Python Programming

MODULE - I

Agenda:

- Python Basics,
- Getting started,
- Python Objects,
- Numbers,
- Sequences:
- Strings,
- Lists,
- Tuples,
- Set and Dictionary.
- Conditionals and Loop Structures

Python Basics

- **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 was invented by **Guido van Rossum** in 1991 at CWI in Netherland.
- The idea of Python programming language has taken from the ABC programming language or we can say that ABC is a predecessor of Python language.
- There is also a fact behind the choosing name Python. Guido van Rossum was a fan of the popular BBC comedy show of that time, "**Monty Python's Flying Circus**". So he decided to pick the name Python for his newly created programming language.
- Python has the vast community across the world and releases its version within the short period.
- Python is *easy to learn* yet powerful and versatile scripting language, which makes it attractive for Application Development.
- Python's syntax and *dynamic typing* with its interpreted nature make it an ideal `language for scripting and rapid application development.
- Python supports *multiple programming pattern*, including object-oriented, imperative, and functional or procedural programming styles.
- Python is not intended to work in a particular area, such as web programming. That is why it is known as *multipurpose* programming language because it can be used with web, enterprise, 3D CAD, etc.
- 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.
- Python makes the development and debugging *fast* because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

Features of Python:

Python provides many useful features to the programmer. These features make it most popular and widely used language. We have listed below few-essential feature of Python.

- Easy to use and Learn
- Open Source Language
- Platform Independent:

- Portability
- Dynamically Typed
- Procedure Oriented and Object Oriented
- Interpreted
- Extensible
- Embeddable
- Extensive Library

Easy to use and learn:

Python is a simple programming language. When we read Python program, we can feel like Reading English statements. The syntaxes are very simple and only 30+ keywords are available. When compared with other languages, we can write programs with very less number of lines. Hence more readability and simplicity.

Open Source Language:

We can use Python software without any licence and it is freeware.Its source code is open,so that we can we can customize based on our requirement.
Eg: Jython is customized version of Python to work with Java Applications.

Platform Independent:
Once we write a Python program, it can run on any platform without rewriting once again. Internally PVM is responsible to convert into machine understandable form.

Portability:
Python programs are portable. ie we can migrate from one platform to another platform very easily. Python programs will provide same results on any paltform.

Dynamically Typed:
In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically.Hence Python is considered as dynamically typed language.But Java, C etc are Statically Typed Languages because we have to provide type at the beginning only.

Procedure Oriented and Object Oriented:
Python language supports both Procedure oriented (like C, pascal etc) and object oriented (like C++,Java) features. Hence we can get benefits of both like security and reusability etc

Interpreted:

We are not required to compile Python programs explcitly. Internally Python interpreter will take care that compilation. If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

Extensible:

We can use other language programs in Python,The main advantages of this approach are:

1. We can use already existing legacy non-Python code
2. We can improve performance of the application

Embedded:

We can use Python programs in any other language programs. i.e we can embedd Python programs anywhere.

Extensive Library:

Python has a rich inbuilt library.Being a programmer we can use this library directly and we are not responsible to implement the functionality.

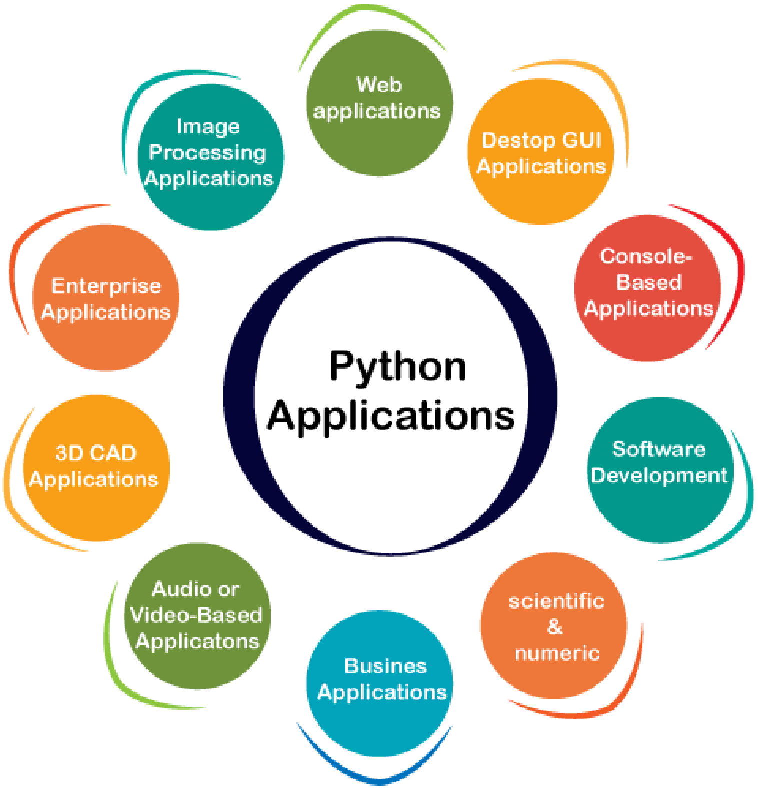
Versions of Python:

Python Version	Released Date
Python 1.0.0	January 1994
Python 1.5.0	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 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 3, 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
Python 3.8	October 14, 2019
Python 3.9	October 2020

Python Applications:

- The following are different area we can use python programming language



Input and output functions

In Python 2 the following 2 functions are available to read dynamic input from the keyboard.

1. raw_input()
2. input()

1. raw_input():

This function always reads the data from the keyboard in the form of String Format. We have to convert that string type to our required type by using the corresponding type casting methods.

Eg:

```
x=raw_input("Enter a Value:")
print(type(x)) It will always print str type only for any input type
```

2. input():

input() function can be used to read data directly in our required format.We are not required to perform type casting.

Eg:

```
x=input("Enter a Value)
type(x)
20 ===> int
"DS"===>str
125.5===>float
True==>bool
```

- In Python 3 we have only input() method and raw_input() method is not available.
- Python3 input() function behaviour exactly same as raw_input() method of Python2. i.e every input value is treated as str type only.

Example:

```
x=input("Enter First Number:")
y=input("Enter Second Number:")
a = int(x)
b = int(y)
print("Sum=",a+b)
```

output:

```
Enter First Number:10
Enter Second Number:20
Sum=30
```

OutPut Function:

We use the print() function or print keyword to output data to the standard output device (screen). This function prints the object/string written in function

Examples:

```
print("Hello World")
We can use escape characters also
 print("Hello \n World")
 print("Hello\tWorld")
 We can use repetetion operator (*) in the string
 print(10*"Hello")
print("Hello"*10)
```

We can use + operator also

```
print("Hello"+"World")
```

Python Comments:

- Python Comment is an essential tool for the programmers.
- Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation.
- A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.
- In the other programming language such as C, It provides the // for single-lined comment and /*.... */ for multiple-lined comment, but Python provides the single-lined Python comment.
- To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Let's understand the following example.

```
# This is the print statement
print("Hello Python")
```

Here we have written comment over the print statement using the hash(#). It will not affect our print statement.

Docstring in Python

- Python has the documentation strings (or docstrings) feature. It gives programmers an easy way of adding quick notes with every Python module, function, class, and method.
- You can define a docstring by adding it as a string constant. It must be the first statement in the object’s (module, function, class, and method) definition.
- The docstring has a much wider scope than a Python comment. Hence, it should describe what the function does, not how. Also, it is a good practice for all functions of a program to have a docstring.
- The strings defined using triple-quotation mark are docstring in Python. However, it might appear to you as a regular comment

Let's understand the following example.

```
'''
hello good morning
welcome to python
'''
print("Doc Sting")
```

```
“””
hello good morning
welcome to python
“””
print("Doc Sting")
```

Identifiers:

- A name in Python program is called identifier.
- It can be class name or function name or module name or variable name
- The following rules we have to follow while creating an didentifiers

1. Alphabet Symbols (Either Upper case OR Lower case)
2. If Identifier is start with Underscore (_) then it indicates it is private.
3. Identifier should not start with Digits.
4. Identifiers are case sensitive.
5. We cannot use reserved words as identifiers

Eg: def=10

6. There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.
7. Dollor ($) Symbol is not allowed in Python.

- The following are Examples
- myVar
- var_3
- cse_ds

Reserved Words

- In Python some words are reserved to represent some meaning or functionality. Such type of words are called Reserved words.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.
- All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	If	or	yield

Data Types or Objects

- Python is an object-oriented programming language, and in Python everything is an object.
- Objects are also called as Data structures.
- All the Data types in python are also called as Data types

- Data Type represents the type of data present inside a variable.
- In Python we are not required to specify the type explicitly. Based on value provided,the type will be assigned automatically.Hence Python is **Dynamically Typed Language.**

Python contains the following inbuilt data types are categorized as follows

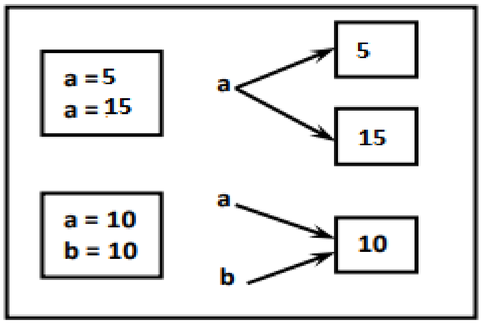
- **Fundamental or Build-in Data types or Data Structures**
- **Composite Data Types or Data Structures**

Object Type	**Description**	**Example**
Fundamental or Build-in Data types or Data Structures		
1. int	We can use to represent the whole/integral numbers	26,10,-12,-26
2. float	We can use to represent the decimal/floating point numbers	26.6,-26.2
3. complex	We can use to represent the complex numbers	26+26j
4. bool	We can use to represent the logical values(Only allowed values are True and False)	True,False
5. str	To represent sequence of Characters	“MREC “,”Raj”
Composite Data Types or Data Structures		

6. range	To represent a range of values	r=range(26)
		r1=range(1,26)
		r2=range(1,2,3)

7. list	To represent an ordered collection of objects	L1=[1,2,3,4,5,]
8. tuple	To represent an ordered collections of objects	t=(1,2,3,4,5)
9. set	To represent an unordered collection of unique objects	S={1,2,3,4,5}
10. dict	To represent a group of key value pairs	d={1:'Raj',2:'Sekhar'}
11. None	None means Nothing or No value associated.	a=None

Example:



Python contains several inbuilt functions as follows:

1.type() : to check the type of variable

2. id(): to get address of object

3. print(): to print the value

Example:

```
>>> a=10
>>> type(a)
<class 'int'>
>>> id(a)
2141527304784
>>> print(a)
10
```

Fundamental or Build-in Data types or Data Structures

1. int data type:

- We can use int data type to represent whole numbers (integral values)

Eg: a=10

type(a) #int

We can represent int values in the following ways

1. Decimal form
2. Binary form
3. Octal form
4. Hexa decimal form

1. Decimal form(base-10):

It is the default number system in Python

The allowed digits are: 0 to 9

Eg: a =10

2. Binary form(Base-2):

The allowed digits are : 0 & 1

Literal value should be prefixed with 0b or 0B

Eg: a = 0B1111

a =0B123

a=b111

3. Octal Form(Base-8):

The allowed digits are : 0 to 7

Literal value should be prefixed with 0o or 0O

Eg: a=0o123

a=0o786

4. Hexa Decimal Form(Base-16):

The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)

Literal value should be prefixed with 0x or 0X

Eg:

```
 a =0XFACE
 a=0XBeef
 a =0XBeer
```

Example:

```
>>> a=10
>>> b=0B0101
>>> c=0o121
>>> d=0xabc
>>> print(a)
10
>>> print(b)
5
>>> print(c)
81
>>> print(d)
2748
```

Base Conversions

Python provide the following in-built functions for base conversions

1.bin():

We can use bin() to convert from any base to binary

Eg:

```
>>> bin(5)
 '0b101'
>>> bin(0o11)
 '0b1001'
 >>> bin(0X10)
 '0b10000'
```

2. oct():

We can use oct() to convert from any base to octal

Eg:

```
>>> oct(10)
 '0o12'
 >>> oct(0B1111)
 '0o17'
```

```
>>> oct(0X123)
'0o443'
```

3. hex():

We can use hex() to convert from any base to hexa decimal

Eg:

```
>>> hex(100)
'0x64'
>>> hex(0B111111)
'0x3f'
>>> hex(0o12345)
'0x14e5'
```

2. float data type:

We can use float data type to represent floating point values (decimal values)

Eg: f=1.234
type(f) float

We can also represent floating point values by using exponential form (scientific notation)

Eg: f=1.2e3
print(f) 1200.0

instead of 'e' we can use 'E'

- The main advantage of exponential form is we can represent big values in less memory.
- We can represent int values in decimal, binary, octal and hexa decimal forms. But we can represent float values only by using decimal form.

Eg:

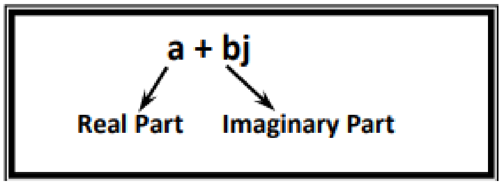
```
 >>> f=0B11.01
 File "<stdin>", line 1
 f=0B11.01
SyntaxError: invalid syntax

 >>> f=0o123.456
 SyntaxError: invalid syntax

 >>> f=0X123.456
 SyntaxError: invalid syntax
```

3. Complex Data Type:

A complex number is of the form



a and b contain intergers or floating point values

Eg:

```
6+3j
9+9.5j
0.5+0.9j
```

In the real part if we use int value or we can specify that either by decimal,octal,binary or hexa decimal form. But imaginary part should be specified only by using decimal form.

```
>>> a=0B011+4j
>>> a
 (3+4j)
 >>> a=3+0B011j
 SyntaxError: invalid syntax
```

Even we can perform operations on complex type values.

```
 >>> a=9+2.5j
 >>> b=4+3.9j
 >>>print(a+b)
(13+6.4j)
>>> a=(20+5j)
>>> type(a)
 <class 'complex'>
```

- Complex data type has some inbuilt attributes to retrieve the real part and imaginary part

```
c=15.4+6.6j
c.real==>15.4
c.imag==>6.6
```

- We can use complex type generally in scientific Applications and electrical engineering Applications

4. bool data type:

- We can use this data type to represent boolean values.
- The only allowed values for this data type are:**True and False**
- Internally Python represents True as 1 and False as 0

```
b=True
type(b) =>bool
```

Eg:

a=20
b=30
c=a<b
print(c)==>True
True+True==>2
True-False==>1

5. str type:

- str represents String data type.
- A String is a sequence of characters enclosed within single quotes or double quotes.
  s1='MREC'
  s1="MREC"
- By using single quotes or double quotes we cannot represent multi line string literals.
  s1="MREC DS"
- For this requirement we should go for triple single quotes(''') or triple double quotes(""")
  s1='''MREC
  DS'''
  s1="""MREC
  DS"""
- We can also use triple quotes to use single quote or double quote in our String.
  >>> s1="""This is mrec"""
  >>> s1
  '''This is mrec'''
- We can embed one string in another string
  >>> s1='''This "Python Programming Session" for DS Students'''
  >>> s1
  'This "Python Programming Session" for DS Students'

Slicing of Strings:

- slice means a piece
- [ ] operator is called slice operator,which can be used to retrieve parts of String.
- In Python Strings follows zero based index.
- The index can be either +ve or -ve.
- +ve index means forward direction from Left to Right
- -ve index means backward direction from Right to Left

Eg:

-7 -6 -5 -4 -3 -2 -1

M	R	E	C		D	S
0	1	2	3	4	5	6

```
>>> s="MREC DS"
>>> s[0]
'M'
>>> s[-7]
'M'
>>> s[3]
'C'
>>> s[-4]
'C'
>>> s[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[50]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> s[1:4]
'REC'
>>> s[0:4]
'MREC'
>>> s[0:]
'MREC DS'
>>> s[:4]
'MREC'
>>> s[:]
'MREC DS'
>>> len(s)
7
```

Type Casting in Python

We can convert one type value to another type. This conversion is called Typecasting or Type conversion.

The following are various inbuilt functions for type casting.

1. int()
2. float()

3. complex()
4. bool()
5. str()

1.int():

- We can use this function to convert values from other types to int Type.
- We can convert from any type to int except complex type.
- we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10

Eg:

```
1) >>> int(13.87)
2) 13
4) >>> int(True)
5) 1
6) >>> int(False)
7) 0
8) >>> int("19")
10) 19
11) >>> int(10+5j)
12) TypeError: can't convert complex to int
13) >>> int("10.5")
14) ValueError: invalid literal for int() with base 10: '10.5'
15) >>> int("ten")
16) ValueError: invalid literal for int() with base 10: 'ten'
17) >>> int("0B1111")
18) ValueError: invalid literal for int() with base 10: '0B1111'
```

2. float():

- We can use float() function to convert other type values to float type.
- We can convert any type value to float type except complex type.
- Whenever we are trying to convert str type to float type compulsary str should be either integral or floating point literal and should be specified only in base-10.

Eg:

```
1) >>> float(26)
2) 26.0
3) >>> float(True)
4) 1.0
5) >>> float(False)
6) 0.0
7) >>> float("26")
8) 26.0
```

```
>>> float("26.5")
26.5
>>> float(26+5j)
TypeError: can't convert complex to float
>>> float("ten")
ValueError: could not convert string to float: 'ten'
>>> float("0B1011")
ValueError: could not convert string to float: '0B1011'
```

3.complex():

- We can use complex() function to convert other types to complex type.
- We can use this function to convert x into complex number with real part x and imaginary
- We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

Eg:

```
1)  complex(26)
      26+0j
2) complex(26.26)
     26.26+0j
3) complex(True)
      1+0j
4) complex(False)
       0j
5) complex("26")
     26+0j
6) complex("26.26")
     26.26+0j
7) complex("MREC")
ValueError: complex() arg is a malformed string
8)complex(26,26)
     26+26j
 9)complex(True,False)
    1+0j
```

4. bool():

- We can use this function to convert other type values to bool type.

Eg:

```
1)  bool(0)
```

```
      False
2)  bool(1)
      True
3)  bool(26)
      True
4)  bool(26.26)
      True
5)  bool(0.26)
      True
6)  bool(0.0)
      False
7)  bool(26-26j)
      True
8)  bool(0+26.26j)
      True
9)  bool(0+0j)
      False
10) bool("True")
      True
11) bool("False")
      True
12) bool("")
      False
```

5. str():

We can use this method to convert other type values to str type

Eg:

```
1) >>> str(26)
         '26'
3) >>> str(26.26)
          '26.26'
5) >>> str(26+5j)
         '(26+5j)'
7) >>> str(True)
         'True'
8)>>>str(False)
         ‘False’
```

Operators in Python

An operator is a symbol that tells the compiler to perform certain mathematical or logical Manipulations. Operators are used in program to manipulate data and variables.
Python language supports the following types of operators.

1. Arithmetic Operators
2. Relational Operators or Comparison Operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Special operators

1. Arithmetic Operators:

Arithmetic operators are used with numeric values to perform common mathematical operations:

- / operator always performs floating point arithmetic. Hence it will always returns float value.
- Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is float type then result is float type.

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Name	Example
+	Addition - Adds values on either side of the operator	x + y=7
-	Subtraction - Subtracts right hand operand from left hand operand	x – y=3
*	Multiplication - Multiplies values on either side of the operator	x * y=10
/	Division - Divides left hand operand by right hand operand	x / y=2.5
%	Modulus - Divides left hand operand by right hand operand and returns remainder	x % y=1
**	Exponent - Performs exponential (power) calculation on operators	x ** y=25
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	x // y=2

Eg:

```
>>> x=5
>>> y=2
>>> print('x+y=',x+y)
x+y= 7
```

```
>>> print('x-y=',x-y)
x-y= 3
>>> print('x*y=',x*y)
x*y= 10
>>> print('x/y=',x/y)
x/y= 2.5
>>> print('x%y=',x%y)
x%y= 1
>>> print('x**y=',x**y)
x**y= 25
>>> print('x//y=',x//y)
x//y= 2
```

2. Relational Operators or Comparison Operators

Comparison operators are used to compare two values:

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Name	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true	x == y=False
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	x != y=True
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	x > y=True
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	x < y=False
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	x >= y=True
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	x <= y=False

Eg:

```
>>> x=5
>>> y=2
>>> print('x==y=',x==y)
x==y= False
>>> print('x!=y=',x!=y)
x!=y= True
>>> print('x>y=',x>y)
x>y= True
```

```
>>> print('x<y=',x<y)
x<y= False
>>> print('x>=y=',x>=y)
x>=y= True
>>> print('x<=y=',x<=y)
x<=y= False
```

3. Logical operators:

Logical operators are used to combine conditional statements:

X	Y	X AND Y	X OR Y	NOT X
False	False	False	False	True
False	True	False	True	True
Ture	False	False	True	False
True	True	True	True	False

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Eg:

```
>>> x=5
>>> y=2
>>> x and y
2
>>> print(x>=5 and y<=5)
True
>>> print(x>=5 or y<=5)
True
>>> print(not x>=5)
False
```

4. Bitwise operators:

- Bitwise operator works on bits and performs bit by bit operation.
- We can apply these operators bitwise on int and boolean types.
- By mistake if we are trying to apply for any other type then we will get Error.

Truth table for bit wise operation

x	Y	x|y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Bit wise operators

Operator_symbol	Operator_name
&	Bitwise_AND
|	Bitwise OR
~	Bitwise_NOT
^	XOR
<<	Left Shift
>>	Right Shift

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Eg:

```
>>> x=5
>>> y=2
>>> print('x & y=',x&y)
x & y= 0
>>> print('x | y=',x|y)
x | y= 7
>>> print('X ^ y=',x^y)
X ^ y= 7
>>> print('~x=',~x)
~x= -6
>>> print('x>>1=',x>>1)
x>>1= 2
>>> print('y<<1=',y<<1)
y<<1= 4
```

6.Assignment operators:

Assignment operators are used to assign values to variables:

Operator	Example	Equal to
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Eg:

```
>>> x=5
>>> x+=3
>>> print('x=x+3=',x)
x=x+3= 8
>>> x-=3
>>> print('x=x-3=',x)
x=x-3= 5
>>> x*=3
>>> print('x=x*3=',x)
x=x*3= 15
>>> x/=3
>>> print('x=x/3=',x)
x=x/3= 5.0
>>> x%=3
>>> print('x=x%3=',x)
x=x%3= 2.0
>>> x//=3
>>> print('x=x//3=',x)
x=x//3= 0.0
```

```
>>> x**=3
>>> print('x=x**3=',x)
x=x**3= 0.0
>>> x=5
>>> x&=3
>>> print('x=x&3=',x)
x=x&3= 1
>>> x|=3
>>> print('x=x|3=',x)
x=x|3= 3
>>> x^=3
>>> print('x=x^3=',x)
x=x^3= 0
>>> x>>=3
>>> print('x=x>>3=',x)
x=x>>3= 0
>>> x<<=3
>>> print('x=x<<3=',x)
x=x<<3=0
```

5. Special operators:

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operators

- Identity Operators in Python are used to compare the memory location of two objects. The two identity operators used in Python are (is, is not).

- Operator is: It returns true if two variables point the same object and false otherwise
- Operator is not: It returns false if two variables point the same object and true otherwise2 identity operators are available.

Operator	**Description**	**Example**
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Eg:

```
>>> x=5
>>> y=5
>>> print(x is y)
True
>>> print(id(x))
2265011481008
>>> print(id(y))
2265011481008
>>> print(x is not y)
False
```

2. Membership Operators

- These operators test for membership in a sequence such as lists, strings or tuples. There are two membership operators that are used in Python. (in, not in). It gives the result based on the variable present in specified sequence or string
- For example here we check whether the value of x=4 and value of y=8 is available in list or not, by using in and not in operators.

Operator	**Description**	**Example**
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Eg:

```
>>> x="MREC CSE-DS Dept"
>>> print('M' in x)
True
>>> print('-' in x)
True
>>> print('DS' in x)
True
>>> print('1' not in x)
True
>>> print('M' not in x)
False
```

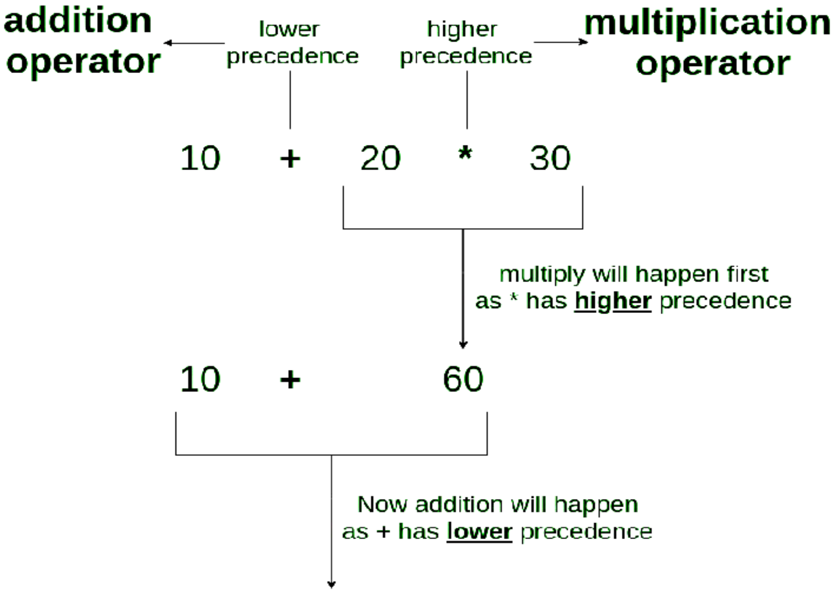
Precedence and Associativity of Operators in Python

- When an expression has more than one operator, then it is the relative priorities of the operators with respect to each other that determine the order in which the expression is evaluated.

Operator Precedence: This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Eg:10+20*30

10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30



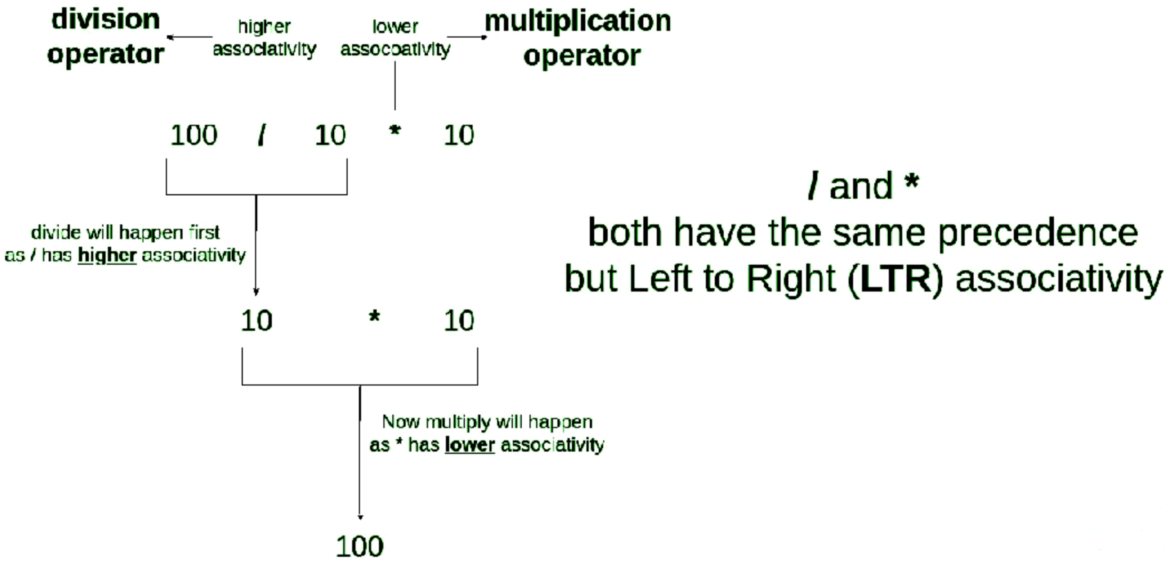
Example:

```
>>> exp=10+20*30
>>> print(exp)
610
```

Operator Associativity:

- When two operators have the same precedence, associativity helps to determine the order of operations.
- Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.
- For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, the left one is evaluated first.

Example: '*' and '/' have the same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".


division operator
higher associativity
lower assocoativity
multiplication operator
100 / 10 * 10
divide will happen first as / has higher associativity
10 * 10
Now multiply will happen as * has lower associativity
100
/ and *
both have the same precedence
but Left to Right (LTR) associativity

Example:

```
>>> exp=100/10*10
>>> print(exp)
100.0
```

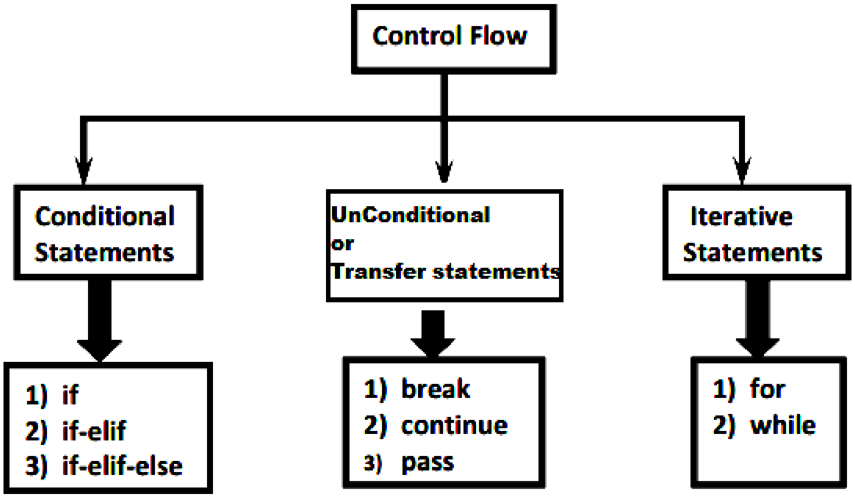
- Please see the following precedence and associativity table for reference. This table lists all operators from the highest precedence to the lowest precedence.

Operator	Description	Associativity
( )	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right

&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
Not	Logical NOT	right-to-left
And	Logical AND	left-to-right
Or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

Conditionals and Loop Structures

A control structure directs the order of execution of the statements in program.The Control statements as categorized as follows.



Conditional statement

- ➢ Conditional statements will decide the execution of a block of code based on the expression.
- ➢ The conditional statements return either True or False.
- ➢ A Program is just a series of instructions to the computer, But the real strength of Programming isn't just executing one instruction after another. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. Flow control statements can decide which Python instructions to execute under which conditions.
- ➢ Python supports four types of conditional statements,

1. Simple if or if statement
2. if – else Statement
3. if else if (elif) Statement
4. nested if statement

Indentation: Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

1) Simple if or if statement

```
if condition : statement
 or
if condition :
            statement-1
            statement-2
            statement-3
```

If condition is true then statements will be executed

Example:

```
>>> a=10
>>> b=5
>>> if(a>b):
            print("a is big")

a is big
>>> if a>b:
            print("a  is big")
a is big
```

2) if else:

```
if condition :
 Statements-1
else :
 Statements-2
```

if condition is true then Statements-1 will be executed otherwise Statements-2 will be executed.

Example:

```
>>> a=10
>>> b=25
>>> if(a>b):
            print("a is big")
else:
            print("b is big")
```

```
b is big
```

3) if elif else:

Syntax:

```
if condition1:
 Statements-1
elif condition2:
Statements -2
elif condition3:
Statements -3
elif condition4:
Statements -4
 …
else:
 Default Action
```

Based condition the corresponding action will be executed.

Example:

```
>>> Option=int(input("Enter a value b/w(1-5)"))
Enter a value b/w(1-5)2
>>> if(Option==1):
        print("you entered one")
elif(Option==2):
        print("You entered Two")
elif(Option==3):
        print("You entered Three")
elif(Option==4):
        print("You entered Four")
elif(Option==5):
        print("You entered Five")
else:
        print("Enter Value b/w (1-5) only")

You entered Two
```

4. nested if statement

We can use if statements inside if statements, this is called nested if statements.

Synatx:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Example:

```
>>> username=input("enter user name:")
enter user name:Raj
>>> pwd=input("Enter password")
Enter passwordRaj
>>> if(username=="Raj"):
            if(pwd=="Raj"):
                        print("Login successful:")
            else:
                        print("Invalid pwd")
else:
            print("Invalid Username")

Login successful:
```

Iterative Statements

If we want to execute a group of statements multiple times then we should go for Iterative statements.

Python supports 2 types of iterative statements.

1. **for loop**
2. **while loop**

1) for loop:

If we want to execute some action for every element present in some sequence(it may be string or collection)then we should go for for loop.

Syntax:

```
for x in sequence :
             body
```

Where sequence can be string or any collection.

Body will be executed for every element present in the sequence.

Eg 1: To print characters present in the given string

```
>>> s="MREC"
>>> for r in s:
            print(r)
M
```

```
R
E
C
```

Eg2: To print characters present in string index wise:

```
>>> i=0
>>> for x in s:
        print('The character present at ',i,'index:',x)
        i+=1

The character present at  0 index: M
The character present at  1 index: R
The character present at  2 index: E
The character present at  3 index: C
```

Eg3: To print Sequence of values:

```
>>> for i in (1,2,3,4,5):
        print(i)

1
2
3
4
5
```

2) while loop:

If we want to execute a group of statements iteratively until some condition false,then we should go for while loop.

Syntax:

```
while condition :
         body
```

Eg: To print numbers from 1 to 5 by using while loop

```
>>> i=1
>>> while(i<=5):
        print(i)
        i+=1
1
2
3
4
5
```

Eg: To display the sum of first n numbers

```
n=int(input("Enter n value:"))
sum=0
i=1
while i<=n:
   sum=sum+i
   i=i+1
print("sum of ",n," elements are=",sum)
```

OutPut:

```
Enter n value:5
sum of  5  elements are= 15
```

Nested Loops:

- Sometimes we can take a loop inside another loop,which are also known as nested loops
- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

Syntax:

```
while expression:
         while expression:
                  statement(s)
     statement(s)
```

Eg1:

```
r=1
while(r<=3):
   c=1
   while(c<=5):
      print("r=",r,"c=",c)
      c=c+1
   print('\n')
   r=r+1
```

OutPut:

```
r= 1 c= 1       r=2   c=1       r=3   c=1
r= 1 c= 2       r=2   c=2       r=3   c=2
r= 1 c= 3       r=2   c=3       r=3   c=3
r= 1 c= 4       r=2   c=4       r=3   c=4
r= 1 c= 5       r=2   c=5       r=3   c=5
```

Eg2:

```
for r in (1,2,3):
   for c in (1,2,3,4,5):
```

```
            print('r=',r,'c=',c)
        print('\n')
```

OutPut:

```
r= 1 c= 1      r=2   c=1      r=3   c=1
r= 1 c= 2      r=2   c=2      r=3   c=2
r= 1 c= 3      r=2   c=3      r=3   c=3
r= 1 c= 4      r=2   c=4      r=3   c=4
r= 1 c= 5      r=2   c=5      r=3   c=5
```

Transfer Statements

1) break:

- ➢ We can use break statement inside loops to break loop execution based on some condition.

Eg:

```
for r in (1,2,3,4,5):
    if(r==3):
        print("Break the loop")
        break
    print(r)
```

OutPut:

```
1
2
Break the loop
```

2) continue:

- ➢ We can use continue statement to skip current iteration and continue next iteration.

Eg 1: To print even numbers in the range 1 to 10

```
for r in (1,2,3,4,5,6,7,8,9,10):
    if(r%2!=0):
        continue
    print(r)
```

OutPut:

```
2
4
6
8
10
```

Composite Data Types or Data Structures

- The following are different Composite data type in python

6. range Data Type:

- range Data Type represents a sequence of numbers. The elements present in range Data type are not modifiable. i.e range Data type is immutable
- We can access elements present in the range Data Type by using index.

Eg:

1. **range(5)→generate numbers from 0 to 4**

   Eg:

   ```
   r=range(5)
   for i in r :
           print(i)
   OutPut: 0 1 2 3 4 5
   ```

2. **range(5,10)→generate numbers from 5 to 9**

   ```
   r = range(5,10)
   for i in r :
           print(i)
   OutPut:5 6 7 8 9
   ```

3. **range(1,10,2)→2 means increment value**

   ```
   r = range(1,10,2)
   for i in r :
           print(i)
   OutPut: 1 3 5 7 9
   ```

4.
   ```
   r=range(0,5)
   r[0]==>0
   r[15]==>IndexError: range object index out of range
   ```

   We cannot modify the values of range data type

7.list data type:

- If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.
- An ordered, mutable, heterogeneous collection of elements is nothing but list, where Duplicates also allowed.
- insertion order is preserved
- heterogeneous objects are allowed
- duplicates are allowed
- Growable in nature
- values should be enclosed within square brackets.

   1. **Eg:**

```
 list=[26,26.5,'Raj',True]
 print(list)
output→ [26,26.5,'Raj',True]
```

2. Eg:

```
 list=[10,20,30,40]
 >>> list[0]
 10
 >>> list[-1]
 40
 >>> list[1:3]
 [20, 30]
 >>> list[0]=100
 >>> print(list)
 …
 100
 40
 30
 40
```

- ➢ list is growable in nature. i.e based on our requirement we can increase or decrease the size.

```
>>> list=[10,20,30]
>>> list.append("raj")
>>> list
[10, 20, 30, ‘raj’]
>>> list.remove(20)
 >>> list
 [10, 30, ‘raj’]
 >>> list1=list*2
>>> list1
 [10, 30, 'raj', 10, 30, 'raj']
```

Creating list by using range data type:

- ➢ We can create a list of values with range data type

Eg:

```
>>> l = list(range(5))
>>>print(l)
[0, 1, 2, 3, 4]
```

8. tuple data type:

- ➢ tuple data type is exactly same as list data type except that it is immutable.i.e we cannot chage values.
- ➢ Tuple elements can be represented within parenthesis.
- ➢ tuple is the read only version of list

Eg:

```
>>> t1=(1,2,3,4)
>>>type(t)
<class 'tuple'>
>>>t1[0]=26
TypeError: 'tuple' object does not support item assignment
>>> t.append("Raj")
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove(2)
AttributeError: 'tuple' object has no attribute 'remove'
```

9. set Data Type:

- ➢ If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type
    - insertion order is not preserved
    - duplicates are not allowed
    - heterogeneous objects are allowed
    - index concept is not applicable
    - It is mutable collection
    - Growable in nature, based on our requirement we can increase or decrease the size

Eg:

```
>>> s={1,2,"raj",True,1,2}
>>> s
{1, 2, 'raj'}
>>> s.remove(2)
>>> s
{1, 'raj'}
>>> s.add(10)
>>> s
{1, 10, 'raj'}
>>> s.add("MREC")
>>> s
{1, 10, 'raj', 'MREC'}
```

10.dict Data Type:

- If we want to represent a group of values as key-value pairs then we should go for dict data type.
- Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

Eg:

```
>>> d={1:"one",2:"Two",3:"Three"}
>>> d[1]
'one'
>>> d
{1: 'one', 2: 'Two', 3: 'Three'}
>>> d[4]="Four"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four'}
>>> d[5]="error"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'error'}
>>> d[5]="Five"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five'}
```

11. None Datatype:

- The None Datatype is used to define the null value or no value, the none value means not 0, or False value, and it is a data it's own
- None keyword is an object and is a data type of nonetype class
- None datatype doesn't contain any value.
- None keyword is used to define a null variable or object.
- None keyword is immutable.

Eg:

Assume a=10, that means a is the reference variable pointing to 10 and if I take a=none then a is not looking to the object 10

```
>>> a=10
>>> type(a)
<class 'int'>
>>> a=None
>>> type(a)
<class 'NoneType'>
```

Python Programming

MODULE - II

Agenda:

- Modules: Modules and Files
- Namespaces
- Importing Modules,
- Importing Module Attributes,
- Module Built-in Functions,
- Packages,
- Other Features of Modules
- Files: File Objects,
- File Built-in Function,
- File Built-in Methods,
- File Built-in Attributes,
- Standard Files,
- Command-line Arguments,
- File System,
- File Execution,
- Persistent Storage Modules.
- Exceptions: Exceptions in Python,
- Detecting and Handling Exceptions,
- Context Management,
- Exceptions as Strings,
- Raising Exceptions,
- Assertions,
- Standard Exceptions,
- Creating Exceptions,
- Why Exceptions,
- Why Exceptions at All?
- Exceptions and the sys Module.

Modules

- Like many other programming languages, Python supports modularity. That is, you can break large code into smaller and more manageable pieces. And through modularity, Python supports code reuse.
- We can import modules in Python into your programs and reuse the code therein as many times as you want.
- Modules provide us with a way to share reusable functions.

A module is simply a "Python file" which contains code we can reuse in multiple Python programs. A module may contain functions, classes, lists, etc.

- Modules in Python can be of two types:
    1. Built-in Modules.
    2. User-defined Modules.

1. Built in Modules in Python

- One of the many superpowers of Python is that it comes with a "rich standard library". This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.
- In Python, modules are accessed by using the import statement
- When our current file is needed to use the code which is already existed in other files then we can import that file (module).
- When Python imports a module called module1 for example, the interpreter will first search for a built-in module called module1. If a built-in module is not found, the Python interpreter will then search for a file named module1.py in a list of directories that it receives from the sys.path variable.
- We can import module in three different ways:
    1. *import <module_name>*
    2. *from <module_name> import <method_name>*
    3. *from <module_name> import **

1.import <module_name>:

- This way of importing module will import all methods which are in that specified module.

  Eg: import math
- Here this import statement will import all methods which are available in math module. We may use all methods or may use required methods as per business requirement.

2.From <module_name> import <method_name>:

- This import statement will import a particular method from that module which is specified in the import statement.
- We can't use other methods which are available in that module as we specified particular method name in the import statement.
- The main advantage of this is we can access members directly without using module name.

Eg:from <module_name>import <*>
from math import factorial
from math import*

Finding members of module by using dir() function:

- Python provides inbuilt function dir() to list out all members of current module or a

Specified module.

- **dir()** ===>To list out all members of current module
- **dir(moduleName)**==>To list out all members of specified module

1.Eg:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

2.Eg:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Some of Standard modules

- **Math module**
- **Calendar module**

Working with math module:

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

8.factorial(x)

….

Eg:

```
>>> from math import*
>>> print(sqrt(5))
2.23606797749979
>>> print(ceil(15.25))
16
>>> print(floor(15.25))
15
>>> print(fabs(-15.6))
15.6
>>> print(fabs(15.6))
15.6
>>> print(log(10.5))
2.3513752571634776
>>> print(sin(1))
0.8414709848078965
>>> print(tan(0))
0.0
>>> print(factorial(5))
120
```

Working with Calendar module:

- Python defines an inbuilt module calendar which handles operations related to calendar.

- Calendar module allows output calendars like the program and provides additional useful functions related to the calendar.

calendar.day_name: An array that represents the days of the week in the current locale.

1. Displaying all week names one by one

```
import calendar
for i in calendar.day_name:
 print(i)

output:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

calendar.month_name:

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

```
>>> import calendar
>>> for i in calendar.month_name:
        print(i)
January
February
March
April
May
June
July
August
September
October
November
December
```

calendar.monthrange(year, month): Returns weekday of first day of the month and number of days in month, for the specified year and month.

```
>>> import calendar
>>> print(calendar.monthrange(2021,6))
```

```
(1, 30)
>>> print(calendar.monthrange(2021,7))
(3, 31)
>>> print(calendar.monthrange(2022,1))
(5, 31)
>>> print(calendar.monthrange(2021,1))
(4, 31)
```

calendar.isleap(year): Returns True if year is a leap year, otherwise False.

```
>>> import calendar
>>> print(calendar.isleap(2020))
True
>>> print(calendar.isleap(2021))
False
```

calendar.leapdays(y1, y2): Returns the number of leap years in the range from y1 to y2 (exclusive), where y1 and y2 are years.

```
>>> import calendar
>>> print(calendar.leapdays(2000,2020))
5
```

calendar.weekday(year, month, day): Returns the day of the week (0 is Monday) for year (1970–…), month (1–12), day (1–31)

```
>>> import calendar
>>> print(calendar.weekday(2020,5,1))
4
>>> print(calendar.weekday(2021,5,1))
5
```

calendar.weekheader(n): Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday

```
>>> import calendar
>>> print(calendar.weekheader(1))
M T W T F S S
>>> print(calendar.weekheader(3))
Mon Tue Wed Thu Fri Sat Sun
>>> print(calendar.weekheader(10))
  Monday    Tuesday   Wednesday   Thursday    Friday    Saturday    Sunday
```

calendar. calendar(year, w, l, c): Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

This function shows the year, width of characters, no. of lines per week and column separations.

```
>>> import calendar
>>> print(calendar.calendar(2021))
```

Output:prints 2021 full calendar

1. *User defined Modules.*

- Another superpower of Python is that it lets you take things in your own hands.
- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.
- Modules in Python provides us the flexibility to organize the code in a logical way.
- To use the functionality of one module into another, we must have to import the specific module.

Creating a Module:

Shown below is a Python script containing the definition of sum() function. It is saved as calc.py.

```
#calc.py
def sum(x, y):
   return x + y
def sub(x, y):
   return x - y
def mul(x, y):
   return x * y
def di(x, y):
   return x / y
```

Importing a Module

We can now import this module and execute the any functions which are there in calac.py module in the Python shell.

```
>>> import calc
>>> print(calc.sum(4,5))
9
>>> print(calc.sub(4,5))
-1
>>> print(calc.mul(4,5))
20
```

```
>>> from calc import *
>>> print(sum(4,5))
9
>>> print(sub(4,5))
-1
>>> print(mul(4,5))
20
```

- Every module, either built-in or custom made, is an object of a module class. Verify the type of different modules using the built-in type() function, as shown below.

```
>>> import calc
>>> type(calc)
<class 'module'>
>>> import math
>>> type(math)
<class 'module'>
```

Renaming the Imported Module

Use the as keyword to rename the imported module as shown below.-

```
>>> import calc as c
>>> import math as raj
>>> import calc as c
>>> import math as raj
>>> print(c.sum(4,5))
9
>>> print(raj.factorial(5))
120
```

Namespaces

- Generally speaking, a **namespace** is a naming system for making names unique to avoid ambiguity.
- Everybody knows a namespacing system from daily life, i.e. the naming of people in firstname and familiy name (surname).
- A namespace is a simple system to control the names in a program. It ensures that names are unique and won't lead to any conflict.
- Some namespaces in Python:
    1. Local Namespace
    2. Global Namespace

3. Built-in Namespace

Local Namespace:

The Variables which are defined in the function are a local scope of the variable. These variables are defined in the function body.

Global Namespace

The Variable which can be read from anywhere in the program is known as a global scope. These variables can be accessed inside and outside the function. When we want to use the same variable in the rest of the program, we declare it as global.

Eg:

```
n=0#global namesapce
def f1():
   n=1#local namespace
   print("local variable n=",n)
f1()
print("Global variable n=",n)
```

OutPut:

```
local variable n= 1
Global variable n= 0
```

Built-in Scope

- If a Variable is not defined in local,or global scope, then python looks for it in the built-in scope.
- In the Following Example, 1 from math module pi is imported, and the value of pi is not defined in global, local and enclosed.
- Python then looks for the pi value in the built-in scope and prints the value. Hence the name which is already present in the built-in scope should not be used as an identifier.

Eg:

```
# Built-in Scope
from math import pi
# pi = 'Not defined in global pi'
def f1():
   print('Not defined in f1() pi')
def f2():
   print('Not defined in f2() pi')
f1()
f2()
print('pi is Built-in scope',pi)
```

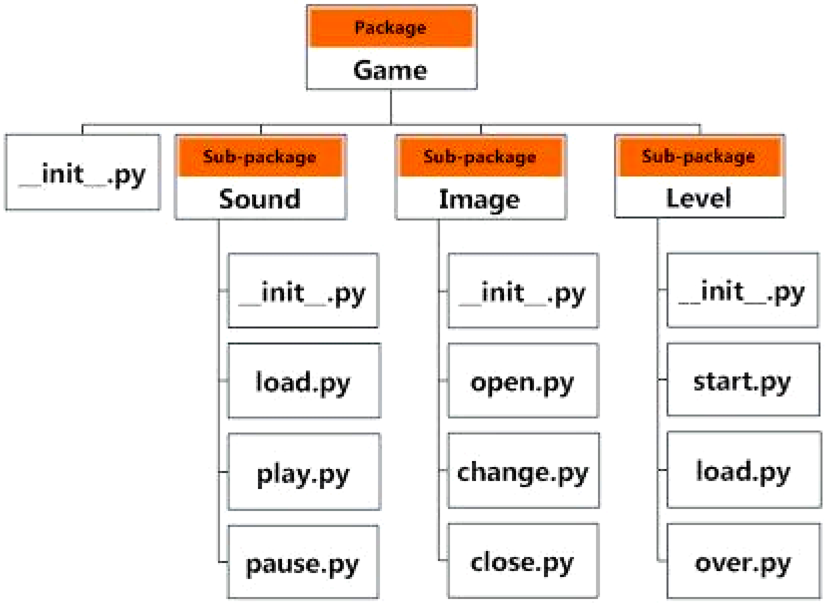
OutPut:

```
Not defined in f1() pi
Not defined in f2() pi
pi is Built-in scope 3.141592653589793
```

Packages in Python

- A Package is nothing but a collection of modules. It is also imported into programs.
- In Package, several modules are present, which you can import in your code.
- Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access.
- Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.
- Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named **__init__.py** in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.
- Any folder or directory contains **__init__.py** file,is considered as a Python package.This file can be empty.
- As we discussed, a package may hold other Python packages and modules. But what distinguishes a package from a regular directory? Well, a Python package must have an **__init__.py** file in the directory.
- You may leave it empty, or you may store initialization code in it. But if your directory does not have an **__init__.py** file, it isn't a package; it is just a directory with a bunch of Python scripts. Leaving **__init__.py** empty is indeed good practice.

Example: Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



The Following Steps to be follow.

Step1: Create a folder or package

Step2: Inside the Folder create a sub folder or package

Step3: Inside the package we have to create _init_.py which indicate its a package

Step4: After that we can create some modules based on requirement

Step5: After that we have to create main module in package folder by importing the created modules in sub package.

Eg 1:

```
F:\>
 |-test.py
 |-python_package
 |-First.py
 |-Second.py
 |-__init__.py
```

test.py

```
from python_package import First,second
First.f1()
second.f2()
```

First.py

```
def f1():
   print("This is First function")
```

Second.py

```
---------------
                    def f2():
                       print("This is Second Function")
```

OutPut:

```
                    This is First function
                    This is Second Function
```

Files Handling in Python

- Python File Handling Before we move into the topic “Python File Handling”, let us try to understand why we need files?
- So far, we have been receiving the input data from the console and writing the output data back to the console.
- The console only displays a limited amount of data. Hence we don’t have any issues if the input or output is small. What if the output or input is too large?
- We use files when we have large data as input or output.
- A file is nothing but a named location on disk which stores data.
- Files are also used to store data permanently since it stores data on non-volatile memory.
- Most modern file systems are composed of three main parts:
    1. **Header:** metadata about the contents of the file (file name, size, type, and so on)
    2. **Data:** contents of the file as written by the creator or editor
    3. **End of file (EOF):** special character that indicates the end of the file

Types of Files in Python

- Text File
- Binary File

1. Text File

- Text file store the data in the form of characters.
- Text file are used to store characters or strings.
- Usually we can use text files to store character data

  eg: abc.txt

2. Binary File

- Binary file store entire data in the form of bytes.
- Binary file can be used to store text, image, audio and video.

- Usually we can use binary files to store binary data like images,video files, audio files etc.

File operation on Text Files:

In Python, we can perform the following file operations:

- Open a file
- Read or write a file
- Close a file

Opening a File:

- Before performing any operations like read or write on a file, the first thing we need to do is open a file.
- Python provides an in-built function open() to open a file.
- The open function accepts two parameters: the name of the file and the access mode.
- The access mode specifies what operation we are going to perform on a file whether it is read or write.
- The open() function in turn returns a file object/handle, with which we can perform file operations based on the access mode.

Syntax: file_object=open(filename, access_mode)

- The allowed modes in Python are

- **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.

- **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 available then this mode will create that file.

- **a :** open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

- **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.

- **w+ :** To write and read data. It will override existing data.

- **a+ :** To append and read data from the file.It wont override existing data.

- **x :** To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

❖ All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

- **rb,wb,ab,r+b,w+b,a+b,xb**

Ex:

1. **file_object=open("test.txt") # when file is in the current directory**
2. **file_object=open("C:/User/Desktop/test.txt") # specify full path when file is in different directory**

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()

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)

Eg:1

```
1) f=open("abcd.txt",'w')
2) f.write("MREC \n")
3) f.write("CSE \n")
4) f.write("DS\n")
5)f.write("Dept\")
6) f.close()
abcd.txt:
MREC
CSE
DS
DEPT
```

Eg 2:

```
1) f=open("abcd.txt",'a')
2) list=["\nAI&ML\n","IOT\n","RAJ"]
3) f.writelines(list)
4) f.close()

abcd.txt:
MREC
CSE
DS
DEPT
AI&ML
IOT
RAJ
```

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()
```

Output

```
MREC
CSE
DS
DEPT
AI&ML
IOT
RAJ
```

Eg 2: To read only first 10 characters:

```
f=open("abc.txt",'r')
data=f.read(10)
print(data)
f.close()
```

Output

```
MREC
CSE
DS
```

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()
```

Output

```
MREC
 CSE
 DS
```

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()
```

Output

```
MREC
CSE
DS
DEPTS
AI&ML
IOT
RAJ
```

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.
- The position(index) of first character in files is zero just like string index.

Eg:

```
f=open('F:/abcd.txt','r')
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(2))
f.close()
```

Output:

```
0
MR
2
EC
```

seek():

- We can use seek() method to move cursor(file pointer) to specified location.
- **Syntax:f.seek(offset)**

Eg:

```
f=open('F:/abcd.txt','r')
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(2))
f.seek(0)
print(f.read(4))
f.seek(4)
print(f.read())
f.close()
```

output:

```
0
MR
2
EC
MREC
```

File Built in Attributes and Built in Methods

- Once we opened a file and we got file object, we can get various details related to that file by using its properties or attributes and methods on it.
- The Following are some of the attributes.
- **name -->** Name of opened file
- **mode -->**Mode in which the file is opened
- **closed -->**Returns boolean value indicates that file is closed or not

Eg:

```
>>>
f=open("C:/Users/rajas/AppData/Local/Programs/Python/Python39/m2.py",'r')
>>> f.name
'C:/Users/rajas/AppData/Local/Programs/Python/Python39/m2.py'
>>> f.mode
'r'
>>> f.closed
False
```

File Built in Methods

Python has the following set of methods available for the file object.

Method	Description
close()	Closes the file
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

File close() Method

- Close a file after it has been opened:

```
f = open("raj.txt", "r")
print(f.read())
f.close()
```

File fileno() Method

- Return the file descriptor of the stream:

```
f = open("raj.txt", "r")
print(f.fileno())
```

File flush() Method

- The flush() method cleans out the internal buffer.
- You can clear the buffer when writing to a file:

```
f = open("myfile.txt", "a")
f.write("Now the file has one more line!")
f.flush()
f.write("...and another one!")
```

File isatty() Method

- The isatty() method returns True if the file stream is interactive, example: connected to a terminal device.

```
f = open("raj.txt", "r")
print(f.isatty())
```

File read() Method

- The read() method returns the specified number of bytes from the file. Default is -1 which means the whole file.

```
f = open("raj.txt", "r")
print(f.read())
```

File readable() Method

- The readable() method returns True if the file is readable, False if not.

```
f = open("raj.txt", "r")
print(f.readable())
```

File readline() Method

- The readline() method returns one line from the file.
- You can also specified how many bytes from the line to return, by using the size parameter.

```
f = open("demofile.txt", "r")
print(f.readline())
```

File readlines() Method

- The readlines() method returns a list containing each line in the file as a list item.

```
f = open("raj.txt", "r")
print(f.readlines())

f = open("raj.txt", "r")
print(f.readline())
```

File seek() Method

- The seek() method sets the current file position in a file stream.
- The seek() method also returns the new postion.

```
f = open("raj.txt", "r")
f.seek(4)
print(f.readline())
```

File seekable() Method

- The seekable() method returns True if the file is seekable, False if not.
- A file is seekable if it allows access to the file stream, like the seek() method.

```
f = open("raj.txt", "r")
print(f.seekable())
```

File tell() Method

- The tell() method returns the current file position in a file stream.

```
f = open("raj.txt", "r")
print(f.tell())
```

File truncate() Method

- The truncate() method resizes the file to the given number of bytes.
- If the size is not specified, the current position will be used.

```
f = open("demofile2.txt", "a")
f.truncate(20)
f.close()
```

```
#open and read the file after the truncate:
f = open("demofile2.txt", "r")
print(f.read())
```

File writable() Method

- The writable() method returns True if the file is writable, False if not.
- A file is writable if it is opened using "a" for append or "w" for write.

```
f = open("raj.txt", "a")
print(f.writable())
```

File write() Method

- The write() method writes a specified text to the file.
- Where the specified text will be inserted depends on the file mode and stream position.
- "a": The text will be inserted at the current file stream position, default at the end of the file.
- "w": The file will be emptied before the text will be inserted at the current file stream position, default 0.

```
f = open("demofile2.txt", "a")
f.write("See you soon!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

File writelines() Method

- The writelines() method writes the items of a list to the file.
- Where the texts will be inserted depends on the file mode and stream position.
- "a": The texts will be inserted at the current file stream position, default at the end of the file.
- "w": The file will be emptied before the texts will be inserted at the current file stream position, default 0.

```
f = open("raj.txt", "a")
f.writelines(["See you soon!", "Over and out."])
f.close()

#open and read the file after the appending:
f = open("raj.txt", "r")
print(f.read())
```

File operation on Binary Files:

- Binary file store entire data in the form of bytes.
- Binary file can be used to store text, image, audio and video.
- Usually we can use binary files to store binary data like images,video files, audio files etc.
- In Python, we can perform the following file operations:

- Open a file
- Read or write a file
- Close a file

Eg: program to Read an image and that to another.

```
f1=open('mrec.jpg','rb')
f2=open('mrec1.jpg','wb')
#bytes=f1.read()
f2.write(f1.read())
print("Image copied from f1 to f2:\n")
f1.close()
f2.close()
```

File System in python

- A file system is a process that manages how and where data on storage disk, typically a hard disk drive (HDD), is stored, accessed and managed. It is a logical disk component that manages a disk's internal operations as it relates to a computer and is abstract to a human user.
- A directory simply is a structured list of documents and folders. A directory can have sub-directories and files. When we have too many files, Python directory comes in handy in file management or system with directories and sub-directories.
- Python has os module with multiple methods defined inside for directory and file management or system

Working with Directories:

It is very common requirement to perform operations for directories like

To Know Current Working Directory:

```
import os
print("The cwd=",os.getcwd())
```

OutPut:

```
The cwd= C:\Users\rajas\AppData\Local\Programs\Python\Python39
```

To create a sub directory in the current working directory:

```
import os
```

```
os.mkdir('Raj')
print("The Directory Raj is Created”)
```

OutPut:

```
The Directory Raj is Created
```

To rename a directory in Python:

- Python has rename( ) function to rename a directory.

Syntax: os.rename(old_name,new_name)

```
import os
os.rename('Raj','mrec')
print("The Directory Raj Renamed to mrec")
```

OutPut:

```
The Directory Raj Renamed to mrec
```

To change directories in Python:

- In Python, chdir( ) function defined in module os is used to change the working directories.

Example: Suppose we want to change our working directory to Raj in F: Here is how it is done.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\rajas\\AppData\\Local\\Programs\\Python\\Python39'
>>> os.chdir('F:/')
>>> os.getcwd()
'F:\\'
>>> os.mkdir('Raj')
>>> os.getcwd()
'F:\\'
>>> os.chdir('Raj')
>>> os.getcwd()
'F:\\Raj'
```

To list directories in Python:

- Python has listdir( ) function in module os to list all the directories and files in a particular location.
- listdir( ) returns a list containing the names of the entries in the directory given by path. The list is in arbitrary order, and does not include the special entries '.' and '..' even if they are present in the directory.

Here is an example:

```
>>> import os
>>> os.chdir('F:/')
>>> os.listdir()
['$RECYCLE.BIN', 'abcd.txt', 'add.txt', 'Applicant Details-Cloud.doc', 'c.py', 'cal.csv',
'certifiates', 'copy.txt', 'cse1.txt', 'DCIM', 'Download', 'ds.py', 'ds1.py', 'ds2.txt',
'eee.txt', 'exp2.py', 'filedemo.c', 'filedemo.exe', 'filedemo.o', 'first.py', 'first.txt',
'fwdresearchmethodologynotes.zip', 'Game', 'hello.txt', 'JAVA PROGRAMMING',
'm.c', 'm.exe', 'm.o', 'Machine Learning', 'MarriagePhotos', 'merge.c', 'merge.exe',
'merge.o', 'Meterials', 'Microsoft Office Enterprise 2010 Corporate Final (full
activated)', 'ML', 'myfile.txt', 'myfile1.txt', 'new.csv', 'new.py', 'new.txt', 'old',
'package', 'Packages', 'python', 'r.py', 'R20-python', 'Raj', 'raj.bin', 'raj.txt']
```

To remove a directory:

- To remove or delete a directory path in Python, rmdir( ) is used which is defined in os module.
- rmdir( ) works only when the directory we want to delete is empty, else it raises an OS error.
- So here are the ways to remove or delete empty and non-empty directory paths.

```
>> import os
>>> os.chdir('F:/Raj')
>>> os.mkdir('cse')
>>> os.listdir()
['cse']
>>> os.rmdir('cse')
```

To remove multiple directories in the path:

```
>>> import os
>>> os.chdir('F:/')
>>> os.removedirs('Raj/A')
```

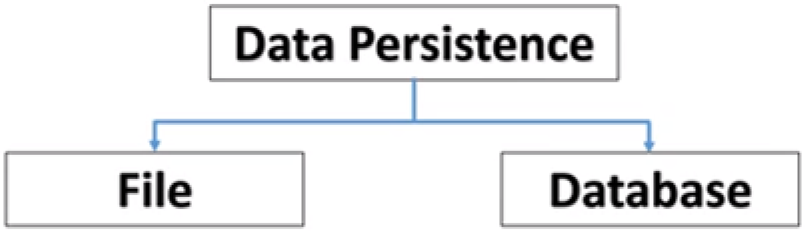
Check if Given Path is File or Directory

- To check if the path you have is a file or directory, import os module and use isfile() method to check if it is a file, and isdir() method to check if it is a directory.

```
>>> import os
>>> os.chdir('F:/')
>>> os.listdir()
['$RECYCLE.BIN', 'abcd.txt', 'add.txt', 'Applicant Details-Cloud.doc', 'c.py',
'cal.csv', 'exp2.py', 'filedemo.c', 'filedemo.exe', 'filedemo.o', 'first.py', 'first.txt',
'Raj']
>>> os.path.isfile('add.txt')
True
>>> os.path.isdir('Raj')
True
```

Persistent Storage Modules

- The word 'persistence' means "the continuance of an effect after its cause is removed".
- The term data persistence means it continues to exist even after the application has ended. Thus, data stored in a non-volatile storage medium such as, a disk file is persistent data storage.
- Data Persistence is the concept of storing data in a persistent form.
- It means that the data should be permanently stored on disk for further manipulation.
- There are two types of system used for data persistence they are



- There are two aspects to preserving data for long-term use: converting the data back and forth between the object in-memory and the storage format, and working with the storage of the converted data.
- The standard library includes a variety of modules that handle both aspects in different situations.

Serialization:

Serialization in Python is a mechanism of translating data structures or object state into a format that can be stored or transmitted and reconstructed later.

De-serialization:

The reverse operation of serialization is called de-serialization

- The type of manual conversion, of an object to string or byte format (and vice versa) is very cumbersome and tedious. It is possible to store the state of a Python object in the form of byte stream directly to a file, or memory stream and retrieve to its original state. This process is called **serialization** and **de-serialization**.
- Python's built in library contains various modules for serialization and de-serialization process. They are as follows.

S.No.	Name of the Module	Description
1	**pickle**	Python specific serialization library
2	**marshal**	Library used internally for serialization
3	**shelve**	Pythonic object persistence
4	**csv**	library for storage and retrieval of Python data to CSV format
5	**json**	Library for serialization to universal JSON format

Eg: Writing data to binary file without pickle module.

```
f=open('bin.bin','wb')
num=[10,20,30,40,50]
arr=bytearray(num)
f.write(arr)
f.close()
f=open('bin.bin','rb')
num=list(f.read())
print(num)
f.close()
```

OutPut:

```
[10,20,30,40,50]
```

- The problem with above program is the binary file requires bytes object only for that we have convert to bytes object only.
- To provide solution for this we have use any above modules

Pickle Module

- Pickling is the process whereby a python object is converted into byte stream.
- Unpickling is the reverse of this whereby a byte stream is converted back into an object.
- We can implement pickling and unpickling by using pickle module of Python.
- pickle module contains dump() function to perform pickling.
- **Syntax:pickle.dump(object,file)**
- pickle module contains load() function to perform unpickling
- **Syntax:obj=pickle.load(file)**

Eg:

```
import pickle
dict={1:"cse",2:"ds"}
f=open('bin.bin','wb')
pickle.dump(dict,f)
f.close()
f=open('bin.bin','rb')
s=pickle.load(f)
print(s)
f.close()
```

OutPut:

```
{1: 'cse', 2: 'ds'}
```

marshal Module

- The marshal module is used to serialize data—that is, convert data to and from character strings, so that they can be stored on file.

- The marshal module uses a simple self-describing data format. For each data item, the marshalled string contains a type code, followed by one or more type-specific fields. Integers are stored in little-endian order, strings are stored as length fields followed by the strings' contents (which can include null bytes), tuples are stored as length fields followed by the objects that make up each tuple, etc.
- Just as pickle module, marshal module also defined load() and dump() functions for reading and writing marshalled objects from / to file.

marshal.dump(value, file[, version]) :
This function is used to write the supported type value on the open writeable binary file. A ValueError exception is raised if the value has an unsupported type.

marshal.load(file) :
This function reads one value from the open readable binary file and returns it. EOF Error, ValueError or TypeError is raised if no value is read.

Example:

```
import marshal
dict={1:"cse",2:"ds"}
f=open('bin.bin','wb')
marshal.dump(dict,f)
f.close()
f=open('bin.bin','rb')
s=marshal.load(f)
print(s)
f.close()
```

OutPut:

```
{1: 'cse', 2: 'ds'}
```

Command-line Arguments

- There are many different ways in which a program can accept inputs from the user. The common way in Python Command-line Arguments is the input() method.
- Another way to pass input to the program is Command-line arguments. Almost every modern programming language support command line arguments.
- In a similar fashion, python does support command line arguments. It's a very important feature as it allows for dynamic inputs from the user.
- In a command-line argument, the input is given to the program through command prompt rather than python script like input() method.
- The Argument which are passing at the time of execution are called **Command Line Arguments.**
- Python supports different modules to handle command-line arguments. one of the popular one of them is **sys module.**

sys module:

- This is the basic and oldest method to handle command-line arguments in python. It has a quite similar approach as the C library argc/argv to access the arguments.
- sys module implements the command line arguments through list structure named sys.argv argv is the internal list structure which holds the arguments passed in command prompt
- argv is not Array it is a List. It is available sys Module.
- argv à list to handle dynamic inputs from the user
  - argv[0] à python filename
  - argv[1] àargument 1
  - argv[2] à argument 2
  - argv[3] à argument 3 and so on.
- Steps to create command line arguments program:
  1. Write a python program
  2. Save the python program as <program name>.py extension
  3. Open a command prompt and change the directory to the python program path
  4. Use the below command to execute the program
  5. py < python file.py > < arg1 > < arg2 > < arg3 >
  6. **Example:** py demo.py 10 20 30 40 50
- The first item in argv list i.e argv[0] is the python file name à in this case demo.py
- argv[1] is the first argument à 10
- argv[2] is the second argument à 20
- argv[3] is the third argument à 30 and so on
- By default, the type of argv is "String" so we have to typecast as per our requirement.

Example1:

```
import sys
print(type(sys.argv))
```

Output:

```
D:\>py c.py
<class 'list'>
```

Example2:

```
from sys import argv
print('The Number of Command Line Arguments:', len(argv))
print('The List of Command Line Arguments:', argv)
print('Command Line Arguments one by one:')
```

```
for x in argv:
    print(x)
```

OutPut:

```
D:\>py c.py Raj cse ds 10
The Number of Command Line Arguments: 5
The List of Command Line Arguments: ['c.py', 'Raj', 'cse', 'ds', '10']
Command Line Arguments one by one:
c.py
Raj
cse
ds
10
```

Example3:Add two values using command line

```
from sys import argv
a=int(argv[1])
b=int(argv[2])
sum=a+b
print("The Sum:",sum)
```

OutPut:

```
D:\>py c.py 1 2
The Sum: 3
```

Example2:Sum of elements

```
from sys import argv
sum=0
args=argv[1:]
for x in args :
    n=int(x)
    sum=sum+n
print("The Sum:",sum)
```

OutPut:

```
D:\>py c.py 1 2 3 4 5
The Sum: 15
```

Exception Handling in Python

Generally any programming language supports two types of errors,

1. Syntax errors
2. Runtime errors

Syntax errors:

- The errors which occur because of invalid syntax are called **syntax errors**.
- Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

Eg 1:

```
a=10
if a==10
 print("Raj")
SyntaxError: invalid syntax
```

Eg 2:

```
print "Raj"
SyntaxError: Missing parentheses in call to 'print'
```

Runtime errors:

- Runtime errors are also called exceptions.
- When the program is executing, if something goes wrong because of end user input or, programming logic or memory problems etc then we will call them runtime errors.

Exception:

An exception is nothing but an unwanted or unexpected block which disturbs the normal execution flow of program.

- An Exception is a run time error that happens during the execution of program.
- An exception is an error that happens during the execution of a program.
- Python raises an exception whenever it tries to execute invalid code.
- Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler.
- Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

Eg:

1. print(2/0) ==>ZeroDivisionError: division by zero
2. print(2/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

```
a=int(input("Enter Number:"))
 print(a)
  D:\>py test.py
2
 Enter Number:ten
 ValueError: invalid literal for int() with base 10: 'ten’
```

Types of Exceptions:

Exceptions are divided into two types they are,

1. System defined exceptions
2. User defined exceptions

System defined exceptions:

- These exceptions are defined by system so these are called **system defined or pre-defined exceptions.**
- Every exception in Python is an object. For every exception type the corresponding classes are available.
- Whevever 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
- Some of system defined exceptions are as follows,

S. No	**Name of the Built□in Exception**	**Explanation**
1	ZeroDivisionError	It is raised when the denominator in a division operation is zero
2	NameError	It is raised when a local or global variable name is not defined
3	IndexError	It is raised when the index or subscript in a sequence is out of range.
4	TypeError	It is raised when an operator is supplied with a value of incorrect data type.
5	ValueError	It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.
6	KeyError	**KeyError** exception is what is raised when you try to access a key that isn't in a dictionary (dict).
7	FileNotFoundError	The error **FileNotFoundError** occurs because you either don't know where a file actually is on your computer. Or, even if you do, you don't know how to tell your **Python** program where it is.
8	ModuleNotFoundError	A **ModuleNotFoundError** is raised when Python cannot successfully import a module.

1. **ZeroDivisionError:**

```
>>> a=10
>>> b=0
>>> print(a/b)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

2. NameError:

```
>>> print("a=",a)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print("a=",a)
NameError: name 'a' is not defined
```

3. IndexError:

```
>>> name="MREC"
>>> print(name[10])
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(name[10])
IndexError: string index out of range
```

4. ValueError:

```
>>> a=int(input("Enter a value:"))
Enter a value:Raj
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a=int(input("Enter a value:"))
ValueError: invalid literal for int() with base 10: 'Raj'
```

5. TypeError:

```
>>> a=10
>>> b="raj"
>>> print(a/b)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(a/b)
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

6. **KeyError:**

```
>>> D={1:'MREC',2:'CSE',3:'DS',4:'RAJ'}
>>> print(D[1])
MREC
>>> print(D[5])
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(D[5])
KeyError: 5
```

7. **FileNotFoundError:**

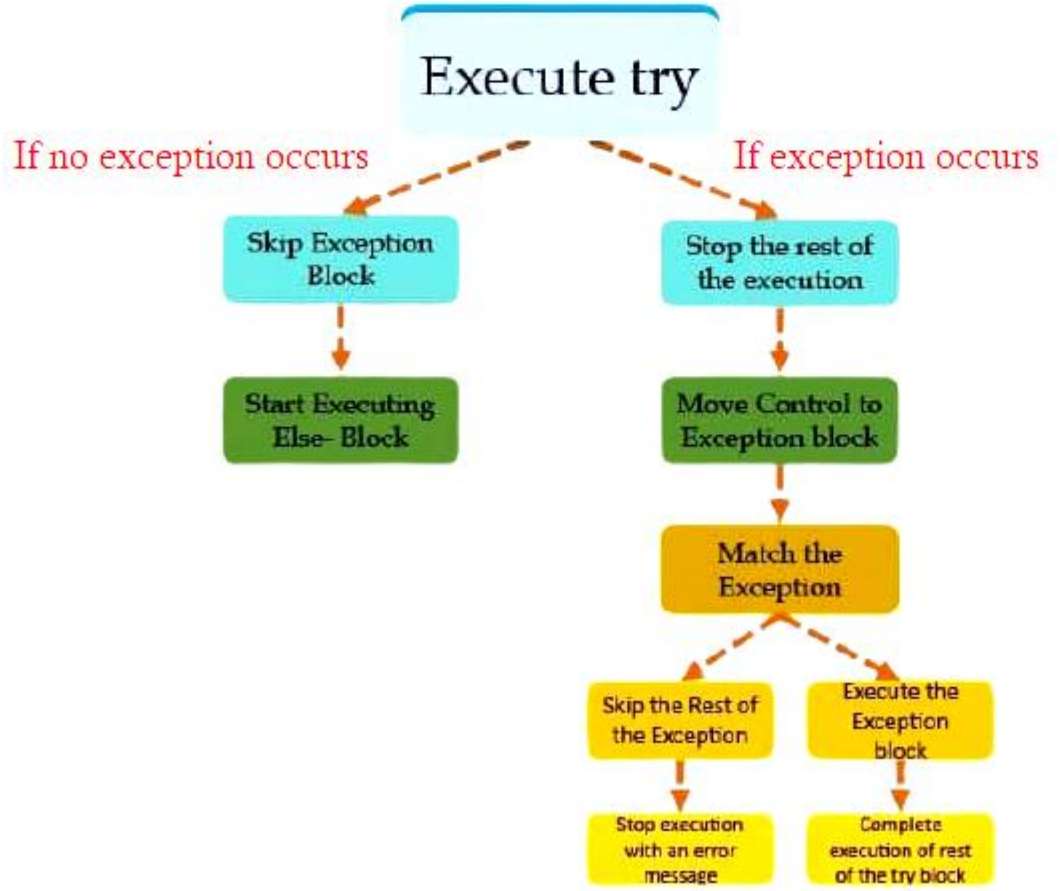
```
>>> f=open('Raj.txt','r')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    f=open('Raj.txt','r')
FileNotFoundError: [Errno 2] No such file or directory: 'Raj.txt'
```

8. **ModuleNotFoundError:**

```
>>> import cse_ds
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import cse_ds
ModuleNotFoundError: No module named 'cse_ds'
```

Detecting and Handling Exceptions or Exception Handing in Python

- Exception handling is a concept used in Python to handle the exceptions that occur during the execution of any program. Exceptions are unexpected errors that can occur during code execution.
- Exception handling does not mean repairing exception; we have to define an alternative way to continue rest of the program normally.
- It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program.
- Exception can be handled in two ways They are
  1. **Default Exception Handling**
  2. **Customized Exception Handling**
- The flowchart describes the exception handling process.


Execute try
If no exception occurs
If exception occurs
Skip Exception Block
Start Executing Else- Block
Stop the rest of the execution
Move Control to Exception block
Match the Exception
Skip the Rest of the Exception
Execute the Exception block
Stop execution with an error message
Complete execution of rest of the try block

Default Exception Handling

- 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. This entire process we call it as **Default Exception Handling**
- If an exception raised inside any method then the method is responsible to create Exception object with the following information.
  - ✓ Name of the exception.
  - ✓ Description of the exception.

- ✓ Location of the exception.

- After creating that Exception object the method handovers that object to the PVM.
- PVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then PVM terminates that method abnormally.
- PVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then PVM terminates that caller also abnormally
- Then PVM handovers the responsibility of exception handling to the default exception handler.
- Default exception handler just print exception information to the console in the following formats and terminates the program abnormally.
- Name of exception: description
- Location of exception

Example:

```
print("Start:")
print("Default Exception Handling:")
print(15/0)
print("No Exception Block:")
print("Stop")
```

OutPut:

```
Start:
Default Exception Handling:
Traceback (most recent call last):
  File "C:/Users/rajas/AppData/Local/Programs/Python/Python39/test.py", line 3, in <module>
    print(15/0)
ZeroDivisionError: division by zero
```

Customized Exception Handling

- It is highly recommended to handle exceptions.
- The Exceptions can be handled with the help of the following keywords or clauses in python.

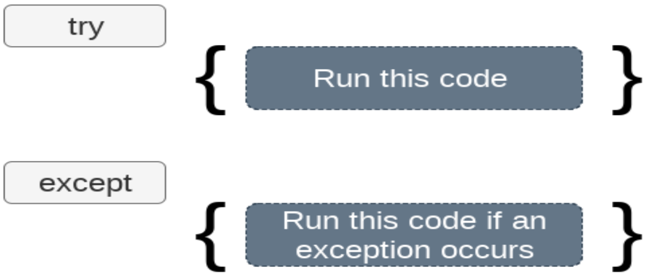
S. No	Name of the Exception Type Keyword	Explanation
1.	try	It will run the code block in which you expect an error to occur.
2.	except	Define the type of exception you expect in the try block

3.	else	If there no exception, then this block of code will be executed
4.	finally	Irrespective of whether there is an exception or not, this block of code will always be executed.
5.	raise	An exception can be raised forcefully by using the **raise** clause in Python.

- 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.
- We can handle the Exception with following ways.

1. The try-expect statement

- If the Python program contains suspicious or risky code that may throw the exception, we must place that code in the try block.
- The try block must be followed with the except statement, which contains a block of code that will be executed if there is some exception in the try block.
- 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.
- If any statement which is not part of try block raises an exception then it is always abnormal termination.



Syntax:

```
try :
   #statements in try block
except :
   #executed when error in try block
```

Example: Without Specific error type:

```
print("Start:")
print("Exception Handling without Specific Error Type:")
try:
   print(15/0)
except:
    print("Error occured")
print("Stop")
```

OutPut:

```
Start:
Exception Handling without Specific Error Type:
Error occured
Stop
```

Example: Catch Specific Error Type

```
print("Start:")
print("Exception Handling with Specific Error Type:")
try:
    print(15/0)
except ZeroDivisionError:
    print("we can't divide the value with zero")
print("Stop")
```

OutPut:

```
Start:
Exception Handling with Specific Error Type:
we can't divide the value with zero
Stop
```

try with multiple except blocks:

- The way of handling exception is varied from exception to exception. Hence for every exception type a separate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.
- As we know, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example:

```
print("Start:")
print("Exception Handling with Specific Error Type:")
try:
    print(15/0)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print("we can't divide the value with zero")
print("Stop")
```

OutPut:

```
Start:
Exception Handling with Specific Error Type:
we can't divide the value with zero
Stop
```

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.
- If try with multiple except blocks available then default except block should be last, otherwise we will get Syntax Error.

Syntax:

```
except:
    statements
```

Eg:

```
print("Start:")
print("Default except block:")
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")
print("Stop")
```

OutPut:

```
Start:
Default except block:
Enter First Number: 5
Enter Second Number: a
Default Except:Plz provide valid input only
Stop
```

except statement using with exception variable:

- We can use the exception variable with the except statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

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

```
  y=int(input("Enter Second Number: "))
  print(x/y)
except Exception as e:
  print("ZeroDivisionError:Can't divide with zero")
  print(e)
print("Stop")
```

OutPut:

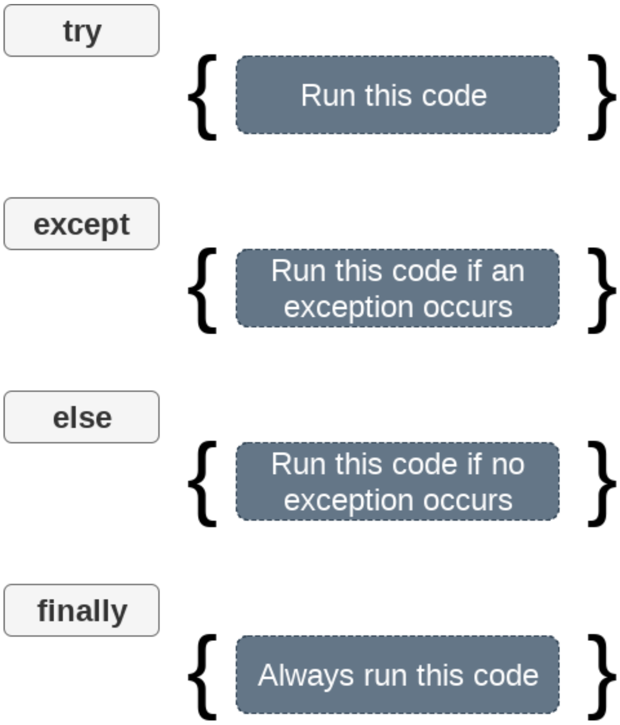
```
Start:
Enter First Number: 5
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero
division by zero
Stop
```

2.else and finally:

- In Python, keywords are else and finally can also be used along with the try and except clauses.
- In python, you can also use else clause on the **try-except block** which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Syntax:

```
try:
  #statements in try block
except:
  #executed when error in try block
else:
  #executed if try block is error-free
finally:
  #executed irrespective of exception occured or not
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

- The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code.
- If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.
- The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

```
print("Start:")
try:
   print('try block')
   x=int(input('Enter a number: '))
   y=int(input('Enter another number: '))
   z=x/y
except ZeroDivisionError:
   print("except ZeroDivisionError block")
   print("Division by 0 not accepted")
else:
   print("else block")
   print("Division = ", z)
finally:
   print("finally block")
```

```
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
print("Stop")
```

OutPut:

```
Start:
try block
Enter a number: 5
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.
Stop
```

3.Raise an Exception

- An exception can be raised forcefully by using the raise clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.
- **Syntax : raise Exception_class,<value>**
- To raise an exception, the raise statement is used. The exception class name follows it.An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type

Example 1:

```
print("Start:")
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed range")
print("Stop")
```

OutPut:6

```
Start:
Enter a number upto 100: 200
```

```
200 is out of allowed range
Stop
```

Example 2 Raise the exception with user defined message

```
print("Start:")
try:
   x=int(input('Enter a positive integer:'))
   if x <0:
      raise ValueError("You entered negative number")
except ValueError as e:
   print(e)
print("Stop")
```

Output:

```
Start:
Enter a positive integer:-2
You entered negative number
Stop
```

User defined 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.
- steps to create user defined exceptions

Step 1: Create User Defined Exception Class

- Write a new class for custom exception and inherit it from an in-build Exception class.
- Define function __init__() to initialize the object of the new class.
- You can add as many instance variables as you want, to support your exception. For simplicity, we are creating one instance variable called message.

```
class YourException(Exception):
  def __init__(self, message):
    self.message = message
```

You have created a simple user-defined exception class.

self :

- self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python.

__init__ :

- "__init__" is a reseved method in python classes. It is known as a constructor in object oriented concepts. This method called when an object is created from the class and it allow the class to initialize the attributes of a class.

Step 2: Raising Exception

- Now you can write a try-except block to catch the user-defined exception in Python.
- For testing, inside the try block we are raising exception using raise keyword.
- **raise YourException("Userdefined Exceptions")**
- It creates the instance of the exception class YourException. You can pass any message to your exception class instance.

Step 3: Catching Exception

- Now you have to catch the user-defined exception using except block.

```
except YourException as err:
  print(err.message)
```

- We are catching user defined exception called YourException.

Step 4: Write a Program for User-Defined Exception in Python

```
class ChildrenException(Exception):
    def __init__(self,arg):
        self.msg=arg
class YouthException(Exception):
    def __init__(self,arg):
        self.msg=arg
class AdultException(Exception):
    def __init__(self,arg):
        self.msg=arg
class SeniorException(Exception):
    def __init__(self,arg):
        self.msg=arg
age=int(input("Enter Age:"))
if (age<18) and (age>0):
    raise ChildrenException("The Person having the age between (0-18)!!!")
elif (age<25) and (age>=19):
    raise YouthException("The Person having the age between (19-24)!!!")
```

```
elif (age<65) and (age>=25):
    raise AdultException("The Person having the age between (25-64)!!!")
elif (age>=65):
    raise SeniorException("The Person having the age between (65 above)!!!")
else:
    print("You have entered invalid age!!!")
```

Output:

```
Enter Age:35
Traceback (most recent call last):
  File "C:/Users/rajas/AppData/Local/Programs/Python/Python39/user.py", line 19, in <module>
    raise AdultException("The Person having the age between (25-64)!!!")
AdultException: The Person having the age between (25-64)!!!-
```

Example2:

```
class PassException(Exception):
    def __init__(self,arg):
        self.msg=arg
class FailException(Exception):
    def __init__(self,arg):
        self.msg=arg
class MarksException(Exception):
    def __init__(self,arg):
        self.msg=arg

try:
    marks=int(input("Enter the marks of a subject:"))
    if(marks<35) and (marks>=0):
        raise FailException("Fail")
    elif(marks>=35):
        raise PassException("Pass")
    else:
        raise MarksException("Marks should be positive")
except FailException as e:
    print(e)
except PassException as e:
    print(e)
except MarksException as e:
    print(e)
print("Stop")
```

Output:

```
Enter the marks of a subject:-25
Marks should be positive
Stop
```

ASSERTIONS in python

- Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program.
- The process of identifying and fixing the bug is called debugging.
- Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug,compulsory we have to delete the extra added print() statments,otherwise these will be executed at runtime which creates performance problems and disturbs console output.
- To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.
- Hence the main purpose of assertions is to perform debugging. Usully we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of assert statements:

There are 2 types of assert statements

1. Simple Version
2. Augmented Version

1. Simple Version:

Syntax: **assert conditional_expression**

2. Augmented Version:

Syntax: **assert conditional_expression,message**

- conditional_expression will be evaluated and if it is true then the program will be continued.If it is false then the program will be terminated by raising AssertionError. By seeing AssertionError, programmer can analyze the code and can fix the problem.

Examples:1

```
assert True
```

```
print("Validation Passed")
```

Output:

```
Validation Passed
```

Examples:2

```
assert False
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D://assert1.py", line 1, in <module>    assert False
AssertionError
```

Examples:3

```
assert False ,"Validation Failed"
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D://assert1.py",, line 1, in <module>
    assert False ,"Validation Failed"
AssertionError: Validation Failed
```

Example: 4

```
assert "Python" in "Python Programming"
print("Validation Passed")
```

Output:

```
Validation Passed
```

Example:5

```
assert "Python" in "python Programming","Validation Failed"
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D: /assert1.py", line 1, in <module>
    assert "Python" in "python Programming","Validation Failed"
AssertionError: Validation Failed
```

Example:6

```
str1="Raj"
str2="Raj"
assert str1==str2,"Strings are not matched"
print("String are matched")
```

Output:

```
String are matched
```

Example:7

```
str1="Raj"
str2="Raj"
assert str1==str2,"Strings are not matched"
print("String are matched")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 3, in <module>
    assert str1!=str2,"Strings are not matched"
AssertionError: Strings are not matched
```

Example:8

```
assert "Raj" in ["MREC","CSE","DS","Raj"],"Validation Failed"
print("Validation passed")
```

Output:

```
Validation passed
```

Example:9

```
assert "raj" in ["MREC","CSE","DS","Raj"],"Validation Failed"
print("Validation passed")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 1, in <module>
    assert "raj" in ["MREC","CSE","DS"],"Validation Failed"
AssertionError: Validation Failed
```

Example:10

```
import math
assert math.factorial(5)==120,"Validation Failed"
print("Validation passed")
```

Output:

```
Validation passed
```

Example:11

```
import math
assert math.factorial(5)!=120,"Validation Failed"
print("Validation passed")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 2, in <module>
    assert math.factorial(5)!=120,"Validation Failed"
AssertionError: Validation Failed
```