Python Notes

Python File Extensions

.py - Default file format
.pyw – Windows file format
.pyc – Compiled version of python code

Comments

Single line comment

We can use # for commenting a single line. The commented statements will not be executed in python.

Example:

```
# This is single line comment in python
print “welcome to python”  # This will display message.
```

Multi line comment

There is no specific multi line comments in python. We can use triple single quotes (''') or triple double quotes (“””) as an alternate to achieve it.

Example:

```
''' This is
multiple line
comments in python
'''
```

Quotes

* Single quotes / Double quotes
* Triple single quotes / Triple double quotes

Sample Code:

```
# This is commented line
print "welcome 'to' \"python\" program" # diplay message
print 'welcome \'to\' "python" program' # diplay message
#print "This is second line and got commented"
print '''welcome
to python
program'''
```

Output:

```
welcome 'to' "python" program
welcome 'to' "python" program
welcome
to python
program
```

Escaping Characters

Some of the escaping characters in python are,

\n – newline

\t – tab

\\ - to print single \ character

Example:

```
>>> print "Thi\ts line contains\\ escaping\n characters"
 Thi    s line contains\ escaping
 characters
```

Variable Declaration

The variable data types are dynamic in python. It is also known as Dynamic Datatype Allocation. It will allocate data type based on the value assigned to the variable.

Example:

```
>>> a=100
>>> type(a)
<type 'int'>
>>> b=123.45
>>> type(b)
<type 'float'>
>>> name='python'
>>> type(name)
<type 'str'>
```

Print statement

print statement is used to display messages.

When to escape single quotes / double quotes ?

- Escape single quotes inside single quotes.
- Escape double quotes inside double quotes.

When to use single quotes / double quotes ?

- If the display message contains more single quotes, use entirely double quotes
- If the display message contains more double quotes, use entirely single quotes

When to use quotes ?

- Use quotes for displaying messages (strings).
- No quotes for variables.

Example:

```
 >>> print 'welcome to python'
 welcome to python
 >>> a=100
 >>> print a
 100
 >>> name = 'python'
 >>> print name
 python
```

To combine string and variable

There are 4 methods available to combine a string and a variable.

1. Comma operator (**,**)

Comma operator is used to combine a variable and string. A space will be added between the combined values.

```
>>> a=123; name='Python'
>>> print "The value of a is", a
The value of a is 123
>>> print "The value of name is", name
The value of name is Python
```

2. Plus operator (+)

Plus operator is used to concatenate a variable and string. In plus operator, we can concatenate values as below:

Number + Number (Addition)

String + String (Concatenation)

If we want to combine a string and number, then we need to typecast the number to string.

```
>>> a=123; name='Python'
>>> print "The value of a is "+ str(a)
The value of a is 123
>>> print "The value of name is "+ name
The value of name is Python
```

3. Percentage operator (**%**)

% operator is similar to 'C' language style format specification.

```
>>> print "My integer is %d and float is %f and string is %s" %(123, 45.67, 'Python')
My integer is 123 and float is 45.670000 and string is Python
```

4. Format method (**.format**)

format method is used to specify the strings as a whole and pass the values to the accessed index position.

```
>>> print "My integer is {0} and float is {1} and string is {2} and int is {0}" .format(123, 45.67, 'Python')
My integer is 123 and float is 45.67 and string is Python and int is 123
```

When to combine string or variable ?

Variable & Variable

Variable & String

String & Variable

Examples,

```
>>> a=123; b=45.67; name = 'Python'
>>> print "The value of a is", a ,"and b is", b ,"and name is", name
The value of a is 123 and b is 45.67 and name is Python
>>> print "The value of a is "+ str(a) +" and b is "+ str(b) +" and name is "+ name
The value of a is 123 and b is 45.67 and name is Python
>>> print "The value of a is %d and b is %f and name is %s" %(a, b, name)
The value of a is 123 and b is 45.670000 and name is Python
>>> print "The value of a is {0} and b is {1} and name is {2}" .format(a, b, name)
The value of a is 123 and b is 45.67 and name is Python
```

Read input from user

There are 4 methods available to get input from the user.

1. input('prompt message here')

input() method is used to get numbers as input from the user. The prompt message will be shown to the user while getting the input.

Example:

```
>>> empid = input('Enter the employee id: ')
Enter the employee id: 12345
>>> percentage = input("Enter the marks percentage: ")
Enter the marks percentatge: 89.56
print empid, percentage
12345 89.56
```

2. raw_input('prompt message here')

raw_input() method is used to get strings as input from the user. The prompt message will be shown to the user while getting the input.

Example:

```
>>> emp_name = raw_input("Enter the employee name: ")
Enter the employee name: Ramesh
>>> print emp_name
Ramesh
```

3. sys.stdin.readline()

readline() method is used to get a line as input from the user. Prompt message is not allowed. A newline will be added automatically.

Example:

```
>>> import sys
>>> line = sys.stdin.readline()
Welcome to python program
>>> line
'Welcome to python program\n'
>>> type(line)
<type 'str'>
```

4. sys.stdin.readlines()

readlines() method is used to get multiple lines as input from the user. Prompt message is not allowed. A newline will be added automatically to each lines.

Example:

```
>>> import sys
>>> lines = sys.stdin.readlines()
welcome
to python
program
It contains
multiple lines
>>> lines
['welcome\n', 'to python\n', 'program\n', 'It contains \n', 'multiple lines\n']
>>> type(lines)
<type 'list'>
```

Data Types

There are 5 basic data types in python.

Numbers - 123 (integer), 45.67 (float), True/False (boolean)
String - 'python', “perl”, '"welcome to python"', “””welcome to python”””
Tuple – (123, 45.67, “welcome”)
List – [123, 45.67, 'welcome']
Dictionary – {'id': 123, 'name': 'Ramesh', 'dept': 'CSC'}

Few points to remember,

- Python variables doesn't require explicit declaration to reserve memory. The memory allocation/declaration happens automatically when we assign value to a variable.
- Numbers, Strings and Tuples are immutable (unmodifiable).
- List and Dictionary are mutable (modifiable).
- Strings, Tuple and Lists will have auto indexing starts with 0.
- Tuples and Lists are similar to arrays. But it can store different data types.
- While Retrieving values,
  [start:stop:step] – String, Tuple, List
  [keyname] - Dictionary

General / Common functions

type()

type() method is used to display the data type of the variable.

```
>>> print type(10)
<type 'int'>
>>> print type('welcome')
<type 'str'>
>>> mynum=123
>>> print type(mynum)
<type 'int'>
>>> mystr = 'welcome to python'
>>> print type(mystr)
<type 'str'>
>>> t1 = (10, 20, 30)
>>> print type(t1)
<type 'tuple'>
>>> l1=['welcome', 'to', 'python']
>>> print type(l1)
<type 'list'>
>>> d1 = {'id': 123, 'name': 'python'}
>>> print type(d1)
<type 'dict'>
```

len()

len() method is used to find the length of the the sequence. It can a string, tuple or list.

```
>>> print len(mystr)
17
```

```
>>> print len(t1)
3
>>> print len(l1)
3
>>> print len('welcome')
7
```

dir()

dir() method is used to list the built-in variables and functions/methods related to the data type.

dir(variable) or dir(value)

```
>>> mystr = 'welcome to python'
>>> dir(mystr)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

help()

help() method is used to display the help / documentation page for a method/function.

```
>>> help(mystr.upper)
Help on built-in function upper:
  upper(...)
S.upper() -> string
Return a copy of the string S converted to uppercase.
```

```
>>> help(".lower)
Help on built-in function lower:
lower(...)
 S.lower() -> string
Return a copy of the string S converted to lowercase.
```

Numbers

Numbers data type is used to store numeric values. We can store integer, float, long and complex values as numbers.

```
a=123   (integer)
b=45.67   (float)
```

Strings

Strings are used to declare a set of characters within quotes (single or double).
A string subset can be taken using slicing operator([] or [:] or [::]) with indexes starts with 0 from the beginning and starts with -1 from the end.

```
>>> mystr = 'welcome to python'
>>> print mystr
welcome to python
>>> print len(mystr)
17
>>> print mystr[0]
w
>>> print mystr[5]
m
>>> print mystr[16]
n
>>> print mystr[-1]
n
>>> print mystr[-5]
y
>>> print mystr[-17]
w
```

Get first 7 characters,

```
>>> print mystr[0:7]
welcome
>>> print mystr[:7]
welcome
```

Skip first 8 characters

```
>>> print mystr[8:]
to python
```

Get any characters in the middle

```
>>> print mystr[8:10]
to
```

Get last 7 characters

```
>>> print mystr[-7:]
 python
```

Skip last 7 characters

```
>>> print mystr[:-7]
welcome to
```

Reverse the string

```
>>> print mystr[::-1]
nohtyp ot emoclew
```

Skip 2nd character in string,

```
>>> print mystr[::2]
'wloet yhn'
```

Concatenate the string

```
>>> print 'Python' + 'Program'
PythonProgram
>>> print mystr + ' program'
welcome to python program
```

Repetition in string,

```
>>> print 'welcome ' * 5
welcome welcome welcome welcome welcome
>>> name='Python'
>>> print name * 6
PythonPythonPythonPythonPythonPython
```

Invalid index access error,

```
>>> print mystr[20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

String functions

Some of the string related functions are as below,

```
>>> mystr = 'welcome TO pytHON'
>>> print mystr
   welcome TO pytHON
>>> print mystr.capitalize()
   Welcome to python
>>> print mystr.count('O')
   2
>>> print mystr.center(25, '*')
   ****welcome TO pytHON****

>>> print mystr.startswith('welcome')
  True
>>> print mystr.endswith('HON')
  True
>>> print mystr.find('TO')
  8
>>> print mystr.index('TO')
  8
>>> print mystr.replace('TO', 'tttooo')
  welcome tttooo pytHON
```

```
>>> print mystr.upper()
  WELCOME TO PYTHON
>>> print mystr.lower()
  welcome to python
>>> print mystr.swapcase()
  WELCOME to PYThon
```

Lists

Lists are similar to arrays in C. The data inside the list can be of different data types. A list contains items separated by comma and enclosed in square brackets ([]).

A list subset can be taken using slicing operator([] or [:] or [::]) with indexes starts with 0 from the beginning and starts with -1 from the end.

The plus (+) sign is list concatenation operator and asterisk (*) sign is repetition operator.

```
>>> mylist = [10, 20.5, 'welcome', "Python"]
>>> print type(mylist)
<type 'list'>
>>> print mylist
[10, 20.5, 'welcome', 'Python']
>>> print mylist[0]
10
>>> print mylist[2]
welcome
>>> print mylist[-1]
Python
>>> print mylist[1:3]
[20.5, 'welcome']
>>> print mylist[:3]
[10, 20.5, 'welcome']
>>> print mylist[1:]
[20.5, 'welcome', 'Python']
```

List functions

Some of the list related functions are as below,

```
>>> mylist = [10, 20, 30]
>>> print mylist
[10, 20, 30]
>>> mylist.append(60)
>>> print mylist
[10, 20, 30, 60]
>>> mylist.extend([50, 40])
>>> print mylist
[10, 20, 30, 60, 50, 40]
>>> mylist.insert(2, 100)
>>> print mylist
[10, 20, 100, 30, 60, 50, 40]
>>> mylist[4:4] = [20, 70]
>>> print mylist
[10, 20, 100, 30, 20, 70, 60, 50, 40]
```

```
>>> print mylist.count(20)
2
>>> print mylist.index(30)
3
>>> mylist.reverse()
>>> print mylist
[40, 50, 60, 70, 20, 30, 100, 20, 10]
>>> mylist.sort()
>>> print mylist
[10, 20, 20, 30, 40, 50, 60, 70, 100]
>>>
>>> mylist = [10, 20, 100, 30, 20, 70, 60, 50, 40]
>>> print mylist
[10, 20, 100, 30, 20, 70, 60, 50, 40]
>>> mylist.sort(reverse=True)
>>> print mylist
[100, 70, 60, 50, 40, 30, 20, 20, 10]
>>>
>>> mylist = [10, 20, 100, 30, 20, 70, 60, 50, 40]
>>> print mylist
[10, 20, 100, 30, 20, 70, 60, 50, 40]
>>> mylist.pop()
40
>>> print mylist
[10, 20, 100, 30, 20, 70, 60, 50]
>>> mylist.pop(3)
30
>>> print mylist
[10, 20, 100, 20, 70, 60, 50]
>>> mylist.remove(100)
>>> print mylist
[10, 20, 20, 70, 60, 50]
```

Tuples

Tuples are immutable version of lists. The data inside the tuple can be of different data types. A tuple contains items separated by comma and enclosed in parenthesis (()).

A tuple subset can be taken using slicing operator([] or [:] or [::]) with indexes starts with 0 from the beginning and starts with -1 from the end.

The plus (+) sign is tuple concatenation operator and asterisk (*) sign is repetition operator.

```
>>> mytuple = (10, 20.5, 'welcome', "Python")
>>> print mytuple
(10, 20.5, 'welcome', 'Python')
>>> print mytuple[0]
10
>>> print mytuple[-1]
Python
>>> print mytuple[2]
welcome
```

```
>>> print mytuple[1:3]
(20.5, 'welcome')
>>> print mytuple[:3]
(10, 20.5, 'welcome')
>>> print mytuple[1:]
(20.5, 'welcome', 'Python')
>>> print mytuple + (100, 200)
(10, 20.5, 'welcome', 'Python', 100, 200)
>>> print mytuple * 2
(10, 20.5, 'welcome', 'Python', 10, 20.5, 'welcome', 'Python')
```

Tuple functions

Some of the list related functions are as below,

```
>>> mytuple = (10, 20, 30, 15, 20, 10, 60)
>>> print mytuple
(10, 20, 30, 15, 20, 10, 60)
>>> print mytuple.count(10)
2
>>> print mytuple.index(15)
3
```

Dictionary

Dictionaries are kind of hash table types. It is a set of key-value pairs.

Dictionary keys can be of any type, but usually numbers or strings. Dictionary values can be any python objects.

Dictionary are enclosed in curly braces ({}) and values can be accessed using square brackets ([]).

```
>>> mydict = {'id': 123, 'name': "Ram"}
>>> print type(mydict)
<type 'dict'>
>>> print mydict
{'id': 123, 'name': 'Ram'}
>>> print mydict['id']
123
>>> print mydict['name']
Ram
>>> mydict['dept'] = 'CSC'
>>> print mydict
{'dept': 'CSC', 'id': 123, 'name': 'Ram'}
>>> mydict.update({'location': 'Chennai', 'salary': 15000})
>>> print mydict
{'salary': 15000, 'dept': 'CSC', 'location': 'Chennai', 'id': 123, 'name': 'Ram'}
>>> print mydict['dept']
CSC
>>> print mydict['salary']
15000
```

Dictionary functions

Some of the dictionary related functions are as below:

```
>>> print mydict.get('location')
Chennai
>>> print mydict.has_key('salary')
True
>>> print mydict.keys()
['salary', 'dept', 'location', 'id', 'name']
>>> print mydict.values()
[15000, 'CSC', 'Chennai', 123, 'Ram']
>>> print mydict.items()
[('salary', 15000), ('dept', 'CSC'), ('location', 'Chennai'), ('id', 123), ('name', 'Ram')]
>>>
>>> print mydict
{'salary': 15000, 'dept': 'CSC', 'location': 'Chennai', 'id': 123, 'name': 'Ram'}
>>> mydict.pop('location')
'Chennai'
>>> print mydict
{'salary': 15000, 'dept': 'CSC', 'id': 123, 'name': 'Ram'}
>>> mydict.popitem()
('salary', 15000)
>>> print mydict
{'dept': 'CSC', 'id': 123, 'name': 'Ram'}
>>> mydict.clear()
None
>>> print mydict
{}
>>> del mydict
>>> print mydict
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mydict' is not defined
```

Data Type Conversion / Type Casting

We can convert one data type to another based on need. Some common type conversion methods are,

```
>>> int('10')
10
>>> float("20.5")
20.5
>>> str(100)
'100'
>>> list((10, 20, 30))
[10, 20, 30]
>>> tuple([10, 20, 30])
(10, 20, 30)
```

Operators

Operators are used to manipulate the value of the operands.
Ex: 10 + 20 . In this, 10 and 20 are called operands and + is called operator.

Arithmetic Operators,

Arithmetic operators are used to do some calculations. Consider a = 10 and b = 5,

+ Addition → a + b = 15
- Subtraction → a – b = 5
* Multiplication → a * b = 50
/ Division → a / b = 2
% Modulus → 11 % 3 = 2
** Exponent / Power → 10 ** 2 = 100

Comparison Operators,

Comparison operator is used to compare the relationship between both sides. Consider a=10 and b=20,

(a == b) is not true
(a != b) or (a <> b) is true
(a > b) is not true
(a < b) is true
(a >= b) is not true
(a <= b) is true

Assignment Operators

Assignment operator is used to assign values to the variables. Consider a =10 and b = 20,

c = a + b (Assignment c=30)
a += b (Add and Assign. Which is equivalent to a = a+ b)
a -= b (Subtract and Assign)
a *= b (Multiply and Assign)
a /= b (Division and Assign)
a %= b (Modulus and Assign)

Logical Operators

Logical operators are used to check more than one conditions. Consider a=10 and b = 20,

(a and b) is true
(a or b) is true
not(a and b) is false

Membership Operators

Membership operator is used to test the membership in a sequence such as string, list and tuples.

x in y (true if x value is present in the y sequence.
x not in y (true is x value is not a member of y sequence)
Ex: 10 in (10, 20, 30) → True
'h' in 'python' → True

Identity Operators

Identity operator is used to compare the memory location of two objects.

x is y (true if both x and y points to same memory location)
x is not y (true if both x and y does not point to same memory location)

Conditions / Decision Making

if condition

Decision making statements are used to execure statements based on the conditions. Python consider and non-zero or non-null value are true and zero or null value as false.

if statement
if .. else statement
if .. elif .. else statement

We need to follow the indentation for the blocks in conditions.

Example:

```
print "Program started"
a=100

if a >=100 and a <=200:
  print "IF MATCH"
  print "The value of a is between 100 and 200"

  if a == 150:
    print " INNER IF MATCH"
    print " The value of a is 150"
  else:
    print " INNER ELSE MATCH"
    print " The value of a is not 150"
  print "IF BLOCK END"

elif a > 200 and a <=300:
  print "ELIF MATCH"
  print "The value of a is between 200 and 300"
  if a == 250:
    print " INNER IF MATCH"
    print " The value of a is 250"
  else:
    print " INNER ELSE MATCH"
    print " The value of a is not 150"
  print "ELIF BLOCK END"

else:
  print "ELSE MATCH"
  print "The value of a is not within range"

print "Program completed"
```

Loops

Loops are used to execute the statement repeatedly for number of times. It allows to execute a statement or group of statements multiple times.

for loop

for loop is used to iterate the values from the sequence such as string, list and tuples. We can use for loop as two types.

To print/get values from sequence, (Type-1)

```
for  tmp_var  in sequence:
    print tmp_var
```

To print/get index and value from sequence, (Type-2)

```
for  index  in  range(len(sequence)):
   print index,  sequence[index]
```

```
>>>for c in 'apple':
        print c
a
p
p
l
e

>>> mystr = 'python'
>>> for i in mystr:
         print i
p
y
t
h
o
n
>>> for i in range(len(mystr)):
          print i, mystr[i]
0 p
1 y
2 t
3 h
4 o
5 n

>>> mylist = [10, 20, 30]
>>> for i in mylist:
         print i
10
20
30
```

while loop

while loop is used to execute the set of statements while the condition is TRUE.

```
while expression:
   statements ...
```

Example:

To print the values in ascending order,

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

To print the values in descending order,

```
>>> i=5
>>> while i>=1:
        print i
        i-=1
5
4
3
2
1
```

To print the multiples of any number,

```
>>> i=1
>>> while i<=25:
        if i%5 == 0:
          print i
          i+=5
        else:
          i+=1
5
10
15
20
25
```

Loop Control Statements

Loop control statements are used to control the execution of the loops.

break statement

break statement is used to terminate the loop and transfer execution to the statement immediately following the loop.

```
>>> while i<=5:
        print i
        if i == 3: break
        i+=1
1
2
3
```

continue statement

continue statement is used to go to the next iteration of the loop.

```
>>> for l in 'python':
        if l == 'h':
          continue
        print "Letter: ", l
Letter:  p
Letter:  y
Letter:  t
Letter:  o
Letter:  n
```

pass statement

pass statement is used when a statement is required syntactically but we don't want any command or code to execute.

```
>>> for l in 'python':
         if l == 'h':
           pass
           print "This is pass block"
         print "Letter: ", l
Letter:  p
Letter:  y
Letter:  t
This is pass block
Letter:  h
Letter:  o
Letter:  n
```

Sets

Sets are unordered collection of unique values. It is also one of the data types in python.

```
>>> s1 = set(['A', 'B', 'C', 'D', 'E', 'A', 'C'])
>>> type(s1)
<type 'set'>
>>> print s1
set(['A', 'C', 'B', 'E', 'D'])
>>> s2 = set(['D', 'E', 'F', 'G', 'D', 'E'])
>>> print s2
set(['E', 'D', 'G', 'F'])
>>> s1 & s2
 set(['E', 'D'])
>>> s1 | s2
 set(['A', 'C', 'B', 'E', 'D', 'G', 'F'])
>>> s1 - s2
 set(['A', 'C', 'B'])
>>> s1 ^ s2
 set(['A', 'C', 'B', 'G', 'F'])
```

Looping Techniques

Looping techniques are used to do operations on the loops.

enumerate()

enumerate() method is used to get/fetch the index and value from the sequence.

```
>>> mylist = [10, 20, 30]
>>> for i,v in enumerate(mylist):
        print i, v
 0 10
 1 20
 2 30
```

zip()

zip method is used to process more than one sequence simultaneously.

```
>>> ques = ('id', 'name', 'location')
>>> ans  = (123, 'Ramesh', 'Chennai')
>>> for q,a in zip(ques, ans):
        print "Your {0} is {1}".format(q, a)
 Your id is 123
 Your name is Ramesh
 Your location is Chennai
```

List Comprehension

List comprehension is used to do the process or calculations and return result as a list.

```
>>> result = [i*5 for i in range(1,11)]
>>> print type(result), " -> ", result
 <type 'list'>  ->  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
>>> result = [i*5 for i in range(1,11) if i!=5]
>>> print result
 [5, 10, 15, 20, 30, 35, 40, 45, 50]
```

Functions

Function is a block of reusable code that is used to perform some actions. If provides better modularity and high re-usability of the code.

Python provides many built-in functions such as print(), type(), len() and etc..

We can also create our own functions and it is called as user-defined functions.

Syntax,

```
def func_name([arguments]):
  statements..
  [return value]
```

Functions with arguments and return value

A function can have arguments and return values. Both are optional. The statements inside function will be executed while calling the function.

Program:

```
# Simple function  (function_args.py)
def myfunc():
  print "This is simple function"
  print "This is second line"

# Function with args
def add(x, y):
  print "The given args are:", x, y
  print "Addition: ", x + y

# Function with args and return value
def mul(a, b):
  return a * b

## Calling functions
myfunc()
add(10, 20)
print "Multiplication: ", mul(5, 6)
i=10
j=20
k = mul(i, j)
print "Multiplication of {0} and {1} is {2}" .format(i, j, k)
```

Result:

```
$ python function_args.py
 This is simple function
 This is second line
 The given args are: 10 20
 Addition:  30
 Multiplication:  30
 Multiplication of 10 and 20 is 200
```

Function with keyword args, default args & variable length args

- A default argument is an argument that assumes a default value if a value is not passed while calling the function. It will be done while declaring the function.
- Keyword arguments are related to function call. If we use keyword arguments in function call, then the caller identifies the argument by parameter name instead of given order.
- Normally if we pass list/tuple/dictionary as argument to a function then the entire list/tuple/dictionary will be passed as single argument to the function. If we want to pass values inside a list/tuple/dictionary as arguments to the function, then we need to use * before a list/tuple and ** before a dictionary.

Program:

```
# Function with default args
def add(x=50, y=60):
  print "The given args are:", x, y
  print "Addition: ", x + y

# Function with variable length args
def myfunc(x, y):
  print "The given args are: ", x, y

add(100, 200)
add(100)
add()
add(y=500, x=600) # Func with keyword args

a=10; b=20
myfunc(a,b)

mylist=[300, 400]
myfunc(*mylist)  # To pass list/tuple values as arguments use *
mydict = {'y': 700, 'x': 800}
myfunc(**mydict)  # To pass dictionary values as arguments use **
```

Result:

```
$ python function_default_keyword.py
The given args are: 100 200
Addition:  300
The given args are: 100 60
Addition:  160
The given args are: 50 60
Addition:  110
The given args are: 600 500
Addition:  1100
The given args are:  10 20
The given args are:  300 400
The given args are:  800 700
```

Global and Local variables

Variables defined inside a function body have a local scope, and those defined outside have a global scope.

Local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Program:

```
a=100 # Global variable / scope

def myfunc():
  global a # Declare global scope of a
  a = 200  # Changing value of global variable
  b = 150  # Local variable / scope
  print "The value of a is", a
  print "The value of b is", b

print "A before function call: ", a
myfunc()
print "A after function call: ", a
```

Result:

```
$ python function3.py
A before function call:  100
The value of a is 200
The value of b is 150
A after function call:  200
```

Lambda functions

Lambda functions are used to define simple calculations or expressions. It is also similar to functions.

Lambda functions are called as anonymous functions.

Syntax:

```
lambda  [args]:  expression
```

Example:

```
>>> add = lambda x,y: x+y
>>> add(100, 200)
300
>>> type(add)
<type 'function'>
>>>
>>> mul = lambda a,b,c:  a*b*c
>>> mul(2,3,4)
24
```

return statement

return statement is exits a function, and optionally passing back an expression to the caller. A return statement with no arguments is same as return None.

Errors and Exception Handling

There are 3 types of errors in python. A error will disturb the normal flow of the program.

Syntax errors (Ex. print 'welcome to)
Runtime errors (Ex. print 10 + 'acd')
Logical errors

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of program.

Program

```
import sys
try:
  print a
  print 10 + 'abc'
  import abcde
except NameError as err:
  print "NAMEERROR Caught!!!", err
except TypeError as t1:
  print "TYPEERROR Caught!!!", t1
except:
  print "Exception caught.", sys.exc_info()[0]
finally:
  print "This is finally block"

print "Program completed"
```

Result

```
NAME ERROR Caught!!! name 'a' is not defined
This is finally block
Program completed
```

Raise Exception

raise() method is used to raise the exception similar to system generated exception.
It will generate the exception with the given user defined error message.

```
>>> raise NameError('This is generated name error')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: This is generated name error
```

Files and Directories

To print the current working directory and list files/directories inside the directory,

```
>>> import os
>>> os.getcwd()
 '/home/manoj/Python/Scripts'
>>> os.listdir('.')
 ['except.py', 'signal.py', 'mymath.pyc', 'inheritance.py', 'function2.py', 'database.py', 'client.py']
```

To create a folder/ change to folder / delete a folder,

```
>>> os.mkdir('MYFOLDER')
>>> os.chdir('MYFOLDER')
>>> os.getcwd()
 '/home/manoj/Python/Scripts/MYFOLDER'
>>> os.chdir('..')
>>> os.rmdir('MYFOLDER')
```

To get file related information,

```
>>> os.stat('sample.txt')
 posix.stat_result(st_mode=33204, st_ino=5357725, st_dev=2049L, st_nlink=1, st_uid=1000,
st_gid=1000, st_size=64, st_atime=1543738347, st_mtime=1543738347, st_ctime=1543738347)
>>> os.path.getsize('sample.txt')
 64
>>> os.path.getmtime('sample.txt')
 1543738347.0399344
>>> os.path.isfile('sample.txt')
 True
>>> os.path.exists('sample.txt')
 True
>>> os.path.abspath('sample.txt')
 '/home/manoj/Python/Scripts/sample.txt'
>>> os.path.dirname('/home/manoj/Python/Scripts/sample.txt')
 '/home/manoj/Python/Scripts'
>>> os.path.basename('/home/manoj/Python/Scripts/sample.txt')
 'sample.txt'
>>> os.rename('sample.txt', 'sample1.txt')
>>> os.remove('sample1.txt')
```

Read/ Write content in file

```
>>> fp = open('sample.txt', 'w')  # open file in write mode. We can use 'w' or 'a' mode to open file.
>>> fp.write('welcome to python\n')  # To write a line
>>> lines = ['welcome to\n', 'python\n', 'program\n']
>>> fp.writelines(lines)  # To write multiple lines
>>> fp.close()

>>> fp = open('sample.txt', 'r') # To open file in read mode & read contents from the file
>>> for line in fp:
        print line,
>>> fp.close()
```

Modules

A module is a collection of variables, functions and classes.

Namespace is a memory location where the variables and functions are stored.

In below program execution, note down the namespace while executing the program directly and while importing and executing it as a module.

Program:

```
# Math module  (mymath.py)
import sys
MYVAR = 100

def getNameSpace():
  print "Namespace: ", __name__

def add(x,y):
  getNameSpace()
  return x + y

def mul(x,y):
  getNameSpace()
  return x * y

if __name__ == "__main__":
  print add(100,200)
  #print "Multiplication: ", mul(100, 5)
```

To execute the program

When we execute a program, that program will be in main namespace.

```
$ python mymath.py
Namespace:  __main__
300
```

To import the module and use it,

When we import a module, it will be in separate namespace.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/dist-packages']
>>> sys.path.append('/home/manoj/Python/Scripts')
>>>
>>> import mymath
>>> mymath.add(10, 20)
 Namespace:  mymath
 30
```

Types of importing a module,

```
 >>> import mymath                  # Functions and variables will be in mymath namespace
 >>> from mymath import add, mul  # This will import add and mul function in main namespace
 >>> from mymath import *          # This will import all variables and functions in main namespace
 >>> reload(mymath)                # To reload a module again
```

Packages

Package is a hierarchical file directory structure that defines a single python application that consists of modules and sub packages and so on.

Create a folder named “vehicle/” and create files inside that folder.

vehicle/bike.py

```
def mybike():
  print "This is bike() function."
```

vehicle/bus.py

```
def mybus():
  print "This is bus() function."
```

vehicle/cars.py

```
def mycar():
  print "This is car() function."
```

vehicle/__init__.py *(This is a mapping file. It contains the mapping of modules for a package)*

```
import bike
from bus import mybus
from cars import *
```

Import the package and use it

```
>>> import vehicle
>>> dir(vehicle)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__', 'bike', 'bus', 'cars',
'mybus', 'mycar']
>>> vehicle.bike.mybike()
This is bike() function.
>>> vehicle.bus.mybus()
This is bus() function.
>>> vehicle.cars.mycar()
This is car() function.

>>> vehicle.mycar()
This is car() function.
>>> vehicle.mybus()
This is bus() function.

>>> print vehicle.__file__
vehicle/__init__.py
>>> print vehicle.__name__
vehicle
>>> print vehicle.__package__
vehicle
>>> print vehicle.__path__
['vehicle']
```

Object Oriented Programming

Classes & Objects

Class is a collection of variables and functions. It can be accessed using objects.

```
# Simple Class
class MyClass():
  '''This is My First Class'''
  myvar = 100  # Class variable

  def myfunc(self):
    print "This is myfunc() method."
    self.mystr = "welcome to python"  # Object / Instance variable

print " Creating Object ".center(30, "*")
myobj = MyClass()
print myobj.myvar
myobj.myfunc()
print myobj.mystr
print "CAlling Using Classname".center(50, "-")
print MyClass.myvar

print "Class Properties".center(30, "*")
print "Name: ", MyClass.__name__
print "Dict: ", MyClass.__dict__
print "Doc: ", MyClass.__doc__
print "Module: ", MyClass.__module__
```

Constructor & Destructor

Constructor method will be called automatically while creating the objects.
Destructor method will be called automatically while deleting the object.

```
## Class with Constructor and Destructor
class MyClass():
  myvar = 100

  # Constructor Method. It will be called at the time of creating the object.
  def __init__(self, myid, myname):
    print "Constructor method called. ", self.__class__.__name__
    self.name = myname
    self.myid = myid

  # Defining a class function. The first argument will be self.
  def myfunc(self):
    print "This is myfunc() method."

  # Destructor Method. It will be called while deleting the object.
  def __del__(self):
    print "This is destructor method.", self
```

```
print "Creating Object".center(30, "*")
myobj = MyClass(1001, 'Ramesh')
print myobj.myvar
myobj.myfunc()

print "Creating Object".center(30, "*")
myobj1 = MyClass(1002, 'Suresh')
print myobj1.myvar
myobj1.myfunc()
del(myobj1)
```

Inheritance

Inheritance is a process of accessing one class members such as variables and functions in another class.

Program

```
class Grandparent():
  def grandparent(self):
    print "This is grandparent function"

#Single level inheritance. Parent has access to Grandparent
class Parent(Grandparent):
  def parent(self):
    print "This is parent function"

# Multi-level inheritance. Child has access to Parent which has access to Grandparent
class Child(Parent):
  def child(self):
    print "This is child function"

  # Overriding the parent() method in child class.
  def parent(self):
    print "This is parent() function in child class"

print "Grandparent Object".center(30, '*')
g1 = Grandparent()
g1.grandparent()

print "Parent object".center(30, '*')
p1 = Parent()
p1.parent()
p1.grandparent()

print "Child object".center(30, '*')
c1 = Child()
c1.child()
c1.parent()
c1.grandparent()
```

Generators and Decorators

Iterators:

Iterator method is used to iterate elements from object as one by one.

```
mylist=[10,20,30]
for i in mylist:
  print i
it = iter(mylist)
it.next()
```

Generators:

Generators are used to generate iterators. We can use yield statement to generate iterators. Normal function has return value and and generator has yield value. It internally maintain references.

```
def mygenerator():
  print "Welcome"
  for i in (10,20):
    yield i
it=iter(mygenerator())
it.next()
```

Decorators:

Decorator provide a very useful method to add functionality to existing functions and classes.
Decorators are functions that wrap other functions or classes.

Ex:

```
def math(func, x, y):
  print "Math Function"
  func(x,y)

def add(a,b):
  print "This is addition function"
  print a + b

def mul(a,b):
  print "This is subtraction function"
  print a * b

math(add, 10,20)
math(mul,10,20)
```

Date and Time Module

```
>>> import time
>>> time.time()
1543859119.511833
>>> time.localtime(time.time())
time.struct_time(tm_year=2018, tm_mon=12, tm_mday=3, tm_hour=23, tm_min=15, tm_sec=32,
tm_wday=0, tm_yday=337, tm_isdst=0)
>>> time.localtime(time.time())[0]
2018
>>> time.strftime('%Y/%m/%d %H:%M:%S', time.localtime(time.time()))
'2018/12/03 23:16:16'
```

System Module

```
>>> import sys
>>> sys.version
'2.7.6 (default, Oct 26 2016, 20:30:19) \n[GCC 4.8.4]'
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=6, releaselevel='final', serial=0)
>>> sys.platform
'linux2'
```

Host & User Information

```
>>> import socket, getpass
>>> socket.gethostname()
'HP-Compaq'
>>> socket.gethostbyname(socket.gethostname())
'127.0.0.1'
>>> getpass.getuser()
'manoj'
```

Sleep & Exit

```
>>> import sys, time
>>> time.sleep(5)
>>> sys.exit(0)
```

Minimum & Maximum

```
>>> min('python')
'h'
>>> max([10, 20, 40, 35, 30])
40
```

Calendar Module

```
>>> import calendar
>>> print calendar.month(2018, 9)
   September 2018
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Math Module

```
>>> math.sqrt(100)
10.0
>>> math.pow(5, 2)
25.0
>>> math.ceil(10.5)
11.0
>>> math.floor(10.5)
10.0
```

Regular expressions

Regular expressions are a special sequence of characters/patterns that is used to match/find strings.
We can use 're' module for regex usage. For specifying patterns it is always better to use raw strings.

Match / Search:

We can use below two methods to search for a pattern in the strings.

re.match(pattern, string, flags)
re.search(pattern, string, flags)

Both functions return a match object on success and none on failure. We can use group(n) or groups() function to get the matched string for given pattern.
group(num=0) method returns entire match or sub-group of match.
groups() method returns all matching subgroups in a tuple. It will be empty if no grouping in patterns.
The difference between match and search is match will search for the pattern in the beginning of the string and search will search for the pattern anywhere in the string.

```
>>> import re
>>> line = 'welcome to python program'
>>>
>>> mobj = re.match('welcome', line)
>>> print mobj
<_sre.SRE_Match object at 0x7f11969cf370>
>>> mobj.groups()
()
>>> mobj.group()
'welcome'
>>> mobj = re.match('to', line)
>>> print mobj
None
>>>
>>> mobj = re.search('welcome', line)
>>> print mobj
<_sre.SRE_Match object at 0x7f11969cf370>
>>> print mobj.group()
welcome
>>> mobj = re.search('to', line)
>>> print mobj.group()
to
```

Search & Replace

We can use substitute method to do the find and replace of patterns in the strings.

re.sub(pattern, replace, string, max=0)

```
>>> line
'welcome to python program'
>>> re.sub('python', 'PYTHON', line)
'welcome to PYTHON program'
```

Some of the regular expression patterns are as below:
^ - Matches beginning of line
$ - Matches end of line
| - Alternate match
. - Single character match except newline
? - 0 or 1 occurrence of previous character
+ - 1 or more occurrence of previous character
* - 0 or more occurrence of previous character

[abc] - Matches any single character in the brackets. Matches a or b or c
[^abc] - Matches any character apart from given characters in brackets
(abc) - Grouping of string. Matches 'abc'
{m} - Matches exactly 'm' times of previous expression
{m,} - Matches m or more times of preceding expression
{m,n} - Matches at least 'm' times and most of 'n' times of preceding expression

\d - Matches digits. Equivalent to [0-9]
\D - Matches any non-digit
\w - Matches any single word character. [0-9a-zA-Z_]
\W - Matches any non-word character
\s - Matches whitespace
\S - Matches any non-whitespace
\1 - Back reference in grouping. \1, \2, \3, … \n

Examples:

```
>>> mobj = re.search(r'WELCOME (.*) program', line, re.I)
>>> print mobj.group()
welcome to python program
>>> mobj = re.search(r'WELCOME (.*) pro', line, re.I)
>>> print mobj.group()
welcome to python pro
>>> print mobj.groups()
('to python',)
>>> print mobj.group(1)
to python

>>> mystr = 'welcome to python My number is 1234567890 and email is abc@gmail.com.'
>>> mobj = re.search('(perl|python) (.*) (\d{10}) and email is (\w+\@\w+.(com|in))', )
>>> mobj.group()
'python My number is 1234567890 and email is abc@gmail.com'
>>> mobj.groups()
('python', ' My number is', '1234567890', 'abc@gmail.com', 'com')
>>> mobj.group(3)
'1234567890'
>>> mobj.group(4)
'abc@gmail.com'
```

Debugging & Misc functions

Executing System Commands

We can execute operating system related commands using below methods.

```
>>> import os
>>> os.system("date")     # Executes system command
Wed Dec  5 23:53:16 IST 2018
0
>>> fp = os.popen('date')  # Executes system command and capture the output
>>> for out in fp:  print out
Thu Dec  6 00:01:45 IST 2018
```

Command line arguments

We can pass values from command line to the program and capture it using sys.argv variable.

Program:

```
import sys
print "Command line args: ", sys.argv
def add(x, y):
  print "Addition: ", x + y

add(10, 20)
add(int(sys.argv[1]), int(sys.argv[2]))
```

```
manoj@HP-Compaq:~/Python/Scripts$ python command_line.py 100 200
Command line args:  ['command_line.py', '100', '200']
Addition:  30
Addition:  300
```

Command line options

Some command line options for python command are,

```
manoj@HP-Compaq:~$ python -V   [or]   python --version
Python 2.7.6

manoj@HP-Compaq:~$ python -c "import os; print os.getcwd(); print 'welcome to python'"
/home/manoj
welcome to python

manoj@HP-Compaq:~$ python -h
 <help page here>

manoj@HP-Compaq:~$ python -mpdb -h
usage: pdb.py scriptfile [arg] ...

manoj@HP-Compaq:~$ python -v <script-name>
 <verbose mode execution of script>
```

Package Installation

To install a python package we can use below commands.

$ pip -V [or] pip --version
pip 9.0.1 from /usr/local/lib/python3.4/dist-packages/pip-9.0.1-py3.4.egg (python 3.4)

$ ***pip install xlrd*** (pip install package-name)
$ ***pip uninstall xlrd***

$ ***easy_install --version***
setuptools 3.3

$ ***easy_install xlrd***

To install the package manually,
1. Download the package from https://pypi.org website and extract the zip file.
2. Go to extracted folder and we will find "***setup.py***" file inside it.
3. Execute below command to install the package.
   $ python setup.py install

Python Debugger

Python debugger is used to debug the python code line by line.

```
manoj@HP-Compaq:~/Python/Scripts$ python -m pdb command_line.py 10 15
> /home/manoj/Python/Scripts/command_line.py(3)<module>()
-> import sys

(Pdb) n
> /home/manoj/Python/Scripts/command_line.py(4)<module>()
-> print "Command line args: ", sys.argv
(Pdb) n
Command line args:  ['command_line.py', '10', '15']
> /home/manoj/Python/Scripts/command_line.py(6)<module>()
-> def add(x, y):

(Pdb) n
> /home/manoj/Python/Scripts/command_line.py(9)<module>()
-> add(10, 20)
(Pdb) n
Addition:  30
> /home/manoj/Python/Scripts/command_line.py(10)<module>()
-> add(int(sys.argv[1]), int(sys.argv[2]))

(Pdb) n
Addition:  25
--Return--
```

Shebang Line

Shebang line is used to specify which interpreter needs to be used to execute the code.

It should be at the first line of the code and should starts with “#!”.

Example:

$ cat command_line.py

```
#!/usr/bin/python
import sys
print "Command line args: ", sys.argv

def add(x, y):
  print "Addition: ", x + y

add(10, 20)
add(int(sys.argv[1]), int(sys.argv[2]))
```

Signal Handling

We can handle signals in python as below:

$ cat signal.py

```
import sys, time
import signal

def signalHandler(a,b):
    print "Signal received ",a, "==", b

signal.signal(signal.SIGINT,signalHandler);
signal.signal(signal.SIGUSR1,signalHandler);

while(1):
    print "Loop Wait..."
    time.sleep(10)
```

manoj@HP-Compaq:~/Python$ ***python signal.py***

```
Loop Wait...
^CSignal received  2 == <frame object at 0x7fc095f28050>
```