tpl))
# need a comma at the end
tpl = ("hello",)
print(type(tpl))
# parentheses is optional
tpl = "hello",
print(type(tpl))
```

Accessing Elements in a Tuple

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0.

```
tpl = 
('d','u','r','g','a','s','o','f
','t')
print(tpl)
print(tpl[2])
print(tpl[5])
# nested tuple
ntpl = ("durga", (7, 8, 6), (1, 
2, 3))
print(ntpl[0][3])
print(ntpl[1][1])
print(ntpl[2][1])
#negative indexing
print(tpl[-4])
print(tpl[-5])
```

Tuple Slicing:

```
tpl =
('d','u','r','g','a','s','o','f
','t')
print(tpl)
print(tpl[1:2])
print(tpl[:4])
print(tpl[2:])
print(tpl[:])
```

Tuple Immutable:

Unlike lists, tuples are immutable

This means that elements of a tuple cannot be changed once it has been assigned

```
tpl=(10,20,30)
print(tpl)
tpl[1]=33
print(tpl)
```

Tuple Concatenation and Multplication:

We can use + operator to combine two tuples. This is also called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the * operator.

```
# Concatenation
print((1, 2, 3) + (4, 5, 6))

# Multiplication
print(("durga",) * 5)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword del.

```
tpl=('d','u','r','g','a')
print(tpl)
del tpl
print(tpl)
```

Tuple Methods():

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Count and index:

```
tpl = ('b','a','n','a','n','a')

print(tpl.count('n'))
print(tpl.index('a'))
```

Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword `in`.

```
tpl = ('b','a','n','a','n','a')
print('b' in tpl)
print('i' in tpl)
print('i' not in tpl)
```

Len(): The len() function returns the number of items in Tuple.

```
tpl =
('b','a','n','a','n','a',)
length=len(tpl)
print(length)
```

Max():

```
print('Maximum is:', max(1, 3,
2, 5, 4))
tpl = (1, 3, 2, 8, 5, 10, 6)
print('Maximum is:', max(tpl))
```

Min():

```
print('Minimum is:', min(1, 3,
2, 5, 4))
tpl = (1, 3, 2, 8, 5, 10, 6)
print('Minimum is:', min(tpl))
```

Sum():

```
tpl = [2.5, 3, 4, -5]
# start parameter is not
provided
Totalsum = sum(tpl)
print(Totalsum)
# start = 6
Totalsum = sum(tpl, 6)
print(Totalsum)


# Converting a string into
tuple
str="abcdef"
print(str)
print(type(str))

tpl=tuple(str)
```

```
print(tpl)
print(type(tpl))

# Converting a list into tuple
lst=[12,"mohan",45.45,100,[1,2,
3]]
print(lst)
print(type(lst))

tpl=tuple(lst)
print(tpl)
print(type(tpl))


# Converting a tuple into
string
tpl=(10,20,30,40,50)
print(tpl)
print(type(tpl))

string=str(tpl)
print(string)
print(type(string))
```

What is set:

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc

Creating set:

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function `set()`.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.).

EX:

```
# set of integers
myset = {1, 2, 3}
print(myset)
# different datatypes
myset = {1.0, "Hello", (1, 2,
3)}
print(myset)
#no duplicates in set
myset = {1,2,3,4,3,2}
print(myset)
```

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the `set()` function without any argument.

```
a = {}
print(type(a))
a = set()
print(type(a))
```

How to change a set in Python?

Sets are mutable. But since they are unordered, indexing have no meaning.

We cannot access or change an element of set using indexing or slicing. Set does not support it.

We can add single element using the `add()` method and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
myset = {1,3}
print(myset)
# add an element
myset.add(2)
print(myset)

# add multiple elements
myset.update([2,3,4])
```

```
print(myset)

# add list and set
# Output: {1, 2, 3, 4, 5, 6, 8}
myset.update([4,5], {1,6,8})
print(myset)
```

How to remove elements from a set?

A particular item can be removed from set using methods, `discard()` and `remove()`.

The only difference between the two is that, while using `discard()` if the item does not exist in the set, it remains unchanged. But `remove()` will raise an error in such condition.

```
# initialize my_set
myset = {1, 3, 4, 5, 6}
print(myset)
# discard an element
myset.discard(4)
print(myset)
# remove an element
myset.remove(6)
print(myset)

# discard an element
# not present in myset
# Output: {1, 3, 5}
myset.discard(2)
print(myset)
#myset.remove(2)
#clear set
myset.clear()
print(myset)
```

Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

Set Union

Union of *A* and *B* is a set of all elements from both sets.

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
# use | operator
print(A | B)
#use union function
print(A.union(B))
```

Set Intersection

Intersection of `A` and `B` is a set of elements that are common in both sets.

Intersection is performed using `&` operator. Same can be accomplished using the method `intersection()`.

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
#print(A & B)
#use intersection method
print(A.intersection(B))
```

Set Difference

Difference of `A` and `B` (`A` - `B`) is a set of elements that are only in `A` but not in `B`. Similarly, `B` - `A` is a set of element in `B` but not in `A`.

Difference is performed using `-` operator. Same can be accomplished using the method `difference()`.

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use - operator on A
# Output: {1, 2, 3}
#print(A - B)
#use difference()
print(A.difference(B))
```

Set Symmetric Difference

Symmetric Difference of `A` and `B` is a set of elements in both `A` and `B` except those that are common in both.

Symmetric difference is performed using `^` operator. Same can be accomplished using the method `symmetric_difference()`.

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use ^ operator
#print(A ^ B)
#use symanticdifference
print(A.symmetric_difference(B)
)
```

ADD():

It will add an element to set
The add() method doesn't add an element to the set if it's already present in it.

```
# set of vowels
vowels = {'a', 'e', 'i', 'u'}
# adding 'o'
vowels.add('o')
print('Vowels are:', vowels)
# adding 'a' again
vowels.add('a')
print('Vowels are:', vowels)
```

Set membership test:

```
# initialize my_set
myset = set("apple")
print('a' in myset)
print('p' not in myset)
```

Len():

```
myset={10,20,30,40,50,60}
print(len(myset))
```

Max and Min():

```
# using max(iterable)
myset = {1, 3, 2, 8, 5, 10, 6}
print('Maximum is:',
max(myset))

# using min(iterable)
myset = {1, 3, 2, 8, 5, 10, 6}
print('Maximum is:',
min(myset))
```

Sum():

```
myset = {2.5, 3, 4, -5}

# start parameter is not
provided
Total = sum(myset)
print(Total)

# start = 7
Total = sum(myset, 7)
print(Total)
```

What is Dictionary:

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.

Creating Dictionary:

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

EX:

```
# empty dictionary
mydict = {}
print(mydict)
# dictionary with integer keys
mydict = {1: 'mohan', 2:
'durga'}
print(mydict)
# dictionary with mixed keys
mydict = {'name': 'sai', 1: [3,
4, 5]}
print(mydict)
# using dict()
mydict = dict({1:'mohan',
2:'sai'})
```

```
print(mydict)
# from sequence having each
item as a pair
mydict = dict([(1,'raj'),
(2,'ram')])
print(mydict)
```

Accessing elements from Dictionary:

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the `get()` method.

The difference while using `get()` is that it returns `None` instead of `KeyError`, if the key is not found.

```
mydict = {'name':'manoj',
'age': 45}
print(mydict['name'])
print(mydict.get('age'))

# Trying to access keys which
doesn't exist throws error
print(mydict.get('address'))
print(mydict['address'])
```

Change or Add elements in Dictionary:

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

Ex:

```
mydict = {'name':'Jack', 'age':
26}
# update value
mydict['age'] = 27
print(mydict)
# add item
mydict['address'] = 'Hyderabad'
print(mydict)
```

delete or remove elements from a dictionary?

We can remove a particular item in a dictionary by using the method `pop()`. This method removes as item with the provided key and returns the value.

The method, `popitem()` can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the `clear()` method.

We can also use the `del` keyword to remove individual items or the entire dictionary itself.

```
# create a dictionary
mydict = {1:1, 2:4, 3:9, 4:16,
5:25}
```

```
# remove a particular item
print(mydict.pop(4))
print(mydict)
#remove an arbitrary item
# Output: (1, 1)
print(mydict.popitem())
print(mydict)
# delete a particular item
del mydict[5]
print(mydict)
# remove all items
mydict.clear()
# Output: {}
print(mydict)
# delete the dictionary itself
del mydict
# Throws Error
print(mydict)
```

Copy():

```
original = {1:'one', 2:'two'}
new = original.copy()
```

```
print('Orignal: ', original)
print('New: ', new)
```

Items:

```
# random sales dictionary
sales = { 'apple': 2, 'orange':
3, 'grapes': 4 }
print(sales.items())
```

Keys:

```
person = {'name': 'mohan',
'age': 34, 'salary': 80000}
print(person.keys())
```

Values:

```
person = {'name': 'mohan',
'age': 34, 'salary': 80000}
print(person.values())
```

Membership Test:

```
mydict = {1: 1, 3: 9, 5: 25, 7:
49, 9: 81}
print(1 in mydict)
print(2 not in mydict)
# membership tests for key only
not value
print(49 in mydict)
```

Len()

```
mydict = {1: 1, 3: 9, 5: 25, 7:
49, 9: 81}
print(len(mydict))
```

What is Function:

If a group of statements is repeatedly required then it is not recommended to write these statements every time separately.

.We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

function is a group of related statements that perform a specific task.

Functions help break our program into smaller and modular chunks.

As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes code reusable

The main advantage of functions is code Reusability.

Note: In other languages functions are known as methods,procedures,subroutines etc

Python supports 2 types of functions

1. Built in Functions

2. User Defined Functions

1. Built in Functions:

The functions which are coming along with Python software automatically, are called built in functions or pre defined functions

Eg: id()

type()

input()

eval()

Len()

etc..

2. User Defined Functions:

The functions which are developed by programmer explicitly according to business requirements , are called user defined functions.

Syntax to create user defined functions:

```
def    function_name(parameters) :
 """ doc string"""     ----     -----
```

Statements...

return value

Above shown is a function definition which consists of following components.

1. Keyword `def` marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

Note: While creating functions we can use 2 keywords

1. def (mandatory)

2. return (optional)

Eg 1: Write a function to print Hello

```
def fun():
    print("Hello")
fun()
fun()
fun()
```

Parameters:

Parameters are inputs to the function. If a function contains parameters,then at the time of calling,compulsory we should provide values otherwise we will get error.

Eg: Write a function to take name of the student as input and print wish message by name.

```
def fun(name):
    print("Hello",name,"good morning")
fun("mohan")
fun("sai")
```

Eg: Write a function to take number as input and print its square value

```
def square(num):
    print("Square value of ",num, "is",
num*num)
square(4)
square(3)
```

Return Statement:

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

Write a function to accept 2 numbers as input and return sum:

```
def add(x,y):
    return  x+y
result=add(10,20)
print("sum is:",result)
```

```
def add(x,y):
   return  x+y
print(add(30,20))
```

If we are not writing return statement then default return value is None

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

```
def myfunction(course =
"Python"):
  print("My course is:" +
course)
myfunction("Dotnet")
myfunction("java")
myfunction()
myfunction("Php")
```

Write a function to check whether the given number is even or odd?

```
def even_odd(num):
    if num%2==0:
       print(num,"is Even")
    else:
        print(num,"is Odd")
even_odd(10)
even_odd(9)
```

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def myfunc():
    x = 10
    print("Value inside
function:",x)
x = 20
myfunc()
print("Value outside
function:",x)
```

Program for calculator using Function:

```
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
def multiply(x, y):
    return x * y
def divide(x, y):
    return x / y
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
#  input from the user
choice = input("Enter
choice(1/2/3/4):")
num1 = int(input("Enter first
number: "))
num2 = int(input("Enter second
number: "))
if choice == '1':
    print(num1,"+",num2,"=",
add(num1,num2))
elif choice == '2':
    print(num1,"-",num2,"=",
subtract(num1,num2))
elif choice == '3':
    print(num1,"*",num2,"=",
multiply(num1,num2))
elif choice == '4':
    print(num1,"/",num2,"=",
divide(num1,num2))
else:
    print("Invalid input")
```

Returning Multiple values from a function:

```
def sum_sub(a, b):
      sum = a + b
      sub = a - b
      return sum,sub
x, y = sum_sub(100, 50)
print("The Sum is :", x)
print("The Subtraction is :",
y)
```

Types of arguments :

```
def  f1(a,b):
    ------
   ------
------
  f1(10,20)
```

a,b are formal arguments where as 10,20 are actual arguments

There are 4 types are actual arguments are allowed in Python.

1. positional arguments
2. keyword arguments
3. default arguments
4. Variable length arguments

1. positional arguments:

These are the arguments passed to function in correct positional order.

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

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

The number of arguments and position of arguments must be matched. If we change the order then result may be changed.

If we change the number of arguments then we will get error.

EX:

```
def sub(x,y):
   print("sum is:",x-y)
sub(10,20)
```

2. keyword arguments:

We can pass argument values by keyword i.e by parameter name.

```
def fun(name,msg):
    print("Hello",name,msg)
fun(name="Durga",msg="Good
Morning")
fun(msg="Good
Morning",name="Durga")
```

Here the order of arguments is not important but number of arguments must be matched.

Note:

We can use both positional and keyword arguments simultaneously.
But first we have to take positional arguments and then keyword arguments, otherwise we will get syntaxerror.

```
def  fun(name,msg):
  print("Hello",name,msg)
fun("Durga","GoodMorning")       ==>valid
fun("Durga",msg="GoodMorning")     ==>valid
fun(name="Durga","GoodMorning")   ==>invalid
SyntaxError: positional argument follows keyword argument
```

3. Default Arguments:

Sometimes we can provide default values for our positional arguments

```
def fun(name="Guest"):
    print("Hello",name,"Good Morning")
```

```
fun("Durga")
fun()
```

If we are not passing any name then only default value will be considered.

***Note:

After default arguments we should not take non default arguments

def fun(name="Guest",msg="Good Morning"): ===>Valid
def fun(name,msg="Good Morning"): ===>Valid
 def fun(name="Guest",msg): ==>Invalid

SyntaxError: non-default argument follows default argument

4. Variable length arguments:

Sometimes we can pass variable number of arguments to our function,such type of arguments are called variable length arguments.

We can declare a variable length argument with * symbol   as follows

def f1(*n):

We can call this function by passing any number of arguments including zero number. Internally all these values represented in the form of tuple.

```
def sum(*n):
    total=0
    for n1 in n:
        total=total+n1
    print("The Sum=",total)
sum()
sum(10)
sum(10,20)
sum(10,20,30,40)
```

Python supports 2 types of variables.

1. Global Variables
2. Local Variables

1. Global Variables

The variables which are declared outside of function are called global variables. These variables can be accessed in all functions of that module.

```
a=10 # global variable
def f1():
    print(a)
def f2():
    print(a)
f1()
f2()
```

2. Local Variables:

The variables which are declared inside a function are called local variables. Local variables are available only for the function in which we declared it.i.e from outside of function we cannot access.

```
def f1():
    a=10
    print(a) #valid
def f2():
    print(a) #invalid

f1()
f2()
```

Error: name a is not defined

global keyword:

We can use global keyword for the following 2 purposes:

1. To declare global variable inside function

2. To make global variable available to the function so that we can perform required modifications

```
a=10
def f1():
    a=777
    print(a)
def f2():
    print(a)

f1()
f2()
```

Ex:

```
a=10
def f1():
    global a
    a=777
    print(a)
def f2():
    print(a)

f1()
f2()
```

```
def f1():
    a=10
    print(a)
def f2():
    print(a)

f1()
f2()
```

Error: a is not defined:

Ex:

```
def f1():
    global a
    a=10
    print(a)
def f2():
    print(a)

f1()
f2()
```

Note: If global variable and local variable having the same name then we can access global variable inside a function as follows

```
a=10 #global variable
def f1():
    a=777 #local variable
    print(a)
    print(globals()['a'])
f1()
```

Recursive Functions :
A function that calls itself is known as Recursive Function.

Eg: factorial(3)
=3*factorial(2)
=3*2*factorial(1)
=3*2*1*factorial(0)
=3*2*1*1 =6
factorial(n)= n*factorial(n-1)

The main advantages of recursive functions are:

1. We can reduce length of the code and improves readability
2. We can solve complex problems very easily.

Ex:

```
def factorial(n):
    if n==0:
        result=1
    else:
         result=n*factorial(n-
1)
    return  result
print("Factorial of 4
is:",factorial(4))
print("Factorial of 5
is:",factorial(5))
```

Anonymous Functions:

Sometimes we can declare a function without any name,such type of nameless functions are called anonymous functions or lambda functions.

The main purpose of anonymous function is just for instant use(i.e for one time usage)

Normal Function:

We can define by using def keyword.

```
def  square(n):
 return n*n
```

lambda Function:
We can define by using lambda keyword
lambda n:n*n

Syntax of lambda Function:

lambda   argument_list : expression

Note: By using Lambda Functions we can write very concise code so that readability of the program will be improved.

Write a program to create a lambda function to find square of given number?

EX:

```
s=lambda n:n*n
print("Square of 4 is:",s(4))
print("Square of 5 is:",s(5))
```

Lambda function to find sum of 2 given numbers

```
s=lambda a,b:a+b
print("Sumof 10,20
is",s(10,20))
print("Sumof 100,200
is",s(100,200))
```

Lambda Function to find biggest of given values

```
s=lambda a,b:a if a>b else b
print("Biggest of 10,20
is:",s(10,20))
print("Biggest of 100,200
is:",s(100,200))
```

Note: Lambda Function internally returns expression value and we are not required to write return statement explicitly.

Note: Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(),map() and reduce() functions,b'z these functions expect function as argument

filter() function:

We can use filter() function to filter values from the given sequence based on some condition.

filter(function,sequence)

where function argument is responsible to perform conditional check sequence can be list or tuple or string.

Program to filter only even numbers from the list by using filter() function?

without lambda Function:

```
def isEven(x):
    if x%2==0:
        return True
    else:
        return False
L=[2,6,3,5,8,12,13]
L1=list(filter(isEven,L))
print(L1)
```

with lambda Function:

```
L=[2,3,4,8,9,12,10,15,13]
L1=list(filter(lambda
x:x%2==0,L))
print(L1)
L2=list(filter(lambda
x:x%2!=0,L))
print(L2)
```

map() function:

For every element present in the given sequence, apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Eg: For every element present in the list perform double and generate new list of doubles.

Syntax: map(function,sequence)

The function can be applied on each element of sequence and generates new sequence.

Without lambda:

```
L=[1,2,3,4,5]
def double(x):
    return 2*x
L1=list(map(double,L))
print(L1)
```

with lambda:

```
L=[2,4,6,8]
L1=list(map(lambda x:2*x,L))
print(L1)
```

To find square of given numbers:

```
L=[2,4,6,8]
L1=list(map(lambda x:x*x,L))
print(L1)
```

We can apply map() function on multiple lists also.But make sure all list should have same length.

```
L1=[1,2,3,4]
L2=[2,3,4,5]
L3=list(map(lambda
x,y:x*y,L1,L2))
print(L3)
```

reduce() function:

reduce() function reduces sequence of elements into a single element by applying the specified function.

reduce(function,sequence)

reduce() function present in functools module and hence we should write import statement.

```
from functools import*
L=[10,20,30,40,50]
L1=reduce(lambda x,y:x+y,L)
print(L1)
L2=reduce(lambda x,y:x*y,L)
print(L2)
L3=reduce(lambda x,y:x-y,L)
print(L3)
```

Note:

In Python every thing is treated as object.
Even functions also internally treated as objects only.

```
def f1():
    print("Hello")
print(f1)
print(id(f1))
```

Function Aliasing:

For the existing function we can give another name, which is nothing but function aliasing.

```
def wish(name):
    print("Good Morning",name)
greeting=wish
print(id(wish))
print(id(greeting))

greeting("Mohan")
wish("mohan")
```

Note: In the above example only one function is available but we can call that function by using either wish name or greeting name.   If we delete one name still we can access that function by using alias name

```
def wish(name):
    print("Good Morning",name)
```

```
greeting=wish

greeting("Mohan")
wish("mohan")

del wish
#wish("sai")
greeting("sai")
```

Nested Function:

A Function which contain one more Function within it is known as Nested Function:

```
def multi(a):
    def mul(b):
        def mu(c):
         return a*b*c
        return mu
    return mul
y=multi(10)(20)(2)
print(y)
```

dir():

The `dir()` function returns all properties and methods of the specified object, without the values.

This function will return all the properties and methods, even built-in properties which are default for all object.

Syntax: dir(object)

Ex:

```
l1=[1,2,3,4]
print(dir(l1))
```

Ex:

```
number = [1, 2, 3]
print(dir(number))
print(dir())
```

Format():

The `format()` function formats a specified value into a specified format.

Syntax: format(value,format)

EX:

```
x = format(0.5, '%')
print(x)
```

```
x = format(10000, '_')
print(x)
x = format(10000, 'e')
print(x)
```

Value: a value of any format

format	The format you want to format the value into. Legal values: `'<'` - Left aligns the result (within the available space) `'>'` - Right aligns the result (within the available space) `'^'` - Center aligns the result (within the available space) `'='` - Places the sign to the left most position `'+'` - Use a sign to indicate if the result is positive or negative `'-'` - Use a sign for negative values only `' '` - Use a leading space for positive numbers `','` - Use a comma as a thousand separator `'_'` - Use a underscore as a thousand separator `'b'` - Binary format `'c'` - Converts the value into the corresponding unicode character `'d'` - Decimal format `'e'` - Scientific format, with a lower case e `'E'` - Scientific format, with an upper case E `'f'` - Fix point number format `'F'` - Fix point number format, upper case `'g'` - General format `'G'` - General format (using a upper case E for scientific notations) `'o'` - Octal format `'x'` - Hex format, lower case `'X'` - Hex format, upper case `'n'` - Number format `'%'` - Percentage format

enumerate():

The `enumerate()` function takes a collection (e.g. a tuple) and returns it as an enumerate object

The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.
A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers' task by providing a built-in function enumerate() for this task.
Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object.

Syntax: enumerate(iterable,start)

Converting List into enumerate object:

```
x = ('apple', 'banana',
'cherry')
y = enumerate(x)
print(x)
print(y)
```

Converting enumerate into List:

```
L1 = ['Pepsi', 'Sprite',
'Thumsup']
L2 = enumerate(L1)
print(type(L2))
```

```
# converting to list
print(list(L2))
# changing the default counter
L2 = enumerate(L1, 10)
print(list(L2))
```

Modules:

A group of functions, variables and classes saved to a file, which is nothing but module.

. Every Python file (.py) acts as a module.

Sample.py:

```
x=123
def add(a,b):
    print("sum is:",a+b)
def mul(a,b):
    print("Product is:",a*b)
```

sample module contain one variable and 2 functions.

If we want to use members of module in our program then we should import that module.

import modulename

We can access members by using module name.

modulename.variable

modulename.function()

test.py:

```
import sample
print(sample.x)
sample.add(10,20)
sample.mul(30,20)
```

Note: whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently

Renaming a module at the time of import (module aliasing):

Eg: import durgamath as m

here durgamath is original module name and m is alias name.

We can access members by using alias name m

```
import sample as s
print(s.x)
s.add(10,20)
s.mul(30,20)
```

from ... import:

We can import particular members of module by using from ... import . The main advantage of this is we can access members directly without using module name.

```
from sample import x,add
print(x)
add(10,20)
mul(20,30) # name mul is not defined
```

We can import all members of a module as follows from sample import *

```
from sample import*
print(x)
add(10,20)
mul(20,30)
```

Various possibilties of import:

import modulename
import module1,module2,module3
import module1 as m
import module1 as m1,module2 as m2,module3
from module import member
from module import member1,member2,memebr3
from module import memeber1 as x
from module import *

member aliasing:

from sample import x as y,add as sum

print(y)
sum(10,20)

Once we defined as alias name, we should use alias name only and we should not use original name

```
from sample import x as y, add
as sum
print(x) #name x is not defined
```

The Special variable __name__:

For every Python program , a special variable __name__ will be added internally.

This variable stores information regarding whether the program is executed as an individual program or as a module.

If the program executed as an individual program then the value of this variable is __main__

If the program executed as a module from some other program then the value of this variable is the name of module where it is defined.

Hence by using this __name__ variable we can identify whether the program executed directly or as a module.

Module1.py:

```
def f1():
    if __name__=='__main__':
        print("The Code
Executed as program")
    else:
        print("The Code
Executed as module from some
other program")
    f1()
```

test1.py:

```
import module1
module1.f1()
```

Working with math module:

Python provides inbuilt module math. This module defines several functions which can be used for mathematical operations.

The main important functions are .

1. sqrt(x)
2. ceil(x)
3. floor(x)
4. fabs(x)

5.log(x)
6. sin(x)
7. tan(x) ....

Ex:

```
from math import*
print(sqrt(4))
print(ceil(10.1))
print(floor(10.1))
print(fabs(-10.6))
print(fabs(10.6))
```

Note: We can find help for any module by using help() function

```
import  math
help(math)
```

Working with random module:

This module defines several functions to generate random numbers. We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication.

1. random() function:

This function always generate some float value between 0 and 1 (not inclusive)

```
from random import *
for i in range(10):
    print(random())
```

2. randint() function:

To generate random integer beween two given numbers(inclusive)

EX:

```
from random  import *

for i in range(10):
    print(randint(1,100))
```

3. uniform():

It returns random float values between 2 given numbers(not inclusive

```
from random  import *

for i in range(10):
    print(uniform(1,10))
```

random() ===>in between 0 and 1 (not inclusive)
randint(x,y)   ==>in between x and y (inclusive)

uniform(x,y) ==> in between x and y (not inclusive)

4. randrange([start],stop,[step])

returns a random number from range
start<= x < stop
start argument is optional and default value is 0
step argument is optional and default value is 1

randrange(10)-->generates a number from 0 to 9
randrange(1,11)-->generates a number from 1 to 10
randrange(1,11,2)-->generates a number from 1,3,5,7,9

Ex:

```
from random  import *

for i in range(10):
    print(randrange(10))
```

5. choice() function:

It wont return random number.
It will return a random object from the given list or tuple.

Ex:
```
from random  import *
list=["mohan","rohan","pavan","
bhuvan"]
```

```
for i in range(10):
    print(choice(list))
```

DateTime Module:

Datetime is module to work with dates as date objects.

Imort datetime module and display current date and time

EX:

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

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

EX:

```
import datetime
x=datetime.datetime.now()
print(x.year)
print(x.day)
print(x.month)
```

Creating Date Objects:

The `datetime()` class requires three parameters to create a date: year, month, day.

Ex:

```
import datetime
x=datetime.datetime(2010,3,23)
#x=datetime.datetime(2010,12,32
) day is out of range for
month
print(x)
```

Strrftime() Method:

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

Ex:

```
import datetime
x=datetime.datetime(2018,9,3)
print(x.strftime('%A'))
print(x.strftime('%B'))
print(x.strftime('%a'))
print(x.strftime('%b'))
print(x.strftime('%y'))
print(x.strftime('%Y'))

%a ----weekday,short version
%A-----weekday, full version
%b-----month, short version
%B-----month, full version
%y---- year,short version
%Y-----year. Full version
```

Calendar:

Python has built in function calendar to work with date related tasks.

we import the `calendar` module. The built-in function `month()` inside the module takes in the year and the month and displays the calendar for that month of the year.

```
import  calendar
yy=2016
mm=10
print(calendar.month(yy,mm))
```

Array :

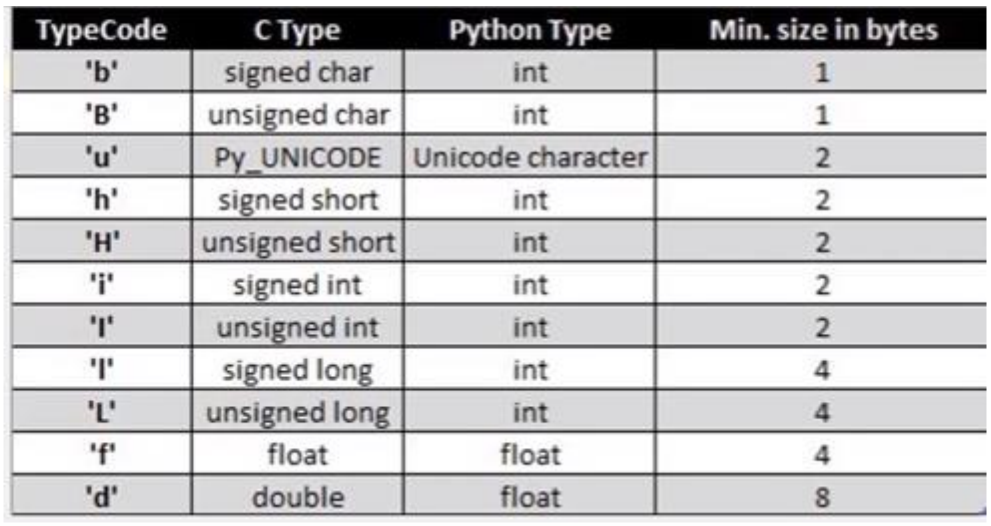
Arrays are fundamental part of most programming languages. It is the collection of elements of a single data type, eg. array of `int`, array of `string`.

However, in Python, there is no native array data structure. So, we use Python lists instead of an array.

Note: If you want to create real arrays in Python, you need to use NumPy's array data structure. For mathematical problems, NumPy Array is more efficient.

Unlike arrays, a single list can store elements of any data type and does everything an array does. We can store an integer, a float and a string inside the same list. So, it is more flexible to work with.

`[10, 20, 30, 40, 50 ]` is an example of what an array would look like in Python, but it is actually a list.



TypeCode	C Type	Python Type	Min. size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'f'	float	float	4
'd'	double	float	8

```
import array
A=array.array('i',[4,3,2,5])
print(A)
print(type(A))
```

```
import array as arr
A=arr.array('i',[4,3,2,5])
print(A)


from array import *
A=array('i',[4,3,2,5])
print(A)

from array import *
A=array('i',[4,3,2,5])
print(A)
print(type(A))
print(A.typecode)
print(A.buffer_info())
print(A.remove(A[2]))
print(A)
print(A.append(33))
print(A)
print(A.reverse())
print(A)
for i in range(4):
    print(A[i])
for i in range(len(A)):
    print(A[i])

#copying Array
from array import *
A=array('i',[2,3,4,5])
print(A)
B=array(A.typecode,(i*i for i
in A))
print(B)


#character array
from array import *
A=array('u',['m','o','h','a','n
'])
print(A)
```

# Accepting the values from user and print Array:

```
from array import *
arr=array('i',[])
n=int(input("Enter Length of
Array"))
for i in range(n):
    x=int(input("Enter the next
value:"))
    arr.append(x)
print(arr)
s=int(input("Enter the value
for search:"))
#searchinng element in Array
print(arr.index(s))
```

What is NumPy?

NumPy is a general-purpose array-processing package.

It provides a high-performance multidimensional array object, and tools for working with these arrays.

It is the fundamental package for scientific computing with Python.

- In NumPy dimensions are called *axes*. The number of axes is *rank*.

To create multi dimension Array in python we use Numpy.

To Install Numpy in command prompt:

Goto command prompt and type : pip3 install numpy

To Install Numpy in Pycharm:

Click on File

Click on settings

Expand Project

Click on Project Interpreter

Click on + Symbol from RHS(Right Hand Side)

Type numpy in search

Choose numpy

Then Click on install numpy package.

Creating Multi dimension Array In different ways like:

Array()

Linespace()

Logspace()

Arange()

Zeros()

Ones()

EX:

Array():

```
from numpy import *
arr=array([1,2,3,4,5])
print(arr)
print(arr.dtype)

from numpy import *
arr=array([1,2,3,4.7,5])
print(arr)
print(arr.dtype)

from numpy import *
arr=array([1,2,3,4.7,5],int)
print(arr)
print(arr.dtype)
```

Linespace()

```
from numpy import *
arr=linspace(0,10,11)#
start,stop,no of parts
print(arr)
```

Logspace()

```
from numpy import *
arr=logspace(1,20,5)#
start,stop,no of parts
print(arr)
```

Arange()

```
from numpy import *
arr=arange(1,20,2)#
start,stop,step
print(arr)
```

Zeros()

```
from numpy import *
arr=zeros(6,int)
print(arr)
```

Ones()

```
from numpy import *
arr=ones(6,int)
print(arr)
```

```
import numpy as np
#creating array
arr=np.array([[1,2,3],
              [4,5,6]])
#printing type of array
print("Array type
is:",type(arr))
# Printing array dimensions
(axes)
print("No. of dimensions: ",
arr.ndim)
# Printing shape of array
print("Shape of array: ",
arr.shape)
# Printing size (total number
of elements) of array
print("Size of array: ",
arr.size)
# Printing type of elements in
array
print("Array stores elements of
type: ", arr.dtype)
```

We can use **flatten** method to get a copy of array collapsed into **one dimension**. It accepts *order* argument.

```
import numpy as np
#creating array
arr=np.array([[1,2,3],
              [4,5,6]])
flat = arr.flatten()
print(flat)
```

Basic Operations in Multi dimension Array:

```
# unary operators in numpy
import numpy as np
arr = np.array([[1, 5, 6],
                [4, 7, 2],
                [3, 1, 9]])
# maximum element of array
print("Largest element is:",
arr.max())
print("Row-wise maximum
elements:",
      arr.max(axis=1))
# minimum element of array
```

```
print("Column-wise minimum
elements:",
      arr.min(axis=0))
# sum of array elements
print("Sum of all array
elements:",
      arr.sum())
```

Object Oriented Programming:

What is Class:

⚽ **In Python every thing is an object.**

To create objects we required some Model or Plan or Blue print, which is nothing but class.

⚽ **We can write a class to represent properties (attributes) and actions (behaviour) of object.**

⚽ Properties can be represented by variables

⚽ Actions can be represented by Methods.

⚽ Hence class contains both variables and methods.

How to Define a class?

We can define a class by using class keyword.

Syntax: class className:

''' documenttation string '''

variables:instance variables,static and local variables

methods: instance methods,static methods,class methods

Documentation string represents description of the class.

Within the class doc string is always optional.

We can get doc string by using the following 2 ways.

1. print(classname.__doc__)

2. help(classname)

```
class student:
    """this is student calss
    with required data"""
print(student.__doc__)
help(student)
```

Within the Python class we can represent data by using variables.

There are 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)

2. Static Variables (Class Level Variables)

3. Local variables (Method Level Variables)

Within the Python class, we can represent operations by using methods.

The following are various types of allowed methods

1. Instance Methods

2. Class Methods

3. Static Methods

Example for class:

```
class student:
    def __int__(self):
        self.id=123
        self.name='mohan'
        self.age=34

    def details(self):
        print("My EmpId:",self.id)
        print("My Name:",self.name)
        print("My Age:",self.age)
```

What is Object:

Pysical existence of a class is nothing but object. We can create any number of objects for a class.

Syntax to create object:

referencevariable = classname()

Example: s = Student()

What is Reference Variable:

The variable which can be used to refer object is called reference variable. By using reference variable, we can access properties and methods of object.

Program: Write a Python program to create a Student class and Creates an object to it. Call the method , to display student details

EX:

```
class Student:

      def __init__(self, name, rollno, marks):

        self.name = name
        self.rollno = rollno
        self.marks = marks


      def talk(self):

        print("Hello My Name is:", self.name)
        print("My Rollno is:", self.rollno)
        print("My Marks are:", self.marks)

s1 = Student("Durga", 101, 80)
s1.talk()
```

Self variable:

self is the default variable which is always pointing to current object (like this keyword in Java..Net)

By using self we can access instance variables and instance methods of object.

Note:
1. self should be first parameter inside constructor   def __init__(self):

2. self should be first parameter inside instance methods   def talk(self):

Constructor Concept:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once.
- Constructor can take at least one argument(at least self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Ex:
```
def __init__(self,name,rollno,marks):
    self.name=name
    self.rollno=rollno
    self.marks=marks
```

Program to demonistrate constructor will execute only once per object:

```
class test:

    def __init__(self):
        print("Constructor Execution")
    def m1(self):
        print("Method Execution")

t1=test()
t2=test()
t3=test()
t1.m1()
```

```
EX:
class Student:
      def __init__(self,x,y,z):
          self.name=x
          self.rollno=y
          self.marks=z

      def display(self):
          print("Student Name:{}\nRollno:{}
\nMarks:{}".format(self.name,self.rollno,self.marks
) )

s1=Student("Durga",101,80)
s1.display()
s2=Student("Sunny",102,100)
s2.display()
```

Differences between Methods and Constructors:

Method	Constructor
1. Name of method can be any name	1. Constructor name should be always __init__
2. Method will be executed if we call that method	2. Constructor will be executed automatically at the time of object creation.
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once
4. Inside method we can write business logic	4. Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)

2. Static Variables (Class Level Variables)

3. Local variables (Method Level Variables)

1. Instance Variables:

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

For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:

1. Inside Constructor by using self variable

2. Inside Instance Method by using self variable

3. Outside of the class by using object reference variable

1. Inside Constructor by using self variable:

We can declare instance variables inside a constructor by using self keyword. Once we creates object, automatically these variables will be added to the object.

EX:

```
#inside constructor by using self variable
class Employee:

    def __init__(self):
        self.eno=101
        self.ename='durga'
        self.esal=23000

e=Employee()

print(e.__dict__)
```

2. Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable.
If any instance variable declared inside instance method, that instance variable will be added once we call that method.

Ex:

```
#inside instance method by using self variable

class test:

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

    def m1(self):
        self.c=30

t=test()
t.m1()
print(t.__dict__)
```

3. Outside of the class by using object reference variable:

We can also add instance variables outside of a class to a particular object.

Ex:

```
#outside of the class by using object reference
variable

class test:

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

    def m1(self):
        self.c=30

t=test()
t.m1()
t.d=40
print(t.__dict__)
```

How to access Instance variables:

We can access instance variables with in the class by using self variable and outside of the class by using object reference.

Ex:

```
#how to access instance variables

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

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

t=test()
t.display()
print(t.a,t.b)
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

del self.variableName

2. From outside of class we can delete instance variables as follows

del objectreference.variableName

EX:

```
#how to delete instance variable from object

class test:

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

    def m1(self):
        del self.d

t=test()
```

```
print(t.__dict__)
t.m1()
print(t.__dict__)
del t.c
print(t.__dict__)
```

Note: The instance variables which are deleted from one object, will not be deleted from other objects

EX:

```
class test:

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

t1=test()
t2=test()
del t1.a
print(t1.__dict__)
print(t2.__dict__)
```

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

Ex:

```
class test:

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

t1=test()
t1.a=123
t1.b=345
t2=test()
print("t1:",t1.a,t1.b)
print("t2:",t2.a,t2.b)
```

1. Static variables:

If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.

For total class only one copy of static variable will be created and shared by all objects of that class.

We can access static variables either by class name or by object reference. But recommended to use class name.

Instance Variable vs Static Variable:

Note: In the case of instance variables for every object a seperate copy will be created,but in the case of static variables for total class only one copy will be created and shared by every object of that class

EX:

```
#static variables

class test:
    x=10
    def __init__(self):
        self.y=20

t1=test()
t2=test()

print("t1:",t1.x,t1.y)
print("t2:",t2.x,t2.y)

test.x=777
t1.y=999

print("t1:",t1.x,t1.y)
print("t2:",t2.x,t2.y)
```

Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside class method by using either class name or class variable
5. Inside static method by using class name

```
EX:  class  test:
    a=10
    def __init__(self):
        test.b=20
    def m1(self):
        test.c=30
    @classmethod
    def m2(cls):
        cls.d1=40
        test.d2=400
    @staticmethod
    def m3():
        test.e=50

print(test.__dict__)
t=test()
print(test.__dict__)
t.m1()
print(test.__dict__)
test.m2()
print(test.__dict__)
test.m3()
print(test.__dict__)
test.f=60
print(test.__dict__)
```

Local variables:

Sometimes to meet temporary requirements of programmer,we can declare variables inside a method directly,such type of variables are called local variable or temporary variables.

Local variables will be created at the time of method execution and destroyed once method completes.

Local variables of a method cannot be accessed from outside of method.

EX:

```
class test:
    def m1(self):
        a=1000
        print(a)
    def m2(self):
        b=2000
       print(a)  #a is not defined
       print(b)
t=test()
t.m1()
t.m2()
```

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods

2. Class Methods

3. Static Methods

1. Instance Methods:

Inside method implementation if we are using instance variables then such type of methods are called instance methods.

Inside instance method declaration, we have to pass self variable.

def m1(self):

By using self variable inside method we can able to access instance variables.

Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

Ex:

```
class student:

    def __init__(self,name,marks):
        self.name=name
        self.marks=marks

    def display(self):
        print("Hi",self.name)
        print("Your marks are:",self.marks)

    def grade(self):
        if self.marks>60:
            print("You Got First Grade")
        elif self.marks>50:
            print("You got Second Grade")
        elif self.marks>=35:
            print("You got third Grade")
        else:
            print("You Failed")

for i in range(1):
    name=input("Enter Name:")
    marks=int(input("Enter Marks"))

s=student(name,marks)
s.display()
s.grade()
print()
```

Setter and Getter Methods:

We can set and get the values of instance variables by using getter and setter methods.

Setter Method:

setter methods can be used to set values to the instance variables.

setter methods also known as mutator methods.

syntax:

```
def  setVariable(self,variable):
  self.variable=variable
```

Example:

```
def  setName(self,name):
  self.name=name
```

Getter Method:

Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

```
def  getVariable(self):
   return self.variable
```

Example:

```
def  getName(self):
```

return self.name

Ex:

```
class Student:
    def setName(self,name):
        self.name=name
    def getName(self):
       return self.name
    def setMarks(self,marks):
        self.marks=marks
    def getMarks(self):
        return self.marks
for i in range(1):
  s=Student()
name=input('Enter Name:')
s.setName(name)
marks=int(input('Enter Marks:'))
s.setMarks(marks)
print('Hi',s.getName())
print('Your Marks are:',s.getMarks())
print()
```

2. Class Methods:

Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

We can declare class method explicitly by using @classmethod decorator.

For class method we should provide class variable at the time of declaration.

We can call classmethod by using classname or object reference variable.

Ex:

```
class Animal:
    legs=4
    @classmethod
    def walk(cls,name):
      print('{} walks with {}
legs...'.format(name,cls.legs))

Animal.walk('Dog')
Animal.walk('Cat')
```

3. Static Methods:

In general these methods are general utility methods. Inside these methods we won't use any instance or class variables.

Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator

We can access static methods by using classname or object reference

EX:

```
class DurgaMath:
    @staticmethod
    def add(x, y):
        print('The Sum:', x + y)
    @staticmethod
    def product(x, y):
        print('The Product:', x * y)
    @staticmethod
    def average(x, y):
        print('The average:', (x + y) / 2)
DurgaMath.add(10, 20)
DurgaMath.product(10, 20)
DurgaMath.average(10, 20)
```

Note: In general we can use only instance and static methods.Inside static method we can access class level variables by using class name.

class methods are most rarely used methods in python.

Accessing one class members into other class.

Ex:

```
class Employee:
    def __init__(self,eno,ename,esal):
        self.eno=eno
        self.ename=ename
        self.esal=esal

    def display(self):
        print('Employee Number:',self.eno)
        print('Employee Name:',self.ename)
        print('Employee Salary:',self.esal)

class Test:
    def modify(emp):
        emp.esal=emp.esal+10000
        emp.display()
```

```
e=Employee(100,'Durga',10000)
Test.modify(e)
```

In the above application, Employee class members are available to Test class

Inner classes:

Sometimes we can declare a class inside another class, such type of classes are called inner classes.

Without existing one type of object if there is no chance of existing another type of object, then we should go for inner classes.

Example: Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

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

Example: Without existing university object there is no chance of existing Department object

```
 class University:
 .....
class Department:
  ......
```

eg3: Without existing Human there is no chance of existing Head. Hence Head should be part of Human.

```
class Human:
   class Head:
```

Note: Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

EX:

```
class Outer:
   def __init__(self):
      print("outer class object creation")

   class Inner:
    def __init__(self):
      print("inner class object creation")

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

o=Outer()
i=o.Inner()
i.m1()
```

Note: The following are various possible syntaxes for calling inner class method

1. o=Outer()
i=o.Inner()
i.m1()

2. i=Outer().Inner()
i.m1()

3. Outer().Inner().m1()

Inside a class we can declare any number of inner classes

Ex:

```
class Human:
    def __init__(self):
        self.name = 'Sunny'
        self.head = self.Head()
        self.brain = self.Brain()
    def display(self):
      print("Hello..", self.name)

    class Head:
         def talk(self):
             print('Talking...')

    class Brain:
        def think(self):
            print('Thinking...')

h = Human()
h.display()
h.head.talk()
h.brain.think()
```

Garbage Collection:

In old languages like C++, programmer is responsible for both creation and destruction of objects.Usually programmer taking very much care while creating object, but neglecting destruction of useless objects. Because of his neglectance, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.

But in Python, We have some assistant which is always running in the background to destroy useless objects.Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.

Hence the main objective of Garbage Collector is to destroy useless objects.

If an object does not have any reference variable then that object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our program:

By default Gargbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.

1. gc.isenabled()

   Returns True if GC enabled

2. **gc.disable()**

   To disable GC explicitly

3. **gc.enable()**

   To enable GC explicitly

EX:

```
import gc

print(gc.isenabled())
gc.disable()
print(gc.isenabled())
gc.enable()
print(gc.isenabled())
```

Destructors:

Destructor is a special method and the name should be __del__ Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc). Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities.

How to find the number of references of an object:

sys module contains getrefcount() function for this purpose.

sys.getrefcount(objectreference)

Example:

Note: For every object, Python internally maintains one default reference variable self.

```
import sys

class Test:
    pass

t1=Test()
t2=t1
t3=t1
t4=t1
print(sys.getrefcount(t1))
```

Encapsulation:

In object oriented programming we can restrict the access to the variables and methods , this process is called Encapsulation.

Why we are using Encapsulation means to prevent the data from modify by accidentally.

There are Two Types of access specifiers

1.private

2.public

Till now we use pubic variables and public methods, means we can access them outside the class also.

But private methods can not access outside the class.

Ex:

```
class car:

    def __init__(self):
        self.__updatesoftware()

    def drive(self):
        print("driving")

    def __updatesoftware(self):
        print("updating software")

c=car()
c.drive()
```

Ex:

```
class car:

    #def __init__(self):
      #  self.__updatesoftware()
    def drive(self):
        print("driving")

    def __updatesoftware(self):
        print("updating software")

c=car()
c.drive()
c.__updatesoftware()
```

in the above example when we call private method outside the class ,it is not possible.

Private Variables:

These variables are access inside the class methods.

We can not modifiy private variables outside the class.

EX:

```
class car:
   __maxspeed=0
   __name=""

   def __init__(self):
        self.__maxspeed=200
        self.__name="FordFigo"

   def drive(self):
        print("driving")
        print(self.__maxspeed)

    def setspeed(self,speed):
        self.__maxspeed=speed
        print(self.__maxspeed)

c=car()
c.drive()
c.setspeed(100)
```

Ex2:

```
class car:
    __maxspeed=0
    __name=""

    def __init__(self):
        self.__maxspeed=200
        self.__name="FordFigo"

    def drive(self):
        print("driving")
        print(self.__maxspeed)


c=car()
c.drive()
c.__maxspeed=100
c.drive()
```

Polymorphism:

Poly means many. Morphs means forms.

Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes.(overriding)

Related to polymorphism the following topics are important

1. Duck Typing Philosophy of Python

2. Overloading

1. Operator Overloading

2. Method Overloading

3. Constructor Overloading

3. Overriding

1. Method overriding

2. constructor overriding

1. Duck Typing Philosophy of Python:

In Python we cannot specify the type explicitly. Based on provided value at runtime the type will be considered automatically. Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):
  obj.talk()
```

What is the type of obj? We cannot decide at the beginning. At runtime we can pass any type.Then how we can decide the type? At runtime if 'it walks like a duck and talks like a duck,it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

Overloading:

We can use same operator or methods for different purposes.

Eg1:

+ operator can be used for Arithmetic addition and String concatenation

```
print(10+20)#30
 print('durga'+'soft')#durgasoft
```

Eg2: * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200
print('durga'*3)#durgadurgadurga
```

Eg3: We can use deposit() method to deposit cash or cheque or dd

```
deposit(cash)
deposit(cheque)
deposit(dd)
```

There are 3 types of overloading

1. Operator Overloading

2. Method Overloading

3. Constructor Overloading

1. Operator Overloading:

We can use the same operator for multiple purposes, which is nothing but operator overloading.

Python supports operator overloading.

Eg1: + operator can be used for Arithmetic addition and String concatenation

print(10+20)#30

print('durga'+'soft')#durgasoft

Eg2: * operator can be used for multiplication and string repetition purposes.

print(10*20)#200

print('durga'*3)#durgadurgadurga

Demo program to use + operator for our class objects:

Ex:

```
class Book:

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

```
b1 = Book(100)
b2 = Book(200)
print(b1 + b2)
```

We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.

For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.

Internally + operator is implemented by using __add__() method. This method is called magic method for + operator.

We have to override this method in our class.

Demo program to overload + operator for our Book class objects:

EX:

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

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

b1=Book(100)
b2=Book(200)
print('The Total Number of Pages:',b1+b2)
```

Program to overload multiplication operator to work on Employee objects:

EX:

```
class Employee:
    def __init__(self,name,salary):
        self.name=name
        self.salary=salary

    def __mul__(self,other):
         return self.salary*other.days

class TimeSheet:
    def __init__(self,name,days):
        self.name=name
        self.days=days

e=Employee('Durga',500)
t=TimeSheet('Durga',25)
print('This Month Salary:',e*t)
```

2. Method Overloading:

If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.

Eg:
m1(int a)
m1(double d)

But in Python Method overloading is not possible.

If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

```
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()
#  #t.m1(10)
# t.m1(10,20)
t.m1(10,20)
```

In the above program python will consider only last method.

3. Constructor Overloading:

Constructor overloading is not possible in Python.
If we define multiple constructors then the last constructor will be considered.

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')

#t1=Test()
#  #t1=Test(10)
t1=Test(10,20)
```

Method overriding:

What ever members available in the parent class are by default available to the child class through inheritance.

If the child class not satisfied with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.
Overriding concept applicable for both methods and constructors.

EX:
```
class P:
    def property(self):
        print('Gold+Land+Cash+Power')

    def marry(self):
        print('Appalamma')

class C(P):
    def marry(self):
        print('Katrina Kaif')

c=C()
c.property()
c.marry()
```

From Overriding method of child class,we can call parent class method also by using super() method.

EX:

```
class P:
    def property(self):
        print('Gold+Land+Cash+Power')

    def marry(self):
        print('Appalamma')
```

```
class C(P):
    def marry(self):
        super().marry()
        print('Katrina Kaif')

c=C()
c.property()
c.marry()
```

Inheritance :

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new class with little or no modification to an existing class.

The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Syntax:

Class Baseclass:

Body of baseclass

Class Derivedclass(Baseclass):

Body of Derivedclass

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Ex:

```
class Vehicle:
    def general_usage(self):
        print("general use: Transportation")

class car(Vehicle):
    def __init__(self):
        print("I am a car")
        self.wheels=4
        self.hasroof=True

    def specific_usage(self):
        print("Specific use:vacation with family")

class Motorcycle(Vehicle):
    def __init__(self):
        print("I am a Motor Cycle")
        self.wheels = 2
        self.hasroof = False

    def specific_usage(self):
        print("Specific use:road trip,racing")


c=car()
c.general_usage()
c.specific_usage()

m=Motorcycle()
m.general_usage()
m.specific_usage()
```

Benefits of Inheritance:

1.Code Reuse

2.Extensibility

3.Readability

Isinstance and issubclass Methods:

Isinsatnce is a builtin function, it will check an object is instance of specific class or not.

Issubclass is builtin function, it will check a class is subclass of other class or not.

```
c=car()
m=Motorcycle()


print(isinstance(c,car))
print(isinstance(c,Motorcycle))

print(issubclass(car,Vehicle))
print(issubclass(car,Motorcycle))
```

Exception Handling:

In any programming language there are 2 types of errors are possible.

1. Syntax Errors   2. Runtime Errors

1. Syntax Errors:

The errors which occurs because of invalid syntax are called syntax errors.

EX:

```
x=10
if x==10
   print("hello")
```

syntax Error: invalid syntax

Ex:

```
print "Hello"
```

SyntaxError: Missing parentheses in call to 'print'

Note: Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2. Runtime Errors:

Also known as exceptions. While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.

Eg: print(10/0) ==>ZeroDivisionError: division by zero

print(10/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

```
x=int(input("Enter Number:"))
print(x)
```

Enter Number :ten   ValueError: invalid literal for int() with base 10: 'ten'

Note: Exception Handling concept applicable for Runtime Errors but not for syntax errors

What is Exception:

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

ZeroDivisionError

TypeError

ValueError

FileNotFoundError

EOFError

SleepingError

TyrePuncturedError

It is highly recommended to handle exceptions.

The main objective of exception handling is Graceful Termination of the program(i.e we should not block our resources and we should not miss anything)

Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Eg:

For example our programming requirement is reading data from remote file locating at Hyd. At runtime if Hyd file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Default Exception Handing in Python:

Every exception in Python is an object. For every exception type the corresponding classes are available.

Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console. The rest of the program won't be executed.

EX:

```
print("Hello")
print(10/0)
print("hai")
```

```
ZeroDivisionError: division by zero
```

Every Exception in Python is a class. All exception classes are child classes of BaseException.i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy.

Most of the times being a programmer we have to concentrate Exception and its child classes.

Customized Exception Handling by using try-except:

It is highly recommended to handle exceptions. The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.

```
try:

  Risky Code

except XXXXX:
```

Handling code/Alternative Code

EX:

without try-except

```
print("Hello")
print(10/0)
print("hai")
```

OUTPUT:

Hello

Division by zero

Abnormal termination/Non-Graceful Termination

with try-except:

Ex:

```
print("Hello")
try:
    print(10/0)
except ZeroDivisionError:
    print(10/2)
print("hai")
```

Normal termination/Graceful Termination

Control Flow in try-except:

```
try:
   stmt-1
   stmt-2
   stmt-3
except XXX:
    stmt-4
stmt-5
```

case-1: If there is no exception

1,2,3,5 and Normal Termination

case-2: If an exception raised at stmt-2 and corresponding except block matched

1,4,5 Normal Termination

case-3: If an exception raised at stmt-2 and corresponding except block not matched

1, Abnormal Termination

case-4: If an exception raised at stmt-4 or at stmt-5 then it is always abnormal termination.

Conclusions:

1. **within the try block if anywhere exception raised then rest of the try block wont be executed even though we handled that exception.**
   Hence we have to take only risky code inside try block and length of the try block should be as less as possible.

2. In addition to try block,there may be a chance of raising exceptions inside except and finally blocks also.

3. If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to print exception information:

EX:

```
try:
    print(10/0)
except ZeroDivisionError as msg:
    print("exception raised and its description is:", msg)
```

try with multiple except blocks:

The way of handling exception is varied from exception to exception.

Hence for every exception type a seperate except block we have to provide.

i.e try with multiple except blocks is possible and recommended to use.

Eg:

```
try:
    -------
   -------
    -------
```

except ZeroDivisionError:

perform alternative

arithmetic operations

except FileNotFoundError:

use local file instead of remote file

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

EX:

```
try:
    x=int(input("Enter First Number: 
"))
    y=int(input("Enter Second Number: 
"))
    print(x/y)
except ZeroDivisionError :
    print("Can't Divide with Zero")
except ValueError:
    print("please provide int value 
only")
```

If try with multiple except blocks available then the order of these except blocks is important .

Python interpreter will always consider from top to bottom until matched except block identified.

EX:

```
try:
    x=int(input("Enter First Number: 
"))
    y=int(input("Enter Second Number: 
"))
    print(x/y)
except ArithmeticError :
    print("ArithmeticError")
except ZeroDivisionError:
    print("ZeroDivisionError")
```

Single except block that can handle multiple exceptions:

We can write a single except block that can handle multiple different types of exceptions.

except (Exception1,Exception2,exception3,..):

or

except (Exception1,Exception2,exception3,..) as msg :

Parenthesis are mandatory and this group of exceptions internally considered as tuple.

EX:

```
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
```

```
except (ZeroDivisionError,ValueError) as msg:
   print("Plz Provide valid numbers only and
problem is: ",msg)
```

Default except block:

We can use default except block to handle any type of exceptions. In default except block generally we can print normal error messages.

Syntax:

except:

statements

EX:

```
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
    print("ZeroDivisionError:Can't divide with
zero")
except:
    print("Default Except:Plz provide valid input
only")
```

*****Note: If try with multiple except blocks available then default except block should be last,otherwise we will get SyntaxError.**

EX:

```
try:
    print(10/0)
except:
    print("Default Except")
except ZeroDivisionError:
    print("ZeroDivisionError")

#SyntaxError: default 'except:' must be last
```

Note:

The following are various possible combinations of except blocks

1. **except ZeroDivisionError:**
2. **except ZeroDivisionError as msg:**
3. **except (ZeroDivisionError,ValueError) :**
4. **except (ZeroDivisionError,ValueError) as msg:**
5. **except :**

finally block:

1. It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.

2. **It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.**

Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.

Hence the main purpose of finally block is to maintain clean up code.

Syntax:

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup code
```

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

Ex:

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

Case-2: If there is an exception raised but handled

EX:

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

Case-3: If there is an exception raised but not handled:

EX:

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

***** Note: There is only one situation where finally block won't be executed i.e whenever we are using os._exit(0) function.**

Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown.In this particular case finally won't be executed.

EX:

```
import os
try:
    print("try")
    os._exit(0)
except NameError:
    print("Except")
finally:
    print("Finally")
```

Note: os._exit(0)   where 0 represents status code and it indicates normal termination   There are multiple status codes are possible.

Control flow in try-except-finally:

try:

  stmt-1

  stmt-2

   stmt-3

except:

   stmt-4

finally:

     stmt-5

stmt6

Case-1: If there is no exception 1,2,3,5,6 Normal Termination

Case-2: If an exception raised at stmt2 and the corresponding except block matched 1,4,5,6 Normal Termination

Case-3: If an exception raised at stmt2 but the corresponding except block not matched 1,5 Abnormal Termination

Case-4:If an exception raised at stmt4 then it is always abnormal termination but before that finally block will be executed.

Case-5: If an exception raised at stmt-5 or at stmt-6 then it is always abnormal termination

Nested try-except-finally blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of tryexcept-finally is possible.

```
try:
 ----------
 ----------
 ----------
     try:
          -------------
           --------------
            --------------
    except:
         --------------
          --------------
         --------------
```

----

except:

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

EX:

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

Control flow in nested try-except-finally:

try:

stmt-1

```
    stmt-2
    stmt-3
    try:
        stmt-4
        stmt-5
       stmt-6
   except X:
       stmt-7
   finally:
        stmt-8
  stmt-9
except Y:
  stmt-10
finally:
    stmt-11
stmt-12
```

case-1: If there is no exception   1,2,3,4,5,6,8,9,11,12 Normal Termination

case-2: If an exception raised at stmt-2 and the corresponding except block matched   1,10,11,12 Normal Termination

case-3: If an exceptiion raised at stmt-2 and the corresponding except block not matched   1,11,Abnormal Termination

case-4: If an exception raised at stmt-5 and inner except block matched 1,2,3,4,7,8,9,11,12 Normal Termination

case-5: If an exception raised at stmt-5 and inner except block not matched but outer except block matched

1,2,3,4,8,10,11,12,Normal Termination

case-6:If an exception raised at stmt-5 and both inner and outer except blocks are not matched

1,2,3,4,8,11,Abnormal Termination

case-7: If an exception raised at stmt-7 and corresponding except block matched 1,2,3,.,.,.,8,10,11,12,Normal Termination

case-8: If an exception raised at stmt-7 and corresponding except block not matched 1,2,3,.,.,.,8,11,Abnormal Termination

case-9: If an exception raised at stmt-8 and corresponding except block matched 1,2,3,.,.,.,.,10,11,12 Normal Termination

case-10: If an exception raised at stmt-8 and corresponding except block not matched 1,2,3,.,.,.,.,11,Abnormal Termination

case-11: If an exception raised at stmt-9 and corresponding except block matched 1,2,3,.,.,.,.,8,10,11,12,Normal Termination

case-12: If an exception raised at stmt-9 and corresponding except block not matched 1,2,3,.,.,.,.,8,11,Abnormal Termination

case-13: If an exception raised at stmt-10 then it is always abnormal termination but before abnormal termination finally block(stmt-11) will be executed.

case-14: If an exception raised at stmt-11 or stmt-12 then it is always abnormal termination.

Note: If the control entered into try block then compulsory finally block will be executed. If the control not entered into try block then finally block won't be executed.

else block with try-except-finally:

We can use else block with try-except-finally blocks.

else block will be executed if and only if there are no exceptions inside try block.

```
try:
        Risky Code
except:
        will be executed if exception inside try
else:
        will be executed if there is no exception inside try
finally:
     will be executed whether exception raised or not raised and handled
or   not handled
```

Eg:

```
try:
  print("try")
  print(10/0)--->1
 except:
 print("except")
else:
print("else")
finally:
 print("finally")
```

If we comment line-1 then else block will be executed b'z there is no exception inside try. In this case the output is:

```
try
else
 finally
```

If we are not commenting line-1 then else block won't be executed b'z there is exception inside try block. In this case output is:

```
try
```

except

finally

Various possible combinations of try-except-else-finally:

1. Whenever we are writing try block, compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.

2. Wheneever we are writing except block, compulsory we should write try block. i.e except without try is always invalid.

3. Whenever we are writing finally block, compulsory we should write try block. i.e finally without try is always invalid.

4. We can write multiple except blocks for the same try,but we cannot write multiple finally blocks for the same try

5. Whenever we are writing else block compulsory except block should be there. i.e without except we cannot write else block.

6. In try-except-else-finally order is important.

7. We can define try-except-else-finally inside try,except,else and finally blocks. i.e nesting of try-except-else-finally is always possible.

Types of Exceptions:
In Python there are 2 types of exceptions are possible.
1. Predefined Exceptions
2. User Definded Exceptions

1. Predefined Exceptions:

Also known as in-built exceptions

The exceptions which are raised automatically by Python virtual machine whenver a particular event occurs, are called pre defined exceptions.

Eg 1: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.
print(10/0)

Eg 2: Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

x=int("ten")===>ValueError

2. User Defined Exceptions:

Also known as Customized Exceptions or Programatic Exceptions

Some time we have to define and raise exceptions explicitly to indicate that something goes wrong ,such type of exceptions are called User Defined Exceptions or Customized Exceptions

Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Eg:
InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

How to Define and Raise Customized Exceptions:

Every exception in Python is a class that extends Exception class either directly or indirectly.

Syntax:

class classname(predefined exception class name):

```
    def __init__(self,arg):

     self.msg=arg
```

EX:

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

TooYoungException is our class name which is the child class of Exception

We can raise exception by using raise keyword as follows

raise TooYoungException("message")

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 TooYoungException("Plz wait
some more time you will get best
match soon!!!")
elif age < 18:
    raise TooOldException(
    "Your age already crossed
marriage age...no chance of getting
ma rriage")
else:
    print("You will get match details
soon by email!!!")
```

Note: raise keyword is best suitable for customized exceptions but not for pre defined exceptions

Regular Expressions:

If we want to represent a group of Strings according to a particular format/pattern then we should go for Regular Expressions.

i.e Regualr Expressions is a declarative mechanism to represent a group of Strings accroding to particular format/pattern.

Eg 1: We can write a regular expression to represent all mobile numbers

Eg 2: We can write a regular expression to represent all mail ids.

The main important application areas of Regular Expressions are

1. To develop validation frameworks/validation logic

2. To develop Pattern matching applications (ctrl-f in windows, grep in UNIX etc)

3. To develop Translators like compilers, interpreters etc

4. To develop digital circuits

5. To develop communication protocols like TCP/IP, UDP etc.

We can develop Regular Expression Based applications by using python module: re

This module contains several inbuilt functions to use Regular Expressions very easily in our applications.

1. compile() re module contains compile() function to compile a pattern into Regex Object.

pattern = re.compile("ab")

2. finditer(): Returns an Iterator object which yields Match object for every Match

matcher = pattern.finditer("abaababa")

On Match object we can call the following methods.

1. start() : Returns start index of the match

2. end() : Returns end+1 index of the match

3. group() : Returns the matched string

EX:

```
import re
count=0
pattern=re.compile("ab")
matcher=pattern.finditer("abaababa")
for match in matcher:
    count+=1

print(match.start(),"...",match.end(),"...",match.g
roup())
print("The number of occurrences: ",count)
```

Character classes:

We can use character classes to search a group of characters

1. [abc]===>Either a or b or c

2. [^abc] ===>Except a and b and c

3. [a-z]==>Any Lower case alphabet symbol

4. [A-Z]===>Any upper case alphabet symbol

5. [a-zA-Z]==>Any alphabet symbol

6. [0-9]==> Any digit from 0 to 9

7. [a-zA-Z0-9]==>Any alphanumeric character

8. [^a-zA-Z0-9]==>Except alphanumeric characters(Special Characters)

EX:

```
import re
matcher=re.finditer("x","a7b@k9z")
for match in matcher:
   print(match.start(),"......",match.group())
```

```
x = [abc]
0 ...... a
 2 ...... b

x = [^abc]
 1 ...... 7
3 ...... @
4 ...... k
5 ...... 9
6 ...... z

x = [a-z]
 0 ...... a
 2 ...... b
 4 ...... k
 6 ...... z
```

x = [0-9]
1 7
5 9

x = [a-zA-Z0-9]
0 ...... a
1 7
2 ...... b
4 ...... k
5 9
6 ...... z

x = [^a-zA-Z0-9]
3 ...... @

Pre defined Character classes:

\s Space character

\S Any character except space character

\d Any digit from 0 to 9

\D Any character except digit

\w Any word character [a-zA-Z0-9]

\W Any character except word character (Special Characters)

. Any character including special characters

EX:

```
import re
matcher=re.finditer("x","a7b@k9z")
for match in matcher:
    print(match.start(),"......",match.group())
```

x = \s

3 ......

x = \S

0 ...... a

1 7

2 ...... b

4 ...... k

5 ...... @

6 9

7 ...... z

x = \d

1 7

6 9

x = \D

0 ....a

2 ...... b

3 ......

4 ...... k

5 ...... @

7 ...... z

x = \w

0 ...... a

1 7

2 ...... b

4 ...... k

6 9

7 ...... z

x = \W

3 ......

5 ...... @

x = .

0 ...... a

1 7

2 ...... b

3 ......

4 ...... k

5 ...... @

6 9

7 ...... z

Qunatifiers:

We can use quantifiers to specify the number of occurrences to match.

a -- Exactly one 'a'

a+ ---Atleast one 'a'

a* --- Any number of a's including zero number

a? --- Atmost one 'a' ie either zero number or one number

a{m}--- Exactly m number of a's

a{m,n} ---Minimum m number of a's and Maximum n number of a's

Ex:

```
import re
matcher=re.finditer("x","ababaaab")
for match in matcher:
   print(match.start(),"......",match.group())
```

x = a

0 ...... a

2 ...... a

3 ...... a

5 ...... a

6 ...... a

7 ...... a

x = a+

0 ...... a

2 ...... aa

5 ...... aaa

x = a*

0 ...... a

1 ......

2 ...... aa

4 ......

5 ...... aaa

8 ......

9 ......

x = a?

0 ...... a

1 ......

2 ...... a

3 ...... a

4 ......

5 ...... a

6 ...... a

7 ...... a

8 ......

9 ......

x = a{3}

5 ...... aaa

x = a{2,4}

2 ...... aa

5 ...... aaa

Note: ^x--- It will check whether target string starts with x or not

x$ ---- It will check whether target string ends with x or not

Important functions of re module:

1. match()

2. fullmatch()

3. search()

4.findall()

5.finditer()

6. sub()

7.subn()

8. split()

9. compile()

1. **match():**

   We can use match function to check the given pattern at beginning of target string.
   If the match is available then we will get Match object, otherwise we will get None.

EX:

```
import re
s=input("Enter pattern to check: ")
m=re.match(s,"abcabdefg")
if m!= None:
   print("Match is available at the beginning of the String")
   print("Start Index:",m.start(), "and End Index:",m.end())
else:
   print("Match is not available at the beginning of the String")
```

2. fullmatch():

We can use fullmatch() function to match a pattern to all of target string.

i.e complete string should be matched according to given pattern.

If complete string matched then this function returns Match object otherwise it returns None.

EX:

```
import re
s=input("Enter pattern to check: ")
m=re.fullmatch(s,"ababab")
if m!= None:
    print("Full String Matched")
else:
    print("Full String not Matched")
```

3. search():

We can use search() function to search the given pattern in the target string. If the match is available then it returns the Match object which represents first occurrence of the match. If the match is not available then it returns None

EX:

```
import re
s=input("Enter pattern to check: ")
m=re.search(s,"abaaaba")
if m!= None:
    print("Match is available")
    print("First Occurrence of match with start
index:",m.start(),"and end index:",m.end())
else:
    print("Match is not available")
```

4. findall():

To find all occurrences of the match.

This function returns a list object which contains all occurrences.

EX:

```
import re
l=re.findall("[0-9]","a7b9c5kz")
print(l)
```

5. finditer():

Returns the iterator yielding a match object for each match.
On each match object we can call start(), end() and group() functions.

Ex:

```
import re
itr=re.finditer("[a-z]","a7b9c5k8z")
for m in itr:
    print(m.start(),"...",m.end(),"...",m.group())
```

6. sub():

sub means substitution or replacement

re.sub(regex,replacement,targetstring)

In the target string every matched pattern will be replaced with provided replacement.

EX:

```
import re
s=re.sub("[a-z]","#","a7b9c5k8z")
print(s)
```

7. subn():

It is exactly same as sub except it can also returns the number of replacements.
This function returns a tuple where first element is result string and second element is number of replacements. (resultstring, number of replacements)

EX:

```
import re
t=re.subn("[a-z]","#","a7b9c5k8z")
print(t)
print("The Result String:",t[0])
print("The number of replacements:",t[1])
```

8. split():

If we want to split the given target string according to a particular pattern then we should go for split() function. This function returns list of all tokens.

EX:

```
import re

l = re.split(",", "sunny,bunny,chinny,vinny,pinny")
print(l)
```

```
for t in l:
    print(t)
```

EX:

```
import re

l = re.split("\.", "www.durgasoft.com")
print(l)
for t in l:
    print(t)
```

^ symbol:

We can use ^ symbol to check whether the given target string starts with our provided pattern or not.

Eg: res=re.search("^Learn",s) if the target string starts with Learn then it will return Match object,otherwise returns None.

EX:

```
import re
s="Learning Python is Very Easy"
res=re.search("^Learn",s)
if res != None:
    print("Target String starts with Learn")
else:
    print("Target String Not starts with Learn")
```

$ symbol:

We can use $ symbol to check whether the given target string ends with our provided pattern or not

Eg: res=re.search("Easy$",s)

If the target string ends with Easy then it will return Match object,otherwise returns None

EX:

```
import re
s="Learning Python is Very Easy"
res=re.search("Easy$",s)
if res != None:
    print("Target String Ends with Easy")
else:
    print("Target String Not Ends with Easy")
```

Write a Regular Expression to represent all 10 digit mobile numbers

Rules:

1. Every number should contains exactly 10 digits

2. The first digit should be 7 or 8 or 9

[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]

Or

[7-9][0-9]{9}

Or

[7-9]\d{9}

Write a Python Program to check whether the given number is valid mobile number or not?

Ex:

```
import re
n=input("Enter number:")
m=re.fullmatch("[7-9]\d{9}",n)
if m!= None:
   print("Valid Mobile Number")
else:
   print("Invalid Mobile Number")
```

Write a Python Program to check whether the given mail id is valid gmail id or not?

Ex:

```
import re
s=input("Enter Mail id:")
m=re.fullmatch("\w[a-zA-Z0-9_.]*@gmail[.]com",s)
if m!=None:
   print("Valid Mail Id");
else:
   print("Invalid Mail id")
```

Write a python program to check whether given car registration number is valid Telangana State Registration number or not?

EX:

```
import re
s=input("Enter Vehicle Registration Number:")
m=re.fullmatch("TS[012][0-9][A-Z]{2}\d{4}",s)
if m!=None:
   print("Valid Vehicle Registration Number");
else:
   print("Invalid Vehicle Registration Number")
```

Python Program to check whether the given mobile number is valid OR not (10 digit OR 11 digit OR 12 digit)

Ex:

```
import re
s=input("Enter Mobile Number:")
m=re.fullmatch("(0|91)?[7-9][0-9]{9}",s)
if m!=None:
   print("Valid Mobile Number");
else:
   print("Invalid Mobile Number")
```

As the part of programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files.

Files are very common permanent storage areas to store our data.

Types of Files:

There are 2 types of files

1. Text Files:

Usually we can use text files to store character data eg: abc.txt

2. Binary Files:

Usually we can use binary files to store binary data like images,video files, audio files etc...

Opening a File:

Before performing any operation (like read or write) on the file,first we have to open that file. For this we should use Python's inbuilt function open()

But at the time of open, we have to specify mode,which represents the purpose of opening file.

f = open(filename, mode)

The allowed modes in Python are:

1. r → open an existing file for read operation. The file pointer is positioned at the beginning of the file.If the specified file does not exist then we will get FileNotFoundError.This is default mode.

2. w ---> open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already avaialble then this mode will create that file.

3. a---> open an existing file for append operation. It won't override existing data.If the specified file is not already avaialble then this mode will create a new file.

4. r+ → To read and write data into the file. The previous data in the file will not be deleted.The file pointer is placed at the beginning of the file.

5. w+ → To write and read data. It will override existing data.

6. a+ →To append and read data from the file.It wont override existing data.

7. x →To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

Eg: rb,wb,ab,r+b,w+b,a+b,xb

f = open("abc.txt","w")

We are opening abc.txt file for writing data

Closing a File:

After completing our operations on the file,it is highly recommended to close the file. For this we have to use close() function.

f.close()

Various properties of File Object:

Once we opend a file and we got file object,we can get various details related to that file by using its properties.

name --- Name of opened file

mode ---Mode in which the file is opened

closed ---Returns boolean value indicates that file is closed or not

readable()--- Retruns boolean value indicates that whether file is readable or not

writable()--- Returns boolean value indicates that whether file is writable or not.

Ex:

```
f=open("abc.txt",'w')
print("File Name: ",f.name)
print("File Mode: ",f.mode)
print("Is File Readable:  ",f.readable())
print("Is File Writable:  ",f.writable())
print("Is File Closed : ",f.closed)
f.close()
print("Is File Closed : ",f.closed)
```

Writing data to text files:

We can write character data to the text files by using the following 2 methods.

write(str)

writelines(list of lines)

EX:

```
f=open("abcd.txt",'w')
f.write("Durga\n")
f.write("Software\n")
f.write("Solutions\n")
print("Data written to the file successfully")
f.close()
```

abcd.txt:

Durga

Software

Solutions

Note: In the above program, data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

f = open("abcd.txt","a")

Ex:

```
f=open("abcd.txt",'w')
list=["sunny\n","bunny\n","vinny\n","chinny"]
f.writelines(list)
print("List of lines written to the file
successfully")
f.close()
```

Note: while writing data by using write() methods, compulsory we have to provide line seperator(\n),otherwise total data should be written to a single line.

Reading Character Data from text files:

We can read character data from text file by using the following read methods

read()--- To read total data from the file

read(n) --- To read 'n' characters from the file

readline()---To read only one line

readlines()---To read all lines into a list

Eg 1: To read total data from the file

```
f=open("abc.txt",'r')
data=f.read()
print(data)
f.close()
```

Eg 2: To read only first 5 characters:

```
f=open("abcd.txt",'r')
data=f.read(10)
print(data)
f.close()
```

Eg 3: To read data line by line:

```
f = open("abc.txt", 'r')
line1 = f.readline()
print(line1, end='')
line2 = f.readline()
print(line2, end='')
line3 = f.readline()
print(line3, end='')
f.close()
```

Eg 4: To read all lines into list:

```
f=open("abc.txt",'r')
lines=f.readlines()
for line in lines:
    print(line,end='')
f.close()
```

The with statement:

The with statement can be used while opening a file.We can use this to group file operation statements within a block. The advantage of with statement is it will take care closing of file,after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

EX:

```
with open("abc.txt", "w") as f:
    f.write("Durga\n")
    f.write("Software\n")
    f.write("Solutions\n")
    print("Is File Closed: ", f.closed)
print("Is File Closed: ", f.closed)
```

The seek() and tell() methods:

tell():

==>We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [can you plese telll current cursor position] The position(index) of first character in files is zero just like string index.

Ex:

```
f = open("abcd.txt", "r")
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(3))
print(f.tell())
```

seek():

We can use seek() method to move cursor(file pointer) to specified location. [Can you please seek the cursor to a particular location]

f.seek(offset, fromwhere)
offset represents the number of positions

The allowed values for second attribute(from where) are
0---->From beginning of file(default value)
1---->From current position
2--->From end of the file

Note: Python 2 supports all 3 values but Python 3 supports only zero.

Ex:
```
data = "All Students are Good"
f = open("abc.txt", "w")
f.write(data)
with open("abc.txt", "r+") as f:
    text = f.read()
    print(text)
    print("The Current Cursor Position: ",
f.tell())
    f.seek(17)
    print("The Current Cursor Position: ",
f.tell())
    f.write("GEMS!!!")
    f.seek(0)
    text = f.read()
    print("Data After Modification:")
    print(text)
```

How to check a particular file exists or not?

We can use os library to get information about files in our computer.
os module has path sub module,which contains isFile() function to check whether a particular file exists or not? os.path.isfile(fname)

Write a program to check whether the given file exists or not. If it is available then print its content?

```
import os,sys
fname=input("Enter File Name: ")
if os.path.isfile(fname):
    print("File exists:",fname)
    f=open(fname,"r")
else:
    print("File does not exist:",fname)
    sys.exit(0)
print("The content of file is:")
data=f.read()
print(data)
```

Note: sys.exit(0) ===>To exit system without executing rest of the program. argument represents status code . 0 means normal termination and it is the default value

Zipping and Unzipping Files:

It is very common requirement to zip and unzip files. The main advantages are:

1. To improve memory utilization 2. We can reduce transport time 3. We can improve performance.

To perform zip and unzip operations, Python contains one in-bulit module zip file. This module contains a class : ZipFile

To create Zip file:

We have to create ZipFile class object with name of the zip file,mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

f = ZipFile("files.zip","w","ZIP_DEFLATED")

Once we create ZipFile object,we can add files by using write() method.

f.write(filename)

```
from zipfile import *

f = ZipFile("files.zip", 'w',
ZIP_DEFLATED)
f.write("file1.txt")
f.write("file2.txt")
f.write("file3.txt")
f.close()
print("files.zip file created
successfully")
```

To perform unzip operation:

We have to create ZipFile object as follows

f = ZipFile("files.zip","r",ZIP_STORED)

ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify. Once we created ZipFile object for unzip operation,we can get all file names present in that zip file by using namelist() method.

names = f.namelist()

```
from zipfile import *
f=ZipFile("files.zip",'r',ZIP_STORED)
names=f.namelist()
for name in names:
        print(   "File Name:  ",name)
        print("The Content of this file is:")
        f1=open(name,'r')
        print(f1.read())
        print()
```

Logging the Exceptions:

It is highly recommended to store complete application flow and exceptions information to a file. This process is called logging.

The main advantages of logging are:

1. **We can use log files while performing debugging**
2. **We can provide statistics like number of requests per day etc**

To implement logging, Python provides one inbuilt module logging.

logging levels:

Depending on type of information, logging data is divided according to the following 6 levels in Python.

1. CRITICAL==>50==>Represents a very serious problem that needs high attention

2. ERROR===>40===>Represents a serious error

3. WARNING==>30==>Represents a warning message, some caution needed. It is alert to the programmer

4. INFO===>20===>Represents a message with some important information

5. DEBUG===>10==>Represents a message with debugging information

6. NOTSET==>0==>Rrepresents that the level is not set.

By default while executing Python program only WARNING and higher level messages will be displayed.

How to implement logging:

To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store.

We can do this by using basicConfig() function of logging module.

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

The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file, we can write messages to that file by using the following methods.

logging.debug(message)

logging.info(message)

logging.warning(message)

logging.error(message)

logging.critical(message)

.Write a Python program to create a log file and write WARNING and higher level messages?

EX:

```
import logging

logging.basicConfig(filename='log.txt
', level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug
message")
logging.info("This is info message")
logging.warning("This is warning
message")
logging.error("This is error
message")
logging.critical("This is critical
message")
```

Note: In the above program only WARNING and higher level messages will be written to log file. If we set level as DEBUG then all messages will be written to log file.

Ex:

```
import logging
logging.basicConfig(filename='log.txt
',level=logging.DEBUG)
print("Logging Module Demo")
logging.debug("This is debug
message")
logging.info("This is info message")
logging.warning("This is warning
message")
logging.error("This is error
message")
logging.critical("This is critical
message")
```

Note: We can format log messages to include date and time, ip address of the client etc at advanced level.

How to write Python program exceptions to the log file:

By using the following function we can write exceptions information to the log file.

logging.exception(msg)

Ex:

Python Program to write exception information to the log file

EX:

```
import logging
logging.basicConfig(filename='mylog.t
xt',level=logging.INFO)
logging.info("A New request Came:")
try:
        x=int(input("Enter First
Number: "))
        y=int(input("Enter Second
Number: "))
        print(x/y)
except ZeroDivisionError as msg:
        print("cannot divide with
zero")
        logging.exception(msg)
except ValueError as msg:
        print("Enter only int
values")
        logging.exception(msg)
logging.info("Request Processing
Completed")
```

Python DataBase Programming:

Storage Areas As the Part of our Applications, we required to store our Data like Customers Information, Billing Information, Calls Information etc..

To store this Data, we required Storage Areas.

There are 2 types of Storage Areas.

1) Temporary Storage Areas

2) Permanent Storage Areas

1.Temporary Storage Areas:

These are the Memory Areas where Data will be stored temporarily. Eg: Python objects like List, Tuple, Dictionary.

Once Python program completes its execution then these objects will be destroyed automatically and data will be lost.

2. Permanent Storage Areas:

Also known as Persistent Storage Areas. Here we can store Data permanently. Eg: File Systems, Databases, Data warehouses, Big Data Technologies etc

Databases:

1) We can store Huge Amount of Information in the Databases.

2) Query Language Support is available for every Database and hence we can perform Database Operations very easily.

3) To access Data present in the Database, compulsory username and pwd must be required. Hence Data is secured.

4) Inside Database Data will be stored in the form of Tables. While developing Database Table Schemas, Database Admin follow various Normalization Techniques and can implement various Constraints like Unique Key Constrains, Primary Key Constraints etc which prevent Data Duplication. Hence there is no chance of Data Inconsistency Problems.

Limitations of Databases:

1) Database cannot hold very Huge Amount of Information like Terabytes of Data.

2) Database can provide support only for Structured Data (Tabular Data OR Relational Data) and cannot provide support for Semi Structured Data (like XML Files) and Unstructured Data (like Video Files, Audio Files, Images etc) To overcome these Problems we should go for more Advanced Storage Areas like Big Data Technologies, Data warehouses etc.

Python Database Programming: Sometimes as the part of Programming requirement we have to connect to the database and we have to perform several operations like creating tables, inserting data,updating data,deleting data,selecting data etc.

We can use SQL Language to talk to the database and we can use Python to send those SQL commands to the database.

Python provides inbuilt support for several databases like Oracle, MySql, SqlServer, GadFly, sqlite, etc. Python has seperate module for each database. Eg: cx_Oralce module for communicating with Oracle database pymssql module for communicating with Microsoft Sql Server

```
import pyodbc
con=pyodbc.connect("Driver={SQL
Server};server=.;database=ASP")
cur=con.cursor()
cur.execute("select name from emp")
for row in cur:
    print(row.name)
cur.close()
con.close()
```

```
import pyodbc

try:

    con=pyodbc.connect("Driver={SQL
Server};server=.;database=ASP")

    cursor=con.cursor()

    cursor.execute('create table
py(eno int,ename varchar(10),esal
int)')

    print("Table created
successfully")
```

```
except pyodbc.DatabaseError as e:
    if con:
        con.rollback()
        print("There is a problem 
with sql",e)
finally:
    if cursor:
        cursor.close()
    if con:
        con.close()
```