UNIT 1: INTRODUCTION TO PYTHON

Unit 1 Introduction to Python

What is Python?

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write a=10 to declare an integer value in a 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.

Python Features (Advantages)

There are a lot of features provided by python programming language.

1) Easy to Use:

Python is easy to use and high level language. Thus it is programmer-friendly language.

2) Expressive Language:
Python language is more expressive. The code is easily understandable.

3) Interpreted Language:
Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.

4) Cross-platform language:
Python can run equally on different platforms such as Windows, Linux, Unix, Macintosh etc. Thus, Python is a portable language.

5) Free and Open Source:
Python language is freely available(www.python.org).The source-code is also available. Therefore it is open source.

6) Object-Oriented language:
Python supports object oriented language. Concept of classes and objects comes into existence.

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

8) Large Standard Library:
Python has a large and broad library.

9) GUI Programming:
Graphical user interfaces can be developed using Python.

10) Integrated:
It can be easily integrated with languages like C, C++, JAVA etc.

Disadvantages of Python

- Python isn't the best for memory intensive tasks.
- Python is interpreted language & is slow compared to C/C++ or java.
- Python not a great choice for a high-graphic 3d game that takes up a lot of CPU.
- Python is evolving continuously, with constant evolution there is little substantial documentation available for the language.
- Python is not a very good language for mobile development.
- Python is not good for multi-processor/multi-core work.

Python Version

Mainly there are 3 versions:

- Python 3.0 - December 3, 2008
- Python 2.0 - October 16, 2000
- Python 1.0 - January 1994

Executing Python

There are three different ways to start Python −

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.
Enter “**py**” in the command line.Start coding right away in the interactive interpreter.

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application,like “py script.py”.

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

Python Basic Elements

Python Basic Syntax

Type the following text at the Python prompt and press the Enter:

```
>>> print("Hello, Python!")
```

output:

```
Hello, Python!
```

Python Variables

Variable is a name of the memory location where data is stored. Once a variable is stored that means a space is allocated in memory.

Assigning values to Variable:

We need not to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.
The assignment is done using the equal (=) operator.

Eg:

```
a=10
b=”hello”
print(a)
print(b)
```

Output:

```
10
hello
```

Multiple Assignment:

Multiple assignment can be done in Python at a time.
There are two ways to assign values in Python:

1. Assigning single value to multiple variables:

Eg:

```
x=y=z=50
print(x)
print(y)
print(z)
```

Output:

```
>>>
50
50
50
>>>
```

2.Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
print(a)
print(b)
print(c)
```

Output:

```
>>>
5
10
15
>>>
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- An identifier is a long sequence of characters and numbers.
- No special character except underscore (_) can be used as an identifier.
- Keyword should not be used as an identifier name.
- Python is case sensitive. So using case is significant.
- First character of an identifier can be character, underscore (_) but not digit.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

And	Exec	not
Assert	Finally	or
Break	For	pass
Class	From	print
Continue	Global	raise
Def	If	return
Del	Import	try
Elif	In	while
Else	Is	with
Except	Lambda	yield

Python - Variable Types

Standard Data Types

Python has five standard data types −

- Numbers
- String
- List
- Tuple
- Dictionary

1) Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. Numbers can be integer, long, float and complex. For example –

```
var1 = 1
var2 = 10.6
```

2) Python Strings

Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string .

Eg:

```
str = 'Hello World!'
print (str)          # prints complete string
```

Output:

```
Hello World!
```

3) Python Lists

A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

Eg:

```
    list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
    tinylist = [123, 'john']
    print (list)          # prints complete list
    print (list[0])       # prints first element of the list
    print(list[1:3])     # prints elements starting from 2nd till 3rd
    print(list[2:])      # print elements starting from 3rd element
    print(tinylist * 2)  # prints list two times
    print(list + tinylist) # prints concatenated lists
```

Output:

```
    ['abcd', 786, 2.23, 'john', 70.2]
    abcd
    [786, 2.23]
    [2.23, 'john', 70.2]
    [123, 'john', 123, 'john']
    ['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

4) Python Tuples

A tuple is another sequence data type that is similar to the list.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists.

Eg:

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
list = ( 'abcd', 786 , 2.23, 'john', 70.2  )
print(tuple )          # prints complete list
print(tuple[0])        # prints first element of the list
print(tinytuple * 2)   # prints list two times
print(tuple + tinytuple) # prints concatenated lists
tuple[2]=1000  #invalid syntax
list[2]=1000    #valid syntax
```

Output:

```
('abcd', 786, 2.23, 'john', 70.2)
Abcd
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

5) Python Dictionary

Python's dictionaries are kind of hash table type. They consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

Eg:

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one'])       # prints value for 'one' key
print(dict[2])           # prints value for 2 key
print(tinydict)          # prints complete dictionary
print(tinydict.keys())   # prints all the keys
print(tinydict.values()) # prints all the values
```

Output:

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Branching in python

Python If Statements

The if statement in python is same as c language which is used to test a condition. If condition is true, statement of if block is executed otherwise it is skipped.

Syntax of python if statement:

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

Example of if statement in python

```
a=10
if a==10:
      print ( "Hello User"  )
```

Output:

```
Hello User
```

Python If Else Statements

Syntax:

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

Example-

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

Output:

```
Year is Leap
```

Nested If Else Statement:

When we need to check for multiple conditions to be true then we use elif Statement.
This statement is like executing a if statement inside a else statement.

Syntax:

```
If statement:
    Body
elif statement:
    Body
else:
    Body
```

Example:

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

Output:

```
All Conditions are false.
```

The elif Statement

The elif statement is similar to SWITCH case in other languages.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Example

```
var = 100
if var == 200:
   print("1 - Got a true expression value“)
   print (var)
elif var == 150:
   print( "2 - Got a true expression value“)
   print( (var)
elif var == 100:
   print( "3 - Got a true expression value“)
   print( (var)
else:
   print ("4 - Got a false expression value“)
   print (var)
print ("Good bye!“)
```

output:

```
            3 - Got a true expression value
100
```

Good bye

Python Strings

Strings are the simplest and easy to use in Python.
String pythons are immutable.
We can simply create Python String by enclosing a text in single as well as double quotes. Python treat both single and double quotes statements same.

Strings Operators

There are basically 3 types of Operators supported by String:

1. Basic Operators.
2. Membership Operators.
3. Relational Operators.

Basic Operators:

There are two types of basic operators in String. They are "+" and "*".

String Concatenation Operator :(+)

The concatenation operator (+) concatenate two Strings and forms a new String.

eg:

```
>>> "ratan" + "jaiswal"
```

Output:

```
'ratanjaiswal'
```

Expression	**Output**
'10' + '20'	'1020'
"s" + "007"	's007'
'abcd123' + 'xyz4'	'abcd123xyz4'

NOTE: Both the operands passed for concatenation must be of same type, else it will show an error.

Eg:

```
'abc' + 3
>>>
```

output:

```
Traceback (most recent call last):
  File "", line 1, in
    'abc' + 3
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

Replication Operator: (*)

Replication operator uses two parameter for operation. One is the integer value and the other one is the String.
The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.
Eg:

```
>>> 5*"Vimal"
```

Output:

```
'VimalVimalVimalVimalVimal'
```

Expression	**Output**
"soono"*2	'soonosoono'
3*'1'	'111'
'$'*5	'$$$$$'

NOTE: We can use Replication operator in any way i.e., int * string or string * int. Both the parameters passed cannot be of same type.

Membership Operators

Membership Operators are already discussed in the Operators section. Let see with context of String.

There are two types of Membership operators:
1) in:"in" operator return true if a character or the entire substring is present in the specified string, otherwise false.
2) not in:"not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

Eg:

```
>>> str1="javatpoint"
>>> str2='sssit'
>>> str3="seomount"
>>> str4='java'
>>> st5="it"
>>> str6="seo"
>>> str4 in str1
True
>>> str5 in str2
>>> st5 in str2
True
>>> str6 in str3
True
```

```
>>> str4 not in str1
False
>>> str1 not in str4
True
```

Relational Operators:

All the comparison operators i.e., (<,><=,>=,==,!=,<>) are also applicable to strings. The Strings are compared based on the ASCII value or Unicode(i.e., dictionary Order).

Eg:

```
>>> "RAJAT"=="RAJAT"
True
>>> "Z"=="z"
False
```

Explanation:

The ASCII value of a is 97, b is 98, c is 99 and so on. The ASCII value of A is 65,B is 66,C is 67 and so on. The comparison between strings are done on the basis on ASCII value.

Slice Notation:

String slice can be defined as substring which is the part of string.
startIndex in String slice is inclusive whereas endIndex is exclusive.

Syntax:

```
<string_name>[startIndex:endIndex],
<string_name>[:endIndex],
<string_name>[startIndex:]
```

Example:

```
>>> str="Nikhil"
>>> str[0:6]
'Nikhil'
>>> str[0:3]
'Nik'
>>> str[2:5]
'khi'
>>> str[:6]
'Nikhil'
>>> str[3:]
'hil'
```

String Functions and Methods:

There are many predefined or built in functions in String. They are as follows:

capitalize()	It capitalizes the first character of the String.
count(string,begin,end)	Counts number of times substring occurs in a String between begin and end index.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
isalnum()	It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.
isalpha()	It returns True when all the characters are alphabets and there is at least one character, otherwise False.
isdigit()	It returns True if all the characters are digit and there is at least one character, otherwise False.
islower()	It returns True if the characters of a string are in lower case, otherwise False.
isupper()	It returns False if characters of a string are in Upper case, otherwise False.
isspace()	It returns True if the characters of a string are whitespace, otherwise false.
len(string)	len() returns the length of a string.
lower()	Converts all the characters of a string to Lower case.
upper()	Converts all the characters of a string to Upper Case.
swapcase()	Inverts case of all characters in a string.
lstrip()	Remove all leading whitespace of a string. It can also be used to remove particular character from leading.
rstrip()	Remove all trailing whitespace of a string. It can also be used to remove particular character from trailing.

Examples:

1) capitalize()
It capitalizes the first character of the String.

```
>>> 'abc'.capitalize()
```

Output:

```
'Abc'
```

2) count(string)
Counts number of times substring occurs in a String between begin and end index.

```
msg = "welcome to sssit"
substr1 = "o"
print  (msg.count(substr1, 4, 16))
substr2 = "t"
print  (msg.count(substr2))
```

Output:

```
>>>
2
2
>>>
```

3) find(string)
It returns the index value of the string where substring is found between begin index and end index.

```
str="Welcome to SSSIT"
substr1="come"
substr2="to"
print (str.find(substr1))
print (str.find(substr2))
print (str.find(substr1,3,10))
print (str.find(substr2,19))
```

Output:

```
>>>
3
8
3
-1
>>>
```

4) isalnum()
It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.

```
str="Welcome to sssit"
   print (str.isalnum())
str1="Python47"
print (str1.isalnum())
```

Output:

```
>>>
False
True
>>>
```

5) isalpha()

It returns True when all the characters are alphabets and there is at least one character, otherwise False.

```
string1="HelloPython"     # Even space is not allowed
print (string1.isalpha())
string2="This is Python2.7.4"
print (string2.isalpha())
```

Output:

```
>>>
True
False
>>>
```

6) isdigit()

It returns True if all the characters are digit and there is at least one character, otherwise False.

```
string1="HelloPython"
print (string1.isdigit())
string2="98564738"
print (string2.isdigit())
```

Output:

```
>>>
False
True
>>>
```

7) islower()

It returns True if the characters of a string are in lower case, otherwise False.

```
string1="Hello Python"
print (string1.islower())
string2="welcome to "
print (string2.islower())
```

Output:

```
>>>
False
True
>>>
```

8) isupper()
It returns False if characters of a string are in Upper case, otherwise False.

```
string1="Hello Python"
print (string1.isupper())
string2="WELCOME TO"
print (string2.isupper())
```

Output:

```
>>>
False
True
>>>
```

9) isspace()
It returns True if the characters of a string are whitespace, otherwise false.

```
string1="    "
print (string1.isspace())
string2="WELCOME TO WORLD OF PYT"
print (string2.isspace())
```

Output:

```
>>>
True
False
>>>
```

10) len(string)
len() returns the length of a string.

```
string1="    "
print (len(string1))
string2="WELCOME TO SSSIT"
print (len(string2))
```

Output:

```
>>>
4
16
>>>
```

11) lower()
Converts all the characters of a string to Lower case.

```
string1="Hello Python"
print (string1.lower())
string2="WELCOME TO SSSIT"
print (string2.lower())
```

Output:

```
>>>
hello python
welcome to sssit
>>>
```

12) upper()
Converts all the characters of a string to Upper Case.

```
string1="Hello Python"
print (string1.upper())
string2="welcome to SSSIT"
print (string2.upper())
```

Output:

```
>>>
HELLO PYTHON
WELCOME TO SSSIT
>>>
```

13) swapcase()
Inverts case of all characters in a string.

```
string1="Hello Python"
print (string1.swapcase())
string2="welcome to SSSIT"
print (string2.swapcase())
```

Output:

```
>>>
```

```
hELLO pYTHON
WELCOME TO sssit
>>>
```

14) lstrip()

Remove all leading whitespace of a string. It can also be used to remove particular character from leading.

```
string1="    Hello Python"
print (string1.lstrip())
string2="@@@@@@@@welcome to SSSIT"
print (string2.lstrip('@'))
```

Output:

```
>>>
Hello Python
welcome to world to SSSIT
>>>
```

15) rstrip()

Remove all trailing whitespace of a string. It can also be used to remove particular character from trailing.

```
string1="    Hello Python     "
print (string1.rstrip())
string2="@welcome to SSSIT!!!"
print (string2.rstrip('!'))
```

Output:

```
>>>
            Hello Python
@welcome to SSSIT
>>>
```

Python Input And Output

Python can be used to read and write data. Also it supports reading and writing data to Files.

"print" statement:

"print" statement is used to print the output on the screen.
print statement is used to take string as input and place that string to standard output.
Whatever you want to display on output place that expression inside the inverted commas. The expression whose value is to printed place it without inverted commas.
Syntax:

print ("expression") **or** **print** (expression)
eg:

```
a=10
print "Welcome to the world of Python"
print (a)
```

Output:

```
>>>
Welcome to the world of Python
10
>>>
```

Input from Keyboard:

Python offers two in-built functions for taking input from user. They are:

1) input()
2) raw_input()

1) input() function input() function is used to take input from the user. Whatever expression is given by the user, it is evaluated and result is returned back.
Syntax:

```
input("Expression")
```

eg:

```
n=input("Enter your expression ")
print ("The evaluated expression is ", n)
```

Output:

```
>>>
Enter your expression 10*2
The evaluated expression is  20
>>>
```

2) raw_input()raw_input() function is used to take input from the user. It takes the input from the Standard input in the form of a string and reads the data from a line at once.

Syntax:

```
raw_input(?statement?)
```

eg:

```
n=raw_input("Enter your name ")
print ("Welcome ", n)
```

Output:

```
>>>
Enter your name Rajat
Welcome  Rajat
```

```
>>>
```

Note: raw_input() function returns a string. Hence in case an expression is to be evaluated, then it has to be type casted to its following data type. Some of the examples are given below:

Program to calculate Simple Interest.

```
prn=int(raw_input("Enter Principal"))
r=int(raw_input("Enter Rate"))
t=int(raw_input("Enter Time"))
si=(prn*r*t)/100
print "Simple Interest is ",si
```

Output:

```
>>>
Enter Principal1000
Enter Rate10
Enter Time2
Simple Interest is  200
>>>
```

Program to enter details of an user and print them.

```
name=raw_input("Enter your name ")
math=float(raw_input("Enter your marks in Math"))
physics=float(raw_input("Enter your marks in Physics"))
chemistry=float(raw_input("Enter your marks in Chemistry"))
rollno=int(raw_input("Enter your Roll no"))
print( "Welcome ",name)
print( "Your Roll no is ",rollno)
print( "Marks in Maths is ",math)
print( "Marks in Physics is ",physics)
print( "Marks in Chemistry is ",chemistry)
print( "Average marks is ",(math+physics+chemistry)/3)
```

Output:

```
>>>
Enter your name rajat
Enter your marks in Math76.8
Enter your marks in Physics71.4
Enter your marks in Chemistry88.4
Enter your Roll no0987645672
Welcome  rajat
```

```
Your Roll no is  987645672
Marks in Maths is  76.8
Marks in Physics is  71.4
Marks in Chemistry is  88.4
Average marks is  78.8666666667
>>>
```

Python Iterations(Loops)

1)Python while Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Syntax

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

Example:

```
count = 0
while (count < 5):
   print('The count is:‘), count
   count = count + 1
print("Good bye!“)
```

Output:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
Good bye!
```

2)Python for Loop Statements

Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

Syntax

```
for iterating_var in sequence:
   statements(s)
```

Example

```
for letter in 'Python':     # First Example
   print( 'Current Letter :‘), letter
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print( 'Current fruit :‘), fruit
print( "Good bye!“)
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

The range() Function

The range() function is used to loop through a set of code a specified number of times. It returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

```
for x in range(5):
  print(x)
```

Output:

```
0
1
2
3
4
```

It is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example

```
for x in range(2, 5):
  print(x)
```

Output:

```
2
3
4
```

It is possible to specify the increment value by adding a third parameter: range(2, 10, 2):

Example

```
for x in range(2, 10, 2):
  print(x)
```

Output:

```
2
4
```

```
6
8
```

Break statement

break statement is a jump statement that is used to pass the control to the end of the loop.
When break statement is applied the control points to the line following the body of the loop , hence applying break statement makes the loop to terminate and controls goes to next line pointing after loop body.

eg:

```
for i in [1,2,3,4,5]:
    if i==4:
        print ("Element found“)
        break
    print (i)
```

Output:

```
>>>
1
2
3
Element found
>>>
```

Continue Statement

continue Statement is a jump statement that is used to skip the present iteration and forces next iteration of loop to take place. It can be used in while as well as for loop statements.

eg:

```
a=0
while a<=5:
    a=a+1
    if a%2==0:
        continue
    print (a)
print ("End of Loop")
```

Output:

```
>>>
1
3
5
End of Loop
>>>
```

Python Operators

Operators are particular symbols which operate on some values and produce an output.
The values are known as Operands.

Eg:
4 + 5 = 9
Here 4 and 5 are Operands and (+) , (=) signs are the operators. They produce the output 9.
Python supports the following operators:

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

Arithmetic Operators:

Operators	**Description**
//	Perform Floor division(gives integer value after division)
+	To perform addition
-	To perform subtraction
*	To perform multiplication
/	To perform division
%	To return remainder after division(Modulus)
**	Perform exponent(raise to power)

eg:

```
>>> 10+20
30
>>> 20-10
10
>>> 10*2
20
>>> 10/2
5
>>> 10%3
1
>>> 2**3
8
>>> 10//3
3
>>>
```

Relational Operators:

Operators	**Description**
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to

!=	Not equal to
<>	Not equal to(similar to !=)

eg:

```
>>> 10<20
True
>>> 10>20
False
>>> 10<=10
True
>>> 20>=15
True
>>> 5==6
False
>>> 5!=6
True
>>> 10<>2
True
>>>
```

Assignment Operators:

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

eg:

```
>>> c=10
>>> c
10
>>> c+=5
>>> c
15
>>> c-=5
>>> c
10
>>> c*=2
>>> c
20
>>> c/=2
>>> c
```

```
10
>>> c%=3
>>> c
1
>>> c=5
>>> c**=2
>>> c
25
>>> c//=2
>>> c
12
>>>
```

Logical Operators:

Operators	**Description**
And	Logical AND(When both conditions are true output will be true)
Or	Logical OR (If any one condition is true output will be true)
Not	Logical NOT(Compliment the condition i.e., reverse)

eg:

```
a=5>4 and 3>2
print a
b=5>4 or 3<2
print b
c=not(5>4)
print c
```

Output:

```
>>>
True
True
False
>>>
```

Membership Operators:

Operators	**Description**
In	Returns true if a variable is in sequence of another variable, else false.
not in	Returns true if a variable is not in sequence of another variable, else false.

eg:

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

Output:

```
>>>
a is in given list
b is given in list
>>>
```

Identity Operators:

Operators	Description
Is	Returns true if identity of two operands are same, else false
is not	Returns true if identity of two operands are not same, else false.

Example:

```
a=20
b=20
if( a is b):
    print  "a,b have same identity"
else:
    print "a, b are different"
b=10
if( a is not b):
    print  "a,b have different identity"
else:
    print "a,b have same identity"
```

Output:

```
>>>
a,b have same identity
a,b have different identity
>>>
```

Python List

1).Python lists are the data structure that is capable of holding different type of data.

2).Python lists are mutable i.e., Python will not create a new list if we modify an element in the list.

3).It is a container that holds other objects in a given order. Different operation like insertion and deletion can be performed on lists.

4).A list can be composed by storing a sequence of different type of values separated by commas.

5).A python list is enclosed between square([]) brackets.

6).The elements are stored in the index basis with starting index as 0.

eg:

```
        data1=[1,2,3,4]
        data2=['x','y','z']
        data3=[12.5,11.6]
        data4=['raman','rahul']
```

```
data5=[]
data6=['abhinav',10,56.4,'a']
```

Accessing Lists

A list can be created by putting the value inside the square bracket and separated by comma.

Syntax:

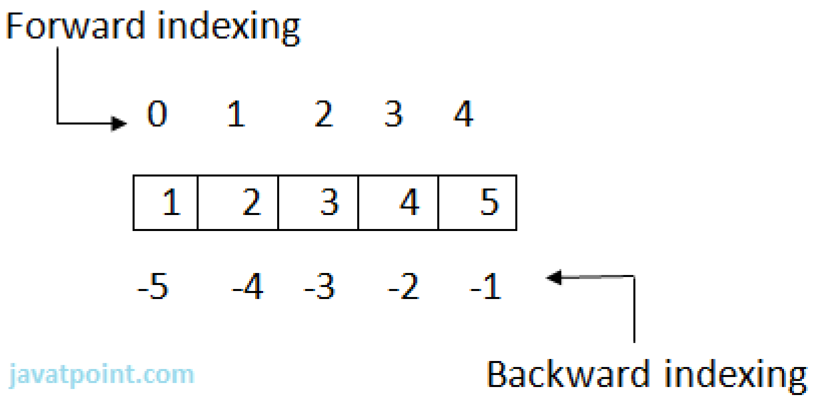
<list_name>=[value1,value2,value3,...,valuen]

For accessing list :

<list_name>[index]

Elements in a Lists:

Data=[1,2,3,4,5]



Data[0]=1=Data[-5] , Data[1]=2=Data[-4] , Data[2]=3=Data[-3] , =4=Data[-2] , Data[4]=5=Data[-1].

<u>**Different ways to access list:**</u>

Eg:

```
data1=[1,2,3,4]
data2=['x','y','z']
print( data1[0] )
print( data1[0:2] )
print( data2[-3:-1])
print( data1[0:]  )
print( data2[:2]  )
```

Output:

```
>>>
1
[1, 2]
['x', 'y']
[1, 2, 3, 4]
['x', 'y']
>>>
```

Note: Internal Memory Organization:
List do not store the elements directly at the index. In fact a reference is stored at each index which subsequently refers to the object stored somewhere in the memory. This is due to the fact that some objects may be large enough than other objects and hence they are stored at some other memory location.

List Operations:

Various Operations can be performed on List. Operations performed on List are given as:

a) Adding Lists:

Lists can be added by using the concatenation operator(+) to join two lists.

Eg:

```
list1=[10,20]
list2=[30,40]
list3=list1+list2
print( list3 )
```

Output:

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

Note: '+'operator implies that both the operands passed must be list else error will be shown.

Eg:

```
list1=[10,20]
list1+30
print( list1)
```

Output:

```
Traceback (most recent call last):
File "C:/Python27/lis.py", line 2, in <module>
list1+30
```

b) Replicating lists:

Replicating means repeating . It can be performed by using '*' operator by a specific number of time.

Eg:

```
list1=[10,20]
print( list1*2 )
```

Output:

```
[10, 20,10,20]
```

c) List slicing:

A subpart of a list can be retrieved on the basis of index. This subpart is known as list slice.

Eg:

```
list1=[1,2,4,5,7]
print( list1[0:2] )
print( list1[4]  )
list1[1]=9
```

```
        print( list1  )
```

Output:

```
>>>
[1, 2]
7
[1, 9, 4, 5, 7]
>>>
```

Note: If the index provided in the list slice is outside the list, then it raises an IndexError exception.

Other Operations:

Apart from above operations various other functions can also be performed on List such as Updating, Appending and Deleting elements from a List:

a) Updating elements in a List:

To update or change the value of particular index of a list, assign the value to that particular index of the List.

Syntax:

```
        <list_name>[index]=<value>
```

Eg:

```
        data1=[5,10,15,20,25]
        print( "Values of list are: "  )
        print( data1  )
        data1[2]="Multiple of 5"
        print( "Values of list are: "  )
        print( data1  )
```

Output:

```
>>>
        Values of list are:
        [5, 10, 15, 20, 25]
        Values of list are:
        [5, 10, 'Multiple of 5', 20, 25]
>>>
```

b) Appending elements to a List:

append() method is used to append i.e., add an element at the end of the existing elements.

Note : append() takes exactly one argument

Syntax:

```
<list_name>.append(item)
```

Eg:

```
        list1=[10,"rahul",'z']
        print( "Elements of List are: " )
        print( list1  )
        list1.append(10.45)
        print( "List after appending: "  )
        print( list1  )
```

Output:

```
>>>
        Elements of List are:
        [10, 'rahul', 'z']
        List after appending:
        [10, 'rahul', 'z', 10.45]
>>>
```

c) Deleting Elements from a List:

del statement can be used to delete an element from the list. It can also be used to delete all items from startIndex to endIndex.

Eg:

```
        list1=[10,'rahul',50.8,'a',20,30]
        print( list1 )
        del list1[0]
        print( list1  )
        del list1[0:3]
        print( list1  )
```

Output:

```
>>>
        [10, 'rahul', 50.8, 'a', 20, 30]
        ['rahul', 50.8, 'a', 20, 30]
        [20, 30]
>>>
```

Functions and Methods of Lists:

There are many Built-in functions and methods for Lists. They are as follows:

There are following List functions:

Function	**Description**
min(list)	Returns the minimum value from the list given.
max(list)	Returns the largest value from the given list.
len(list)	Returns number of elements in a list.
list(sequence)	Takes Tuple and converts them to lists.

1) min(list):

Eg:

```
        list1=[101,981,'abcd','xyz','m']
        list2=['aman','shekhar',100.45,98.2]
        print( "Minimum value in List1: ",min(list1) )
        print( "Minimum value in List2: ",min(list2)  )
```

Output:

```
>>>
        Minimum value in List1:  101
        Minimum value in List2:  98.2
>>>
```

2) max(list):

Eg:

```
        list1=[101,981,'abcd','xyz','m']
        list2=['aman','shekhar',100.45,98.2]
        print( "Maximum value in List : ",max(list1)  )
        print( "Maximum value in List : ",max(list2)  )
```

Output:

```
>>>
        Maximum value in List :  xyz
        Maximum value in List :  shekhar
>>>
```

3) len(list):

Eg:

```
        list1=[101,981,'abcd','xyz','m']
        list2=['aman','shekhar',100.45,98.2]
        print( "No. of elements in List1: ",len(list1))
        print( "No. of elements in List2: ",len(list2) )
```

Output:

```
>>>
        No. of elements in List1 :  5
        No. of elements in List2 :  4
>>>
```

4) list(sequence):

Eg:

```
seq=(145,"abcd",'a')
data=list(seq)
print( "List formed is : ",data  )
```

Output:

```
>>>
List formed is :  [145, 'abcd', 'a']
>>>
```

There are following built-in methods of List:

Methods	**Description**
index(object)	Returns the index value of the object.
count(object)	It returns the number of times an object is repeated in list.
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the popped object.
insert(index,object)	Insert an object at the given index.
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.
sort()	It is used to sort the elements of the List.

1) index(object):

Eg:

```
        data = [786,'abc','a',123.5]
        print( "Index of 123.5:", data.index(123.5) )
        print( "Index of a is", data.index('a') )
```

Output:

```
>>>
        Index of 123.5 : 3
        Index of a is 2
>>>
```

2) count(object):

Eg:

```
        data = [786,'abc','a',123.5,786,'rahul','b',786]
        print( "Number of times 123.5 occured is", data.count(123.5) )
        print( "Number of times 786 occured is", data.count(786)  )
```

Output:

```
        >>>
        Number of times 123.5 occured is 1
        Number of times 786 occured is 3
        >>>
```

3) pop()/pop(int):

Eg:

```
        data = [786,'abc','a',123.5,786]
        print( "Last element is", data.pop() )
        print( "2nd position element:", data.pop(1)  )
        print( data  )
```

Output:

```
>>>
        Last element is 786
        2nd position element:abc
        [786, 'a', 123.5]
>>>
```

4) insert(index,object):

Eg:

```
        data=['abc',123,10.5,'a']
        data.insert(2,'hello')
        print( data  )
```

Output:

```
>>>
        ['abc', 123, 'hello', 10.5, 'a']
>>>
```

5) extend(sequence):

Eg:

```
        data1=['abc',123,10.5,'a']
        data2=['ram',541]
        data1.extend(data2)
        print( data1 )
        print( data2  )
```

Output:

```
>>>
        ['abc', 123, 10.5, 'a', 'ram', 541]
        ['ram', 541]
>>>
```

6) remove(object):

Eg:

```
        data1=['abc',123,10.5,'a','xyz']
        data2=['ram',541]
        print( data1)
        data1.remove('xyz')
        print( data1 )
        print( data2  )
        data2.remove('ram')
        print( data2  )
```

Output:

```
>>>
        ['abc', 123, 10.5, 'a', 'xyz']
        ['abc', 123, 10.5, 'a']
        ['ram', 541]
        [541]
>>>
```

7) reverse():

Eg:

```
        list1=[10,20,30,40,50]
        list1.reverse()
        print( list1  )
```

Output:

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

8) sort():

```
        Eg:
        list1=[10,50,13]
        list1.sort()
        print( list1 )
```

Output:

```
>>>
        [10, 13, 50]
>>>
```

Python Tuple

A tuple is a sequence of immutable objects, therefore tuple cannot be changed.
The objects are enclosed within parenthesis and separated by comma.
Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List have mutable objects whereas Tuple have immutable objects.

eg:

```
>>> data=(10,20,'ram',56.8)
>>> data2="a",10,20.9
>>> data
(10, 20, 'ram', 56.8)
>>> data2
('a', 10, 20.9)
>>>
```

NOTE: If Parenthesis is not given with a sequence, it is by default treated as Tuple.
There can be an empty Tuple also which contains no object.

eg:

```
tuple1=()
```

For a single valued tuple, there must be a comma at the end of the value.

eg:

```
Tuple1=(10,)
```

Tuples can also be nested.

eg:

```
tupl1='a','mahesh',10.56
tupl2=tupl1,(10,20,30)
   print( tupl1)
   print( tupl2 )
```

Output:

```
>>>
        ('a', 'mahesh', 10.56)
        (('a', 'mahesh', 10.56), (10, 20, 30))
>>>
```

Accessing Tuple

Tuple can be accessed in the same way as List.
Some examples are given below:

eg:

```
data1=(1,2,3,4)
data2=('x','y','z')
print( data1[0]  )
print( data1[0:2])
print( data2[-3:-1])
```

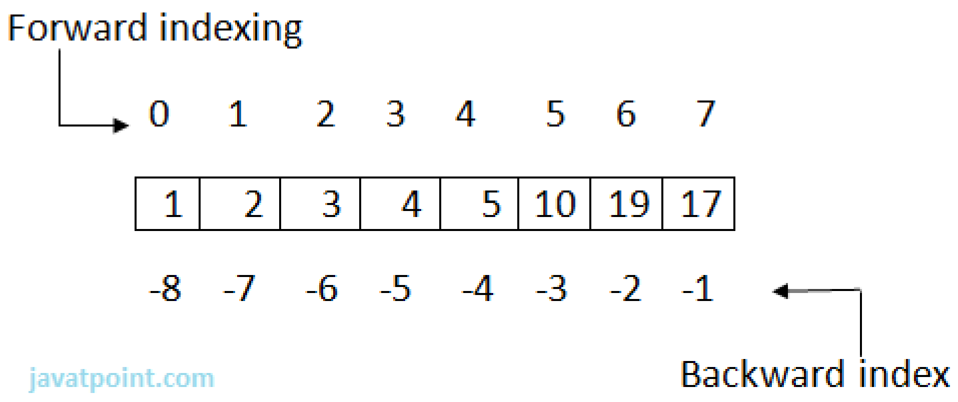
```
        print( data1[0:] )
        print( data2[:2]  )
```

Output:

```
>>>
        1
        (1, 2)
        ('x', 'y')
        (1, 2, 3, 4)
        ('x', 'y')
>>>
```

Elements in a Tuple

Data=(1,2,3,4,5,10,19,17)



Data[0]=1=Data[-8] , Data[1]=2=Data[-7] , Data[2]=3=Data[-6] , Data[3]=4=Data[-5] , Data[4]=5=Data[-4] , Data[5]=10=Data[-3], Data[6]=19=Data[-2],Data[7]=17=Data[-1]

Tuple Operations

Various Operations can be performed on Tuple. Operations performed on Tuple are given as:

a) Adding Tuple:

Tuple can be added by using the concatenation operator(+) to join two tuples.

eg:

```
        data1=(1,2,3,4)
        data2=('x','y','z')
        data3=data1+data2
        print( data1 )
        print( data2 )
        print( data3 )
```

Output:

```
>>>
        (1, 2, 3, 4)
        ('x', 'y', 'z')
        (1, 2, 3, 4, 'x', 'y', 'z')
```

```
>>>
```

Note: The new sequence formed is a new Tuple.

b) Replicating Tuple:

Replicating means repeating. It can be performed by using '*' operator by a specific number of time.

Eg:

```
        tuple1=(10,20,30)
        tuple2=(40,50,60)
        print( tuple1*2 )
        print( tuple2*3 )
```

Output:

```
>>>
        (10, 20, 30, 10, 20, 30)
        (40, 50, 60, 40, 50, 60, 40, 50, 60)
>>>
```

c) Tuple slicing:

A subpart of a tuple can be retrieved on the basis of index. This subpart is known as tuple slice.

Eg:

```
        data1=(1,2,4,5,7)
        print( data1[0:2] )
        print( data1[4] )
        print( data1[:-1])
        print( data1[-5:] )
        print( data1  )
```

Output:

```
>>>
        (1, 2)
        7
        (1, 2, 4, 5)
        (1, 2, 4, 5, 7)
        (1, 2, 4, 5, 7)
>>>
```

Note: If the index provided in the Tuple slice is outside the list, then it raises an IndexError exception.

Other Operations:

a) Updating elements in a List:

Elements of the Tuple cannot be updated. This is due to the fact that Tuples are immutable. Whereas the Tuple can be used to form a new Tuple.

Eg:

```
        data=(10,20,30)
        data[0]=100
        print( data )
```

Output:

```
>>>
Traceback (most recent call last):
         File "C:/Python27/t.py", line 2, in
        data[0]=100
TypeError: 'tuple' object does not support item assignment
>>>
```

Creating a new Tuple from existing:

Eg:

```
        data1=(10,20,30)
        data2=(40,50,60)
        data3=data1+data2
        print( data3 )
```

Output:

```
>>>
        (10, 20, 30, 40, 50, 60)
>>>
```

b) Deleting elements from Tuple:

Deleting individual element from a tuple is not supported. However the whole of the tuple can be deleted using the del statement.

Eg:

```
        data=(10,20,'rahul',40.6,'z')
        print( data )
        del data      #will delete the tuple data
        print( data)  #will show an error since tuple data is already deleted
```

Output:

```
>>>
        (10, 20, 'rahul', 40.6, 'z')
        Traceback (most recent call last):
                File "C:/Python27/t.py", line 4, in
                print( data)
        NameError: name 'data' is not defined
>>>
```

Functions of Tuple:

There are following in-built Type Functions:

Function	**Description**
min(tuple)	Returns the minimum value from a tuple.
max(tuple)	Returns the maximum value from the tuple.
len(tuple)	Gives the length of a tuple
tuple(sequence)	Converts the sequence into tuple.

1) min(tuple):

Eg:

```
        data=(10,20,'rahul',40.6,'z')
        print( min(data) )
```

Output:

```
>>>
        10
>>>
```

2) max(tuple):

Eg:

```
        data=(10,20,'rahul',40.6,'z')
        print( max(data) )
```

Output:

```
>>>
        z
>>>
```

3) len(tuple):

Eg:

```
        data=(10,20,'rahul',40.6,'z')
        print( len(data)  )
```

Output:

```
>>>
        5
>>>
```

4) tuple(sequence):

Eg:

```
        dat=[10,20,30,40]
        data=tuple(dat)
        print( data  )
```

Output:

```
>>>
        (10, 20, 30, 40)
>>>
```

Why Use Tuple?

1. Processing of Tuples are faster than Lists.
2. It makes the data safe as Tuples are immutable and hence cannot be changed.
3. Tuples are used for String formatting.

Python Dictionary

- Dictionary is an unordered set of key and value pair.
- It is a container that contains data, enclosed within curly braces.
- The pair i.e., key and value is known as item.
- The key passed in the item must be unique.
- The key and the value is separated by a colon(:). This pair is known as item. Items are separated from each other by a comma(,). Different items are enclosed within a curly brace and this forms Dictionary.

eg:

```
        data={100:'Ravi' ,101:'Vijay' ,102:'Rahul'}
        print( data )
```
Output:
```
>>>
        {100: 'Ravi', 101: 'Vijay', 102: 'Rahul'}
>>>
```
Dictionary is mutable i.e., value can be updated.
Key must be unique and immutable. Value is accessed by key. Value can be updated while key cannot be changed.
Dictionary is known as Associative array since the Key works as Index and they are decided by the user.

eg:
```
        plant={}
        plant[1]='Ravi'
        plant[2]='Manoj'
        plant['name']='Hari'
        plant[4]='Om'
        print( plant[2]  )
        print( plant['name']  )
        print( plant[1]  )
        print( plant   )
```
Output:
```
>>>
        Manoj
        Hari
        Ravi
        {1: 'Ravi', 2: 'Manoj', 4: 'Om', 'name': 'Hari'}
>>>
```
Accessing Values

Since Index is not defined, a Dictionaries value can be accessed by their keys.

Syntax:

[key]

Eg:
```
        data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
        data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
        print( "Id of 1st employer is",data1['Id']
        print( "Id of 2nd employer is",data2['Id']
        print( "Name of 1st employer:",data1['Name']
        print( "Profession of 2nd employer:",data2['Profession']
```
Output:
```
>>>
        Id of 1st employer is 100
        Id of 2nd employer is 101
        Name of 1st employer is Suresh
        Profession of 2nd employer is Trainer
>>>
```

Updation

The item i.e., key-value pair can be updated. Updating means new item can be added. The values can be modified.

Eg:

```
        data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
        data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
        data1['Profession']='Manager'
        data2['Salary']=20000
        data1['Salary']=15000
        print( data1)
        print( data2)
```

Output:

```
>>>
        {'Salary': 15000, 'Profession': 'Manager','Id': 100, 'Name': 'Suresh'}
        {'Salary': 20000, 'Profession': 'Trainer', 'Id': 101, 'Name': 'Ramesh'}
>>>
```

Deletion

del statement is used for performing deletion operation.

An item can be deleted from a dictionary using the key.

Syntax:

```
        del  [key]
```

Whole of the dictionary can also be deleted using the del statement.

Eg:

```
        data={100:'Ram', 101:'Suraj', 102:'Alok'}
        del data[102]
        print( data
        del data
        print( data)   #will show an error since dictionary is deleted.
```

Output:

```
>>>
        {100: 'Ram', 101: 'Suraj'}

Traceback (most recent call last):
         File "C:/Python27/dict.py", line 5, in
          print( data
NameError: name 'data' is not defined
>>>
```

Methods

Dictionary Methods:

Methods	**Description**

keys()	Return all the keys element of a dictionary.
values()	Return all the values element of a dictionary.
items()	Return all the items(key-value pair) of a dictionary.
update(dictionary2)	It is used to add items of dictionary2 to first dictionary.
clear()	It is used to remove all items of a dictionary. It returns an empty dictionary.
copy()	It returns an ordered copy of the data.
has_key(key)	It returns a boolean value. True in case if key is present in the dictionary ,else false.
get(key)	Returns the value of the given key. If key is not present it returns none.

1) keys():

Eg:

```
        data1={100:'Ram', 101:'Suraj', 102:'Alok'}
        print( data1.keys())
```

Output:

```
>>>
        [100, 101, 102]
>>>
```

2) values():

Eg:

```
        data1={100:'Ram', 101:'Suraj', 102:'Alok'}
        print( data1.values())
```

Output:

```
>>>
        ['Ram', 'Suraj', 'Alok']
>>>
```

3) items():

Eg:

```
        data1={100:'Ram', 101:'Suraj', 102:'Alok'}
        print( data1.items())
```

Output:

```
>>>
        [(100, 'Ram'), (101, 'Suraj'), (102, 'Alok')]
>>>
```

4) update(dictionary2):

Eg:

```
        data1={100:'Ram', 101:'Suraj', 102:'Alok'}
        data2={103:'Sanjay'}
        data1.update(data2)
        print( data1)
        print( data2)
```

Output:
```
>>>
        {100: 'Ram', 101: 'Suraj', 102: 'Alok', 103: 'Sanjay'}
        {103: 'Sanjay'}
>>>
```
5) clear():

Eg:
```
        data1={100:'Ram', 101:'Suraj', 102:'Alok'}
        print( data1)
        data1.clear()
        print( data1)
```
Output:
```
>>>
        {100: 'Ram', 101: 'Suraj', 102: 'Alok'}
        {}
>>>
```
6) copy():

Eg:
```
        data={'Id':100 , 'Name':'Aakash' , 'Age':23}
        data1=data.copy()
        print( data1)
```
Output:
```
>>>
        {'Age': 23, 'Id': 100, 'Name': 'Aakash'}
>>>
```
7) has_key(key):

Eg:
```
        data={'Id':100 , 'Name':'Aakash' , 'Age':23}
        print( data.has_key('Age'))
        print( data.has_key('Email'))
```
Output:
```
>>>
        True
        False
>>>
```
8) get(key):

Eg:
```
        data={'Id':100 , 'Name':'Aakash' , 'Age':23}
        print( data.get('Age'))
        print( data.get('Email'))
```
Output:
```
>>>
        23
        None
>>>
```

Python Functions

- A Function is a self block of code.
- A Function can be called as a section of a program that is written once and can be executed whenever required in the program, thus making code reusability.
- A Function is a subprogram that works on data and produces some output.

Types of Functions:
There are two types of Functions.
a) Built-in Functions: Functions that are predefined. We have used many predefined functions in Python.
b) User- Defined: Functions that are created according to the requirements.

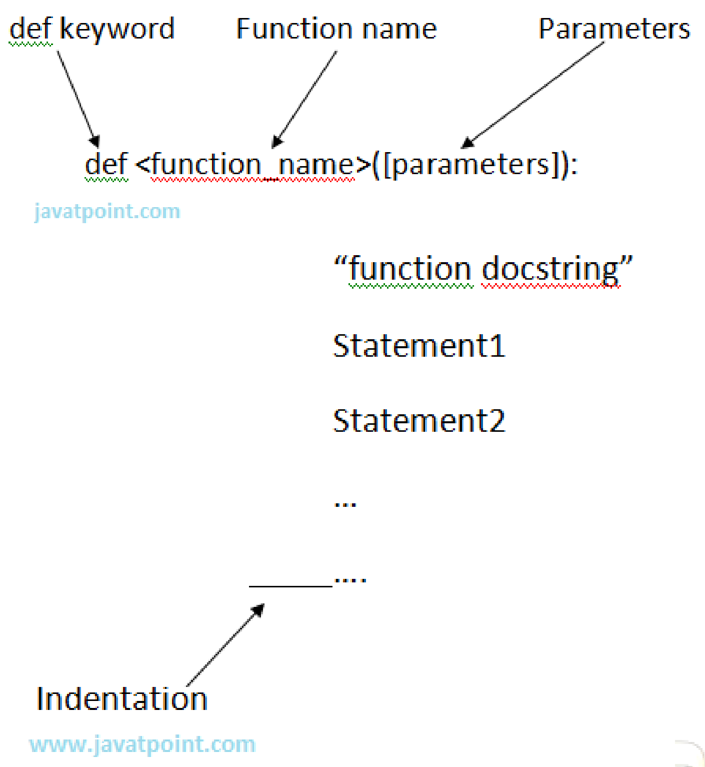
Defining a Function:
A Function defined in Python should follow the following format:
1) Keyword def is used to start the Function Definition. Def specifies the starting of Function block.
2) def is followed by function-name followed by parenthesis.
3) Parameters are passed inside the parenthesis. At the end a colon is marked.
Syntax:

```
        def <function_name>([parameters]):
```

eg:

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

4) Before writing a code, an Indentation (space) is provided before every statement. It should be same for all statements inside the function.
5) The first statement of the function is optional. It is <Documentation string> of function.
6) Following is the statement to be executed.
Syntax:



Invoking a Function:

To execute a function it needs to be called. This is called function calling.
Function Definition provides the information about function name, parameters and the definition what operation is to be performed. In order to execute the Function Definition it is to be called.

Syntax:

```
<function_name>(parameters)
```

eg:

```
sum(a,b)
```

Here sum is the function and a, b are the parameters passed to the Function Definition.
Let's have a look over an example:

eg:

```
#Providing Function Definition
def sum(x,y):
    "Going to add x and y"
     s=x+y
    print( "Sum of two numbers is"  )
      print( s )
      #Calling the sum Function
     sum(10,20)
     sum(20,30)
```

Output:

```
>>>
        Sum of two numbers is
        30
        Sum of two numbers is
```

```
    50
>>>
```

NOTE: Function call will be executed in the order in which it is called.

return Statement:

return[expression] is used to send back the control to the caller with the expression.
In case no expression is given after return it will return None.
In other words return statement is used to exit the Function definition.

Eg:

```
    def sum(a,b):
        "Adding the two values"
       print( "Printing within Function"  )
        print( a+b  )
      return a+b
    def msg():
        print ("Hello"  )
        return

    total=sum(10,20)
    print (“Printing Outside: “,total  )
    msg()
    print ("Rest of code"  )
```

Output:

```
>>>
    Printing within Function
    30
    Printing outside:  30
    Hello
    Rest of code
>>>
```

Argument and Parameter:

There can be two types of data passed in the function.
1) The First type of data is the data passed in the function call. This data is called ‘arguments’.
2) The second type of data is the data received in the function definition. This data is called ‘parameters’.
Arguments can be literals, variables and expressions.
Parameters must be variable to hold incoming values.
Alternatively, arguments can be called as actual parameters or actual arguments and parameters can be called as formal parameters or formal arguments.

Eg:

```
    def addition(x,y):
    print( x+y)
    x=15
```

```
            addition(x ,10)
            addition(x,x)
            y=20
            addition(x,y)
Output:
>>>
            25
            30
            35
>>>
```

Passing Parameters

Python supports following types of formal argument:

1) Positional argument (Required argument).
2) Default argument.
3) Keyword argument (Named argument)

Positional/Required Arguments:

When the function call statement must match the number and order of arguments as defined in the function definition it is Positional Argument matching.

Eg:

```
#Function definition of sum
            def sum(a,b):
                        "Function having two parameters"
                     c=a+b
                         print( c)

            sum(10,20)
            sum(20)
```

Output:

```
>>>
30

Traceback (most recent call last):
    File "C:/Python27/su.py", line 8, in <module>
        sum(20)
TypeError: sum() takes exactly 2 arguments (1 given)
>>>
```

Explanation:

1) In the first case, when sum() function is called passing two values i.e., 10 and 20 it matches with function definition parameter and hence 10 and 20 is assigned to a and b respectively. The sum is calculated and print(ed.

2) In the second case, when sum() function is called passing a single value i.e., 20 , it is passed to function definition. Function definition accepts two parameters whereas only one value is being passed, hence it will show an error.

Default Arguments

Default Argument is the argument which provides the default values to the parameters passed in the function definition, in case value is not provided in the function call.

Eg:

```
#Function Definition
        def msg(Id,Name,Age=21):
                "Printing the passed value"
                print( Id)
              print( Name)
              print( Age  )
              return
        #Function call
        msg(Id=100,Name='Ravi',Age=20)
        msg(Id=101,Name='Ratan')
```

Output:

```
>>>
        100
        Ravi
        20
        101
        Ratan
        21
>>>
```

Explanation:

1) In first case, when msg() function is called passing three different values i.e., 100 , Ravi and 20, these values will be assigned to respective parameters and thus respective values will be print(ed.

2) In second case, when msg() function is called passing two values i.e., 101 and Ratan, these values will be assigned to Id and Name respectively. No value is assigned for third argument via function call and hence it will retain its default value i.e, 21.

Keyword Arguments:

Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

Eg:

```
        def msg(id,name):
              "Printing passed value"
                  print( id  )
                  print( name )
            return
        msg(id=100,name='Raj')
        msg(name='Rahul',id=101)
```

Output:

```
>>>
    100
    Raj
    101
    Rahul
>>>
```

Explanation:

1) In the first case, when msg() function is called passing two values i.e., id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.

2) In second case, when msg() function is called passing two values i.e., name and id, although the position of two parameters is different it initialize the value of id in Function call to id in Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

Scope of Variable:

Scope of a variable can be determined by the part in which variable is defined. Each variable cannot be accessed in each part of a program. There are two types of variables based on Scope:

1) Local Variable.

2) Global Variable.

1) Local Variables:

Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

Eg:

```
    def msg():
        a=10
        print( "Value of a is",a )
        return

    msg()
    print( a) #it will show error since variable is local
```

Output:

```
>>>
    Value of a is 10

Traceback (most recent call last):
  File "C:/Python27/lam.py", line 7, in <module>
   print( a #it will show error since variable is local
NameError: name 'a' is not defined
>>>
</module>
```

b) Global Variable:

Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

Eg:

```
        b=20
        def msg():
                a=10
                print( "Value of a is",a )
                print( "Value of b is",b )
                return

                msg()
                print( b )
```

Output:

```
>>>
        Value of a is 10
        Value of b is 20
        20
>>>
```

Passing collections to a function

Let's assume, we are passing a list to a function.

eg:

```
        >>> def func1(list):
        print( list)
```

output:

```
        l1=[1,2,3,4]
        func1(l1)
        [1,2,3,4]
```

Variable Length of Parameters

We will introduce now functions, which can take an arbitrary number of arguments. The asterisk "*" is used in Python to define a variable number of arguments. The asterisk character has to precede a variable identifier in the parameter list.

eg:

```
        >>> def locations(city, *other_cities):
        print((city, other_cities)

        >>> locations("Paris")
        ('Paris', ())
```

```
>>> locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux", "Marseille")
('Paris', ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille'))
>>>
```

*** in Function Calls**

A * can appear in function calls as well, as we have just seen in the previous exercise: The semantics is in this case "inverse" to a star in a function definition. An argument will be unpacked and not packed. In other words, the elements of the list or tuple are singularized:

```
>>> def f(x,y,z):
    print(x,y,z)

>>> p = (47,11,12)
>>> f(*p)
(47, 11, 12)
```

Arbitrary Keyword Parameters

There is also a mechanism for an arbitrary number of keyword parameters. To do this, we use the double asterisk "**" notation:

```
>>> def f(**args):
...     print(args)
...
>>> f()
{}
>>> f(de="German",en="English",fr="French")
{'fr': 'French', 'de': 'German', 'en': 'English'}
>>>
```

Double Asterisk in Function Calls

The following example demonstrates the usage of ** in a function call:

```
>>> def f(a,b,x,y):
    print(a,b,x,y)
>>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
>>> f(**d)
('append', 'block', 'extract', 'yes')
```

and now in combination with *:

```
>>> t = (47,11)
>>> d = {'x':'extract','y':'yes'}
>>> f(*t, **d)
(47, 11, 'extract', 'yes')
>>>
```

Recursion

Recursion is the process of defining something in terms of itself.
A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of constructs are termed as recursive functions.
Following is an example of recursive function to find the factorial of an integer.
Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example:

```
# An example of a recursive function to
# find the factorial of a number
def calc_factorial(x):
    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, calc_factorial() is a recursive functions as it calls itself.
When we call this function with a positive integer, it will recursively call itself by decreasing the number.
Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)              # 1st call with 4
4 * calc_factorial(3)          # 2nd call with 3
4 * 3 * calc_factorial(2)      # 3rd call with 2
4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
4 * 3 * 2 * 1                  # return from 4th call as number=1
4 * 3 * 2                      # return from 3rd call
4 * 6                          # return from 2nd call
24                             # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.
Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Python Modules

What is a Module?

Consider a module to be the same as a code library.
A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("BCA6")
```

Output:

```
Hello, BCA6
```

Note: When using a function from a module, use the syntax: module_name.function_name.

Variables in Module

The module can contain functions, as already described, but also variables of all types (lists, dictionaries, objects etc):

Example

Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Output:

```
36
```

Naming a Module
You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module
You can create an alias when you import a module, by using the as keyword:
Example
Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Output:

```
36
```

Built-in Modules
There are several built-in modules in Python, which you can import whenever you like.
Example
Import and use the platform module:

```
import platform

x = platform.system()
print(x)
```

Output:

```
Windows
```

Using the dir() Function
There is a built-in function to list all the function names (or variable names) in a module. The dir() function:
Example
List all the defined names belonging to the math module:

```
import math
```

```
x = dir(math)
print(x)
```

Output:

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees',
'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

Note: The dir() function can be used on all modules, also the ones you create yourself.

Import From Module
You can choose to import only parts from a module, by using the from keyword.
Example
The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example
Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Note: When importing using the from keyword, do not use the module name when referring to elements in the module. Example: person1["age"], not mymodule.person1["age"]

File Handling
File handling is an important part of any web application.
Python has several functions for creating, reading, updating, and deleting files.

Python File Open
The key function for working with files in Python is the open() function.
The open() function takes two parameters filename, and mode.

There are four different methods (modes) for opening a file:
"r" - Read - Default value. Opens a file for reading, error if the file does not exist
"a" - Append - Opens a file for appending, creates the file if it does not exist
"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists
In addition you can specify if the file should be handled as binary or text mode
"t" - Text - Default value. Text mode
"b" - Binary - Binary mode (e.g. images)

Syntax
To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.
Note: Make sure the file exists, or else you will get an error.

Python File Open

Open a File on the Server
Asume we have the following file, located in the same folder as Python:
demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in open() function.
The open() function returns a file object, which has a read() method for reading the content of the file:
Example

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

Read Only Parts of the File
By default the read() method returns the whole text, but you can also specify how many character you want to return:

Example
Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines
You can return one line by using the readline() method:
Example
Read one line of the file:

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

By calling readline() two times, you can read the two first lines:

Example

Read two lines of the file:

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

By looping through the lines of the file, you can read the whole file, line by line:

Example

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Python File Write

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile.txt" and append content to the file:

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

Example

Open the file "demofile.txt" and overwrite the content:

```
f = open("demofile.txt", "w")
f.write("Woops! I have deleted the content!")
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Python Delete File

To delete a file, you must import the OS module, and run its os.remove() function:

Example1

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exist before you try to delete it:

Example

Check if file exist, then delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove empty folders.

Formatting Instructions:

Font Size: 22pt (Main Heading 1)
Font Size: 18pt (Sub Heading 2)
Font Size: 16pt (Sub Heading 3)
Font size: 12pt (Sub content)
Font Name: Time New Roman for all content
Line Spacing: 1.15
Bullet Style: Square (As mentioned above)
Number Style: Natural Numbers (i.e. 1,2,3) & Roman small numbers for sub content (i.e. i,ii,iii)
Table Style: Line Spacing 1, Alignment Left, Cell (Vertical Alignment Center), as per following format.

Title	**Title**	**Title**
1	Sample Text	Sample Text
1	Sample Text	Sample Text
1	Sample Text	Sample Text
1	Sample Text	Sample Text

For code: Use border as per following format. Font size 10 and background color light gray. And line spacing 1.

```
Void ()
Int x
```