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.

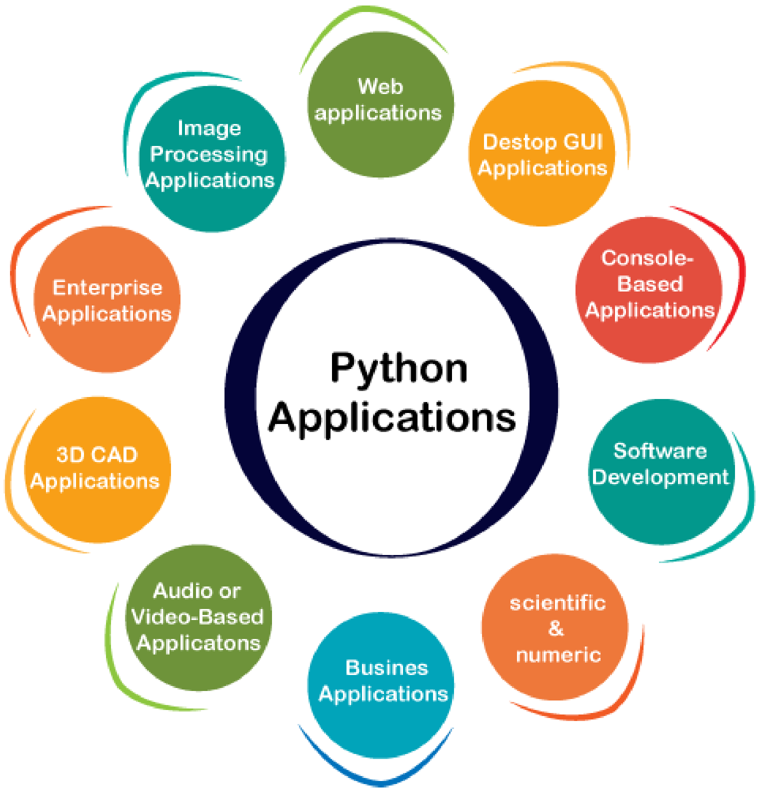
Versions of Python:

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

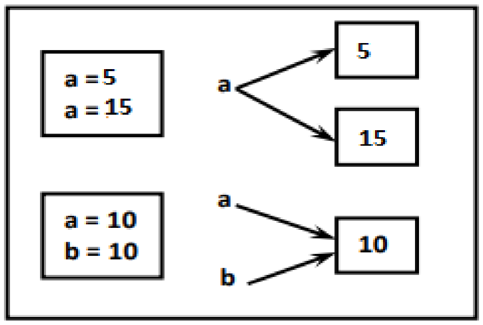
- **Fundamental or Build-in Data types or Data Structures**
- **Composite Data Types or Data Structures**

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:

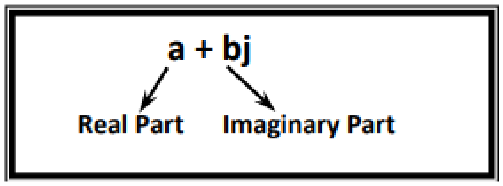
```
>>> f=0B11.01
File "<stdin>", line 1
f=0B11.01
SyntaxError: invalid syntax

>>> 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
```

```
9) >>> float("26.5")
10) 26.5
11) >>> float(26+5j)
12) TypeError: can't convert complex to float
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1011")
16) 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
```

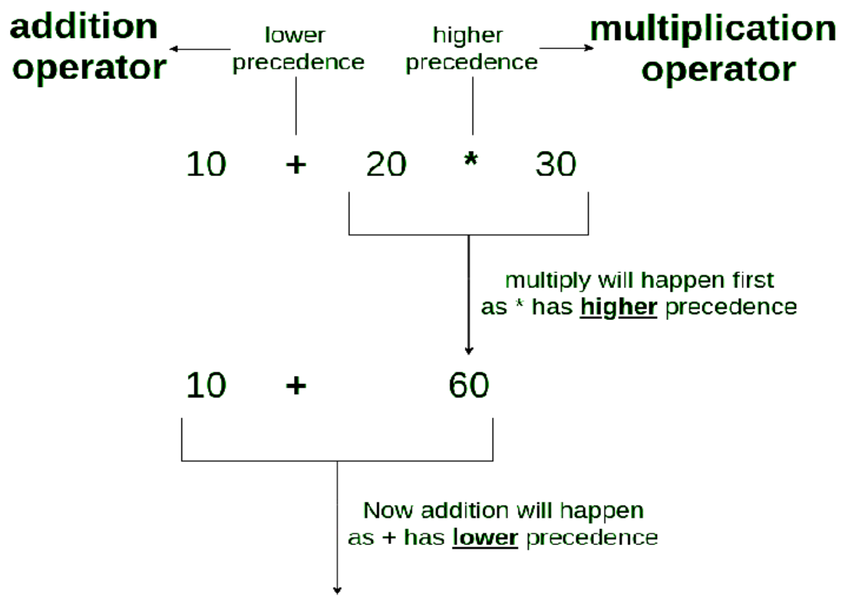
Precedence and Associativity of Operators in Python

- 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


addition operator
lower precedence
higher precedence
multiplication operator
10 + 20 * 30
multiply will happen first as * has higher precedence
10 + 60
Now addition will happen as + has lower precedence

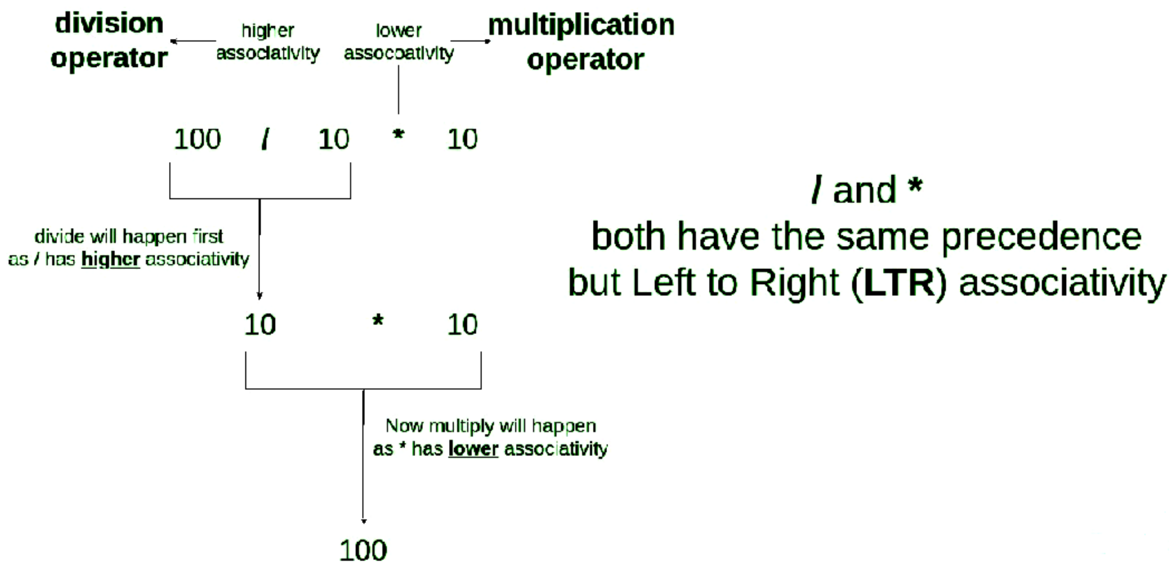
Example:

```
>>> 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 assocoatlvlty
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
```

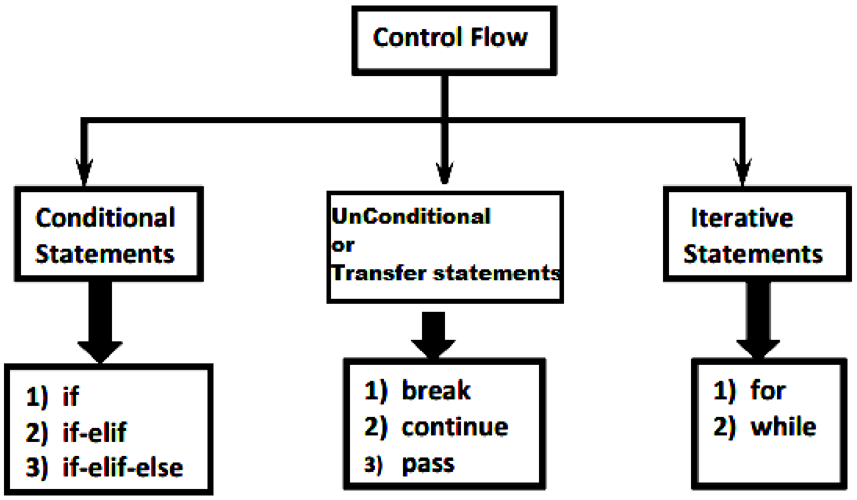
- Please see the following precedence and associativity table for reference. This table lists all operators from the highest precedence to the lowest precedence.

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'>
```