Python

Topics:

Introduction

Indentation

Numbers

strings

lists

tuples

dictionaries

loops

if

while

for

functions

modules

files

Python is a high level, interpreted and object oriented scripting language

Python was developed in 1990 by Guido Van Rossum, who works at google.

Python is quick, frictionless and nice language for automation.

Python is Interpreted

Python is processed at runtime by the interpreter. Similar to PHP and PERL, we don’t have to compile python program before executing it

Python is Interactive

Using python prompt we can directly write programs.

Python is Object-Oriented

Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is 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.

History:

Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python's feature highlights include:

Easy-to-learn: Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

Easy-to-read: Python code is more clearly defined and visible to the eyes.

Easy-to-maintain: Python's source code is fairly easy-to-maintain.

A broad standard library: Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

Interactive Mode: Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

Extendable: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

Databases: Python provides interfaces to all major commercial databases.

GUI Programming: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

Scalable: Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

It supports functional and structured programming methods as well as OOP.

It can be used as a scripting language or can be compiled to byte-code for building large applications.

It provides very high-level dynamic data types and supports dynamic type checking.

IT supports automatic garbage collection.

It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python in case sensitive a != A

The most up-to-date and current source code, binaries, documentation, news, etc. is available at the official website of Python:

Python Official Website : http://www.python.org/

Python documentation can be downloaded from the following site. The documentation is available in HTML, PDF, and PostScript formats.

Python Documentation Website : www.python.org/doc/

Interactive Mode Programming:

Invoking the interpreter without passing a script file as a parameter brings up the following prompt:

```
root# python

Python 2.5 (r25:51908, Nov  6 2007, 16:54:01)

[GCC 4.1.2 20070925 (Red Hat 4.1.2-27)] on linux2

Type "help", "copyright", "credits" or "license" for more info.
```

>>>

Type the following text to the right of the Python prompt and press the Enter key:

>>> print "Hello, Python!";

This will produce following result:

Hello, Python!

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 following identifier naming convention for Python:

Class names start with an uppercase letter and all other identifiers with a lowercase letter.

Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private.

Starting an identifier with two leading underscores indicates a strongly private identifier.

If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words:

The following list shows the reserved words in Python. These reserved words may not be used as constant or variable or any other identifier names.

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

Lines and Indentation:

One of the first caveats programmers encounter when learning Python is the fact that there are no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Both blocks in this example are fine:

```
if True:
    print "True"
else:
  print "False"
```

However, the second block in this example will generate an error:

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
  print "False"
```

Multi-Line Statements:

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example:

```
total = item_one + \
        item_two + \
```

item_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example:

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation in Python:

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes can be used to span the string across multiple lines. For example, all the following are legal:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python:

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment, and the Python interpreter ignores them.

```
#!/usr/bin/python3
# First comment
print "Hello, Python!";  # second comment
```

This will produce following result:

```
Hello, Python!
```

A comment may be on the same line after a statement or expression:

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows:

This is a comment.

This is a comment, too.

This is a comment, too.

I said that already.

Variables

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

First program:

[root@reviewb var]# cat helloworld.py

#!/usr/bin/python

print "Hello World";

[root@reviewb var]# ./helloworld.py

Hello World

[root@reviewb var]#

Getting variables from keyboard using input:

[root@reviewb var]# python

Python 2.7.5 (default, Aug 2 2016, 04:20:16)

```
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Enter a number"
Enter a number
>>> x=input()
10
>>> x
10
>>> x=input()
"10"
>>> x
'10'
>>> type(x)
<type 'str'>
>>> x=input()
10
>>> x
10
>>> type(x)
<type 'int'>
>>>
>>> x=10.90
>>> type(x)
<type 'float'>
>>>
```

Note:

User does not have to specify the data type of the variable; python internals stores as per the value assigned

```
[root@reviewb var]# cat variables.py
#!/usr/bin/python
x=input("Enter a value");
y=input("Enter another value");
print x;
print y;
```

```
[root@reviewb var]# ./variables.py
Enter a value10
Enter another value11
10
11
[root@reviewb var]#
```

Getting variables from keyboard using raw_input:

Here the type Is always string irrespective of quotes we give as input

```
[root@reviewb var]# python
Python 2.7.5 (default, Aug  2 2016, 04:20:16)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> x=raw_input("Enter data")
Enter data10
>>> x
'10'
>>> type(x)
<type 'str'>
>>> x=raw_input("Enter data")
Enter data"str"
>>> type(x)
<type 'str'>
>>> x=raw_input("Enter data")
Enter data'str'
>>> type(x)
<type 'str'>
>>>
```

Standard Data Types

Numbers

Strings

List

Tuple

Directory

Numbers:

Number data types store numeric values. Number objects are created when you assign a value to them. For example -

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is -

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example -

del var

del var_a, var_b

Python supports four different numerical types -

int (signed integers)

float (floating point real values)

complex (complex numbers)

Strings:

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. 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 and working their way from -1 at the end.

Strings are immutable [read only]

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example -

```
#!/usr/bin/python3

str = 'Hello World!'

print (str)          # Prints complete string

print (str[0])       # Prints first character of the string

print (str[2:5])     # Prints characters starting from 3rd to 5th

print (str[2:])      # Prints string starting from 3rd character

print (str * 2)      # Prints string two times
```

```
print (str + "TEST") # Prints concatenated string
```

This will produce the following result -

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Lists:

Lists are the most versatile of Python's compound data types. 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.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example -

Lists are mutable

```
#!/usr/bin/python3

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:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

This produce the following result -

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Tuples:

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

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. For example -

Tuples are immutable, ready onlye

```
#!/usr/bin/python3

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print (tuple)           # Prints complete tuple
```

```
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produce the following result -

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists -

```
#!/usr/bin/python3

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Dictionaries:

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and 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 ([]). For example -

```
#!/usr/bin/python3

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
```

This produce the following result -

```
This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Note:

More than one entry per key is not valid, no duplicate key is allowed, key has to be unique across the dictionary

Keys are immutable. Strings, tuples, numbers can be used as dictionary keys but not lists []

```
[root@reviewb python]# cat dict.py
#!/usr/bin/python
#dictionary

dict={"name": "john", "age" : 25, "state": "Austin"}
print ("dict['name']: --> ", dict['name'])
print ("dict['age']: --> ", dict['age'])
print len(dict)
print str(dict)
[root@reviewb python]# ./dict.py
("dict['name']: --> ", 'john')
("dict['age']: --> ", 25)
3
{'age': 25, 'name': 'john', 'state': 'Austin'}
[root@reviewb python]#

>>> dict={"name": "john", "age" : 25, "state": "Austin"}
>>> dict.has_key("name")
True
>>> dict.has_key("name1")
False
>>> dict.items
<built-in method items of dict object at 0x2092560>
>>> dict.items()
[('age', 25), ('name', 'john'), ('state', 'Austin')]
>>>
```

Operators

Python language supports the following types of operators.

Arithmetic Operators

Comparison (Relational) Operators

Assignment Operators

Logical Operators

Bitwise Operators

Membership Operators

Identity Operators

Arithematic:

+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1

% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Sample program 1 – prints hello on screen

```
#!/usr/bin/python
#Defining a function
def main():
        print “Hello”

#Standard boiler plate that calls main function
if __name__ == '__main__':
        main()
```

Sample program 2 – to print command line args

```
#!/usr/bin/python
import sys
#Defining a function
def main():
        print sys.argv

#Standard boiler plate that calls main function
if __name__ == '__main__':
        main()

[root@reviewb var]# ./1.py a ab c
['./1.py', 'a', 'ab', 'c']
[root@reviewb var]#

[root@reviewb var]# python
Python 2.7.5 (default, Aug  2 2016, 04:20:16)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_debugmallocstats', '_getframe', '_mercurial', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright',
'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
'exec_prefix', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace', 'hexversion',
'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
```

'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'py3kwarning', 'pydebug', 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info', 'warnoptions']

>>>

>>> help(sys)

Help on built-in module sys:

NAME

sys

FILE

(built-in)

MODULE DOCS

http://docs.python.org/library/sys

DESCRIPTION

This module provides access to some objects used or maintained by the

interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known

path -- module search path; path[0] is the script directory, else ''

modules -- dictionary of loaded modules

```
>>> help(sys.exit)
```

```
Help on built-in function exit in module sys:

exit(...)
    exit([status])

    Exit the interpreter by raising SystemExit(status).
    If the status is omitted or None, it defaults to zero (i.e., success).
    If the status is numeric, it will be used as the system exit status.
    If it is another kind of object, it will be printed and the system
    exit status will be one (i.e., failure).
```

```
>>>
>>> len("Hello")
5
>>> len
<built-in function len>
>>>
```

Type in google

Python sys exit

Sample progam 3:

```
[root@reviewb var]# cat 3.py
#!/usr/bin/python
import sys
def Hello(name):
        if name == 'john':
                name = name + '!!!!'
        else:
                Doesnotexists(name)
        print 'Hello', name
def main():
        Hello(sys.argv[1])
if __name__ == '__main__':
        main()

[root@reviewb var]#

[root@reviewb var]# ./3.py john
Hello john!!!!
```

There is a function Doesnotexists in the program which is not defined but it does not throw error if we provide the username john

Because it did not hit john

Python checks the line only when it runs

Python string type is enclosed in ‘ ’ or “ “

A = “ this is a \”exercice\””

Strings in python are immutable.

a = “Hello”

a.lower() hello

```
>>> a = "Hello"
>>> a.lower()
'hello'
>>>
>>> a
'Hello'
>>>
```

Here original a is unchanced

But it created a new string hello

```
>>> a
'Hello'
>>> a.find('e')
1
```

```
>>>



>>> a[0]
'H'
>>> a[1]
'e'
>>> a[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>




>>> 'Hi %s I have %d rupees ' % ('john', 42)
'Hi john I have 42 rupees '
>>>



>>> a
'Hello'
>>> a[1:3]
'el'
>>>
```

Starting at 1 and can go upto3 but not including 3

Python slicing

```
>>> a[1:]
'ello'
>>> a[:5]
'Hello'
>>>
```

-1 refers to right most character

```
>>> a[-1]
'o'
>>>
```

```
>>> a[:-3]
'He'
```

To omit last 3 characters

Time module:

```
>>> import time

>>> time.time()

1481415067.432948

>>> time.localtime()

time.struct_time(tm_year=2016, tm_mon=12, tm_mday=11, tm_hour=5,
tm_min=41, tm_sec=36, tm_wday=6, tm_yday=346, tm_isdst=0)

>>>

>>> localtime = time.asctime( time.localtime(time.time()) )

>>> print ("Local current time :", localtime)

('Local current time :', 'Sun Dec 11 05:43:02 2016')

>>>

>>> time.sleep(10)
```

Sleeps the script for 10 sec

Loops:

If

While

For

Nested

If:

```
[root@reviewb python]# cat if.py

#!/usr/bin/python

print "Enter a number"

x=input()

if x>0:

        print "x is positive"
```

```
elif x<0:
        print "x is negative"
else:
        print "x is zero"
[root@reviewb python]# ./if.py
Enter a number
-2
x is negative
[root@reviewb python]#
```

While:

```
[root@reviewb python]# cat while.py
#!/usr/bin/python
#while
i=0
while i<100:
        print i
        i=i+1
```

prints 0 to 100 numbers

For:

```
[root@reviewb python]# cat for2.sh
#!/usr/bin/python
for i in range(1,6):
        for j in range(1,6):
                print i,
```

```
        print "\n"
```

```
[root@reviewb python]# ./for2.sh
1 1 1 1 1


2 2 2 2 2


3 3 3 3 3


4 4 4 4 4


5 5 5 5 5
```

Loop control statements:

break statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

```
[root@reviewb python]# cat break.py
#!/usr/bin/python
i=0
```

```
while i<100:
        i=i+1
        if i==50:
                break
        print i

[root@reviewb python]# cat continue.py
#!/usr/bin/python
i=0
while i<100:
        i=i+1
        if i==50:
                continue
        print i

[root@reviewb python]# cat pass.py
#!/usr/bin/python
i=0
while i<10:
        i=i+1
        if i==5:
                pass
                print "inside pass"
        print i

[root@reviewb python]#
```

Functions:

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

Function blocks begin with the keyword def followed by the function name and parentheses (( )).

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or docstring.

The code block within every function starts with a colon (:) and is indented.

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

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
[root@reviewb python]# cat function.py
#!/usr/bin/python
def function(str):
        print str
function("Hello")
[root@reviewb python]# ./function.py
Hello
[root@reviewb python]#
```

Call by reference vs value:

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects in the calling function.

```
[root@reviewb python]# cat function2.py
#!/usr/bin/python
#function2

def function( list ):
        print ("values inside the function before change :", list)
        list[2]=100
        print ("values inside the function after change :", list)
        return
list = [10,20,30]
function( list )
print ("values outside the function after change :", list)

[root@reviewb python]# ./function2.py
('values inside the function before change :', [10, 20, 30])
('values inside the function after change :', [10, 20, 100])
('values outside the function after change :', [10, 20, 100])
[root@reviewb python]#
```

Function Arguments:

You can call a function by using the following types of formal arguments:

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
[root@reviewb python]# cat function.py
#!/usr/bin/python
def function(str):
        print str
function("Hello")
[root@reviewb python]# ./function.py
Hello
```

```
[root@reviewb python]# cat function.py
#!/usr/bin/python
def function(str):
        print str
function()
[root@reviewb python]# ./function.py
Traceback (most recent call last):
  File "./function.py", line 4, in <module>
    function()
TypeError: function() takes exactly 1 argument (0 given)
[root@reviewb python]#
```

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
[root@reviewb python]# cat function.py
#!/usr/bin/python
def function(str):
        print str
function(str="string")
[root@reviewb python]# ./function.py
string
[root@reviewb python]#
```

```
[root@reviewb python]# cat function3.py
#!/usr/bin/python
#function
def function( name, age ):
        print ("Name: ", name)
        print ("Age: ", age)
        return
function( age=40, name="john" )
[root@reviewb python]# ./function3.py
('Name: ', 'john')
('Age: ', 40)
[root@reviewb python]#
```

Default arguments:

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
[root@reviewb python]# cat function3.py
#!/usr/bin/python
#function
def function( name="default", age=40 ):
        print ("Name: ", name)
        print ("Age: ", age)
        return
function( age=40, name="hello" )
function( name="hi" )
function( age=10 )
[root@reviewb python]# ./function3.py
('Name: ', 'hello')
('Age: ', 40)
('Name: ', 'hi')
('Age: ', 40)
('Name: ', 'default')
('Age: ', 10)
[root@reviewb python]#
```

Variable length arguments:

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
[root@reviewb python]# cat function4.py
#!/usr/bin/python
#function
def function( arg1, *vartuple ):
        print "Output is ..."
        print (arg1)
        for var in vartuple:
```

```
            print var
        return
function( 10 )
function( 70,60,50 )
[root@reviewb python]# ./function4.py
Output is ...
10
Output is ...
70
60
50
[root@reviewb python]#
```

Lambda functions/The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires an expression

Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of lambda functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how lambda form of function works −

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2



# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
print ("Value of total : ", sum( 20, 20 ))
```

When the above code is executed, it produces the following result −

```
Value of total :  30
Value of total :  40
```

```
[root@reviewb python]# cat lambda.py
#!/usr/bin/python
sum = lambda arg1,arg2: arg1+arg2
print ("value of total: ", sum(10,20))
print ("value of total: ", sum(100,20))
[root@reviewb python]# ./lambda.py
```

('value of total: ', 30)

('value of total: ', 120)

[root@reviewb python]#

The return Statement

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

```
#!/usr/bin/python

def sum( a, b ):
        total = a+b;
        print "inside the function", total
        return total
total = sum(10,20)
print "outside the function", total
```

[root@reviewb python]# ./return.py

inside the function 30

outside the function 30

[root@reviewb python]#

Scope of variables in function:

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

Global variables

Local variables

Local vs Global scope:

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

This means that 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. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
[root@reviewb python]# cat scope.py
#!/usr/bin/python
#scope
total = 0 # global variable
def sum( a, b):
        total = a+b # here total is local variable
        print "Inside the function local", total
        return total
sum(10, 40)
print "Outside the function Global", total
[root@reviewb python]# ./scope.py
Inside the function local 50
Outside the function Global 0
[root@reviewb python]#
```

Modules:

A module allows to logically organize Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Any python source file can be used as module by using import statement

Import module1

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax –

From module1 import hi imports hi function only

From module1 import hello imports hello function only

From module1 import * imports all functions

Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences −

The current directory.

If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.

If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python3/.

The module search path is stored in the system module sys as the sys.path variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The PYTHONPATH Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

set PYTHONPATH=c:\python34\lib;

And here is a typical PYTHONPATH from a UNIX system:

set PYTHONPATH=/usr/local/lib/python

reload() – to re execute the module

Files

The open Function

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a file object, which would be utilized to call other support methods associated with it.

file object = open(file_name [, access_mode][, buffering])

Here are parameter details:

file_name: The file_name argument is a string value that contains the name of the file that you want to access.

access_mode: The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

buffering: If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).

different modes of opening a file -

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

ab+ Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

R opens for reading cursor is at beginning

R+ opens for reading and writing, cursor is placed at the beginning of file

W opens for writing only, overwrites if file is there

W+ opens for reading and writing , overwrites the file

A opens for appending, cursor at the end

A+ both appending and reading , cursor at the end

```
#!/usr/bin/python

# Open a file

fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

print ("Closed or not : ", fo.closed)

print ("Opening mode : ", fo.mode)

fo.close()
```

```
[root@reviewb ~]# python
Python 2.7.5 (default, Aug  2 2016, 04:20:16)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> file=open('test','w')
>>> file.write("This is my first line\n")
>>> file.write("Thisis my second line\n"
... )
>>> file.close()
>>>
[root@reviewb ~]# cat test
This is my first line
Thisis my second line
[root@reviewb ~]#
```

```
>>> file=open('test')
>>> file.read()
'This is my first line\nThisis my second line\n'
>>>


>>> file=open('test')
>>> print (file.read())
This is my first line
Thisis my second line

>>>


>>> file=open('test')
>>> file.readline()
'This is my first line\n'
>>> file.readline()
'Thisis my second line\n'
>>> file.readline()
''
>>>
```

Creating connection between file handle and file

Opening file

Closing file

Readling file

Writing file

```
>>> file.close()
>>> file=open('test3','w')
>>> file.write("This is my first line\n")
>>> file.close()
>>> file.open('test3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'file' object has no attribute 'open'
>>> file=open('test3')
>>> file.read(5)
'This '
>>> file.tell()
5
>>> file.read(5)
'is my'
>>> file.seek(0, 0)
>>> file.read(5)
'This '
>>>
```

File.read – to read a file , prints raw output

File,readline – to read line by line

File.write

File.tell

File.seek

```
>>> file=open('test3','r')
>>> for line in file:
...     print line
...     print "\n"
...
This is my second lineThis is my second lineThis is my second line
```

To read line by line

File=open(“file”,’r’)

For line in file:

Print line

Splitting the file using the delimeter:

```
>>> file=open("testfile",'r')
>>> for line in file:
...     words=line.split(" ")
...     print words
...
['This', 'is', 'my', 'test', 'file', 'for', 'exception', 'handling!!HelloHiiii\n']
['123abc']
>>>
```

Appending toa file

```
>>> file=open("test123",'a')
>>> linesoftext=['a','b','c']
>>> file.writelines(linesoftext)
>>> file.close()
>>>
```

Exceptions:

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax:

```
try:
   You do your operations here
```

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.

Note:

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all –

[root@reviewb python]# cat exception.py

#!/usr/bin/python

try:

fh = open("test", "w")

fh.write("This is my test file for exception handling!!")

except IOError:

```
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
[root@reviewb python]# ./exception.py
Written content in the file successfully
[root@reviewb python]#
```

```
[root@reviewb python]# cat exception1.py
#!/usr/bin/python
try:
   fh = open("test", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print ("Error: can\'t find file or read data")
else:
   print ("Written content in the file successfully")
   fh.close()
[root@reviewb python]# ./exception1.py
Error: can't find file or read data
[root@reviewb python]#
```

In the above example, we are trying to write to a file, where we have opened with read permission hence it throws exception.

Try-Finally Clause:

We can use finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this -

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

```
[root@reviewb python]# cat exception3.py
#!/usr/bin/python

try:
   fh = open("testfile123", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print ("Error: can\'t find file or read data")
   fh.close()
[root@reviewb python]# ./exception3.py
Error: can't find file or read data
[root@reviewb python]# cat testfile123
This is my test file for exception handling!![root@reviewb python]#
```

Here the finally blocks executed even if there is no exception thrown by exception block.

```
[root@reviewb python]# cat exception4.py
#!/usr/bin/python


try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
[root@reviewb python]# ./exception4.py
Going to close the file
[root@reviewb python]#
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

Next prog:

Int(var) will check if var is int or not:

```
[root@reviewb python]# python
```

```
Python 2.7.5 (default, Aug  2 2016, 04:20:16)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> var="xyz"
>>> int(var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'xyz'
>>> var="str"
>>> str(var)
'str'
```

A program to check if provided input is integer or not:

```
[root@reviewb python]# cat exception5.py
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError:
      print ("The argument does not contain numbers ", var)

# Call above function here.
temp_convert("xyz")
#temp_convert(123)
[root@reviewb python]# ./exception5.py
```

('The argument does not contain numbers ', 'xyz')

[root@reviewb python]#

Raise exception:

We can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

Syntax

raise [Exception [, args [, traceback]]]

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows -

```
def functionName( level ):
   if level <1:
      raise Exception(level)
      # The code below to this would not be executed
      # if we raise the exception
   return level
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows -

```
try:
   Business Logic here...
except Exception as e:
   Exception handling here using e.args...
else:
   Rest of the code here...
```

```
[root@reviewb python]# cat exception6.py
#!/usr/bin/python
def functionName( level ):
    if level <1:
        raise Exception(level)
        # The code below to this would not be executed
        # if we raise the exception
    return level

try:
    l=functionName(-10)
    #l=functionName(1)
    print ("level=",l)
except Exception as e:
```

```
    print ("error in level argument",e.args[0])
[root@reviewb python]# ./exception6.py
('error in level argument', -10)
[root@reviewb python]#
```

A program that explains exception try finally clauses clearly

```
[root@reviewb python]# cat exception7.py
#!/usr/bin/python
def divide(x,y):
        try:
                result = x/y
        except ZeroDivisionError:
                print "division by zero"
        else:
                print "result is", result
        finally:
                print "executing finally clause"


[root@reviewb python]# python
Python 2.7.5 (default, Aug  2 2016, 04:20:16)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import os
>>> import sys
>>> from exception7 import *
>>> divide(2,2)
result is 1
executing finally clause
>>> divide(2,1)
result is 2
executing finally clause
>>> divide(4,2)
result is 2
executing finally clause
>>> divide(4,0)
division by zero
executing finally clause
>>>
```

User defined exceptions:

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to RuntimeError. Here, a class is created that is subclassed from RuntimeError. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class Networkerror.

```
class Networkerror(RuntimeError):
```

```
  def __init__(self, arg):
    self.args = arg
```

So once you defined above class, you can raise the exception as follows -

```
try:
  raise Networkerror("Bad hostname")
except Networkerror,e:
  print e.args
```

Object Oriented Python:

creating and instatiating a class

What is the difference between class and instance

Class is a blue print of for creating instances

why to use classes

not only python most of other languages are using classes

Classes allow us to logically group our data and function in way what is easy to reuse.

Classes contain attributes and methods.

method is a function

there is a company and we need to represent the employees in terms of class
employee attributes:

name
email
sal

let us create a simple employee class

```
class Employee:
    pass
```

class is blue print for creating instances
employee is a instances of employee class

Program 1:

```
[root@reviewb python]# cat class1.py
#!/usr/bin/python

class Employee:
        pass
emp_1 = Employee()
emp_2 = Employee()

print emp_1
print emp_2

[root@reviewb python]# ./class1.py
<__main__.Employee instance at 0x7f431bc3d248>
<__main__.Employee instance at 0x7f431bc3d200>
[root@reviewb python]#
```

both of these are emp objects, created at diff locations

instance variables contain data that is unique to instance.

Program 2:

```
[root@reviewb python]# cat class1.py
#!/usr/bin/python

class Employee:
        pass
emp_1 = Employee()
emp_2 = Employee()

print emp_1
print emp_2

emp_1.first = 'user1'
emp_1.last = 'test'
emp_1.email = 'user1.test@company.com'
emp_1.sal=100000

emp_2.first = 'user2'
emp_2.last = 'test'
emp_2.email = 'user2.test@company.com'
emp_2.sal=100000

print emp_1.email
```

```
print emp_2.email
[root@reviewb python]# ./class1.py
<__main__.Employee instance at 0x7ff221794200>
<__main__.Employee instance at 0x7ff22179a488>
user1.test@company.com
user2.test@company.com
[root@reviewb python]#
```

there is lot of manual work , and it is prone to mistakes

so we don't get much benefit this way

how to do automatically
we have to use special init method - initialize or constructor

by convention we call this instance self

Program 3:

```
[root@reviewb python]# cat class2.py
#!/usr/bin/python

class Employee:
        def __init__(self, first, last, pay):
                self.first = first
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'

emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)


print emp_1.email
print emp_2.email
[root@reviewb python]# ./class2.py
user1.test1@company.com
user2.test2@company.com
[root@reviewb python]#
```

```
[root@reviewb python]# cat class2.py
#!/usr/bin/python
```

```
class Employee:
        def __init__(self, first, last, pay):
                self.first = first
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'

emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)


print emp_1.email
print emp_2.email

print emp_1.first, emp_1.last
[root@reviewb python]# ./class2.py
user1.test1@company.com
user2.test2@company.com
user1 test1
[root@reviewb python]#
```

we can create a method full name so that it prints a full name
automatically instance takes self by default as the first argument.

```
[root@reviewb python]# cat class2.py

#!/usr/bin/python


class Employee:

        def __init__(self, first, last, pay):

                self.first = first

                self.last = last

                self.pay = pay

                self.email = first + '.' + last + '@company.com'


        def fullname(self):

                return '{} {}'.format(self.first, self.last)
```

```
emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)

print emp_1.email
print emp_2.email

#print emp_1.first, emp_1.last
print emp_1.fullname()
print Employee.fullname(emp_1)
#emp_1.fullname()

[root@reviewb python]# ./class2.py
user1.test1@company.com
user2.test2@company.com
user1 test1
user1 test1
```

Class Variables:

Instance variables are the ones which are set using the self-variable.

Example

Name

Email

Pay

Class variables are the variables that are shared among all the instances of class .

Employee class – annual raise – same for all employee

Regular methods, class methods and static methods:

Regular methods automatically takes self as the first arg.

Class method:

We can change a regular method to class method by using the decorators

@classmethod

[root@reviewb python]# cat class4.py

```
#!/usr/bin/python
#Defining the class variable
class Employee:
        raise_amount = 1.10
        def __init__(self, first, last, pay):
                self.first = first
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'

        def fullname(self):
                return '{} {}'.format(self.first, self.last)



        def apply_raise(self):
#               self.pay = int(self.pay * 1.04) # instead of giving manually here we can have variable for raise amount
                self.pay = int(self.pay * Employee.raise_amount)
```

```
emp_1 = Employee('user1', 'test1', 100000)
emp_2 = Employee('user2', 'test2', 110000)

print emp_1.pay
emp_1.apply_raise()
print emp_1.pay



#print emp_1.fullname()
#print Employee.fullname(emp_1)

[root@reviewb python]#



[root@reviewb python]# cat class5.py
#!/usr/bin/python
# instance variable

class Employee:
        raise_amount = 1.04  # class variable
        def __init__(self, first, last, pay):
                self.first = first   # instance variables
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'
```

```
    def fullname(self):
        return '{} {}'.format(self.first, self.last)



    def apply_raise(self):
#            self.pay = int(self.pay * 1.04) # instead of giving manually here we can have variable for raise amount
            self.pay = int(self.pay * self.raise_amount)



emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)

emp_1.raise_amount=1.05
emp_2.raise_amount=1.10

print Employee.raise_amount
print emp_1.raise_amount
print emp_2.raise_amount



#print emp_1.__dict__ #This is to print the name space of employee 1 , here raise_amount is instance variblae because we delcared it
#print emp_2.__dict__

[root@reviewb python]#
```

```
[root@reviewb python]# cat class6.py
#!/usr/bin/python

# Add one class variblae num_of_emps

class Employee:
        num_of_emps = 0  # class variable
        raise_amount = 1.04  # class variable
        def __init__(self, first, last, pay):
                self.first = first   # instance variables
                self.last = last # instance variables
                self.pay = pay # instance variables
                self.email = first + '.' + last + '@company.com'         # instance variables
                Employee.num_of_emps += 1

        def fullname(self):
                return '{} {}'.format(self.first, self.last)




        def apply_raise(self):
#               self.pay = int(self.pay * 1.04) # instead of giving manually here we can have variable for raise amount
                self.pay = int(self.pay * self.raise_amount)
```

```
print "Before creating users"
print Employee.num_of_emps
print "Creating users ..."
emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)
emp_3 = Employee('user3', 'test3', 150000)
emp_4 = Employee('user4', 'test4', 160000)
print "After creating users"

emp_1.raise_amount=1.05




#print Employee.raise_amount
#print emp_1.raise_amount
#print emp_2.raise_amount

print Employee.num_of_emps



#print emp_1.__dict__ #This is to print the name space of employee 1 , here raise_amount is instance variblae because we delcared it
#print emp_2.__dict__

[root@reviewb python]#
```

```
[root@reviewb python]# cat class7.py
#!/usr/bin/python


# class methods


class Employee:
        num_of_emps = 0
        raise_amt = 1.04  # class variable
        def __init__(self, first, last, pay):
                self.first = first    # instance variables
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'


                Employee.num_of_emps += 1


        def fullname(self):
                return '{} {}'.format(self.first, self.last)




        def apply_raise(self):
#                self.pay = int(self.pay * 1.04) # instead of giving manually here we can have variable for raise amount
                self.pay = int(self.pay * self.raise_amt)
        @classmethod  # Decorator
        def set_raise_amt(cls,amount):
```

```
            cls.raise_amt = amount


emp_1 = Employee('user1', 'test1', 50000)
emp_2 = Employee('user2', 'test2', 60000)
Employee.set_raise_amt(1.05)



print Employee.raise_amt
print emp_1.raise_amt
print emp_2.raise_amt

[root@reviewb python]#




[root@reviewb python]# cat class8.py
#!/usr/bin/python

# static methods

# to find out given day is working day or not

#Regular methods pass instance as first arg (self), class methods pass class as first arg, static methods dont pass anything automatically they are like normal functions

class Employee:
    num_of_emps = 0
```

```
    raise_amt = 1.04  # class variable
    def __init__(self, first, last, pay):
        self.first = first   # instance variables
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return '{} {}'.format(self.first, self.last)




    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)
    @classmethod
    def set_raise_amt(cls,amount):
        cls.raise_amt = amount

    @staticmethod
    def is_workday(day):
        if day.weekday() == 5 or day.weekday() == 6:  #weekday is a method
monday=0, sunday=6
            return False
        return True



emp_1 = Employee('user1', 'test1', 50000)
```

```
emp_2 = Employee('user2', 'test2', 60000)
import datetime
my_date = datetime.date(2016, 12, 11)
#print my_date #prints in 2016-12-11 format
print Employee.is_workday(my_date)
```

The below program explains Inheritance

```
[root@reviewb python]# cat class9.py
#!/usr/bin/python

#Inheritence

class Employee:
        num_of_emps = 0
        raise_amt = 1.04  # class variable
        def __init__(self, first, last, pay):
                self.first = first   # instance variables
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'

                Employee.num_of_emps += 1

        def fullname(self):
                return '{} {}'.format(self.first, self.last)
```

```
    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


class Developer(Employee): # inherited from parent class Employee
    pass
    raise_amt = 1.10 # if this is not specified the hike amount is taken from parent class which is 1.04

    def __init__(self, first, last, pay, prog_lang):
        #super(Developer, self).__init__(first, last, pay)
        super(Employee, self).__init__(first, last, pay)
        self.prog_lang = prog_lan



dev_1 = Developer('user1', 'test1', 100000,'python')
dev_2 = Developer('user2', 'test2', 60000,'java')

print dev_1.email
print dev_2.email

print dev_1.pay
dev_1.apply_raise()
print dev_1.pay

print dev_1.prog_lan
```

[root@reviewb python]#

The below program explains Method overloading

```
[root@reviewb python]# cat class10.py
#!/usr/bin/python
#Method overloading
class Parent:
        def mymethod(self):
                print "Calling parent method"
class Child(Parent):
        def mymethod(self):
                print "Calling child method"
#       pass # if no method is declared in child class, it prins from parent class
c = Child() # creating instance of a class
c.mymethod() # child calls overridden method

[root@reviewb python]#
```

```
[root@reviewb python]# cat class11.py
#!/usr/bin/python
```

```
#operator overloading
#teach python how do add two objects

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self): # with str function it prints the values properly
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
[root@reviewb python]#
```

The below program explains Inheritance

```
[root@reviewb python]# cat classinherit.py
#!/usr/bin/python
```

```
#Inheritence

class Employee:
    num_of_emps = 0
    raise_amt = 1.04  # class variable
    def __init__(self, first, last, pay):
        self.first = first   # instance variables
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

        Employee.num_of_emps += 1

    def fullname(self):
        return '{} {}'.format(self.first, self.last)



    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amt)


class Developer(Employee): # inherited from parent class Employee
    raise_amt = 1.10  # if we comment this line rise will be 52000
    def __init__(self, prog_lang):
        Employee.__init__(self, first, last, pay)
        self.prog_lang = prog_lan
```

```
class Manager(Employee):
        def __init__(self, employees=None):
                Employee.__init__(self, first, last, pay)
                if employees is None:
                        self.employees = []
                else:
                        self.employees = employees
        def add_emp(self,emp):
                if emp not in self.employees:
                        self.employees.append(emp)
        def remove_emp(self,emp):
                if emp  not in self.employees:
                        self.employees.remove(emp)


        def print_emp(self):
                for emp in self.employees:
                        print '-->', emp.fullname()


#dev_1 = Developer('user1', 'test1', 50000)
#dev_2 = Developer('user2', 'test2', 60000)


dev_1 = Developer('user1', 'test1', 50000, 'java')
dev_2 = Developer('user2', 'test2', 60000, 'python')


#dev_1 first looks for init method in Developer class, its not there so it gets from
Employee
```

```
#print dev_1.email
#print dev_2.email

#print dev_1.pay
#dev_1.apply_raise()
#print dev_1.pay




mgr_1 = Manager('manager1', 'test1', 90000, [dev_1])
mgr_2 = Manager('manager2', 'test2', 190000, [dev_2])
print mgr_1.email
print mgr_2.email
```

The below program explains Inheritance .

```
[root@reviewb python]# cat classinherit.py-backup
#!/usr/bin/python

#Inheritence

class Employee:
        num_of_emps = 0
        raise_amt = 1.04  # class variable
        def __init__(self, first, last, pay):
                self.first = first    # instance variables
                self.last = last
                self.pay = pay
                self.email = first + '.' + last + '@company.com'

                Employee.num_of_emps += 1

        def fullname(self):
                return '{} {}'.format(self.first, self.last)




        def apply_raise(self):
                self.pay = int(self.pay * self.raise_amt)


class Developer(Employee): # inherited from parent class Employee
        raise_amt = 1.10  # if we comment this line rise will be 52000
        def __init__(self, first, last, pay, prog_lang):
```

```
        super(Developer, self).__init__(first, last, pay)
        self.prog_lang = prog_lan



class Manager(Employee):
    def __init__(self, first, last, pay, employees=None):
        super(Manager, self).__init__(first, last, pay)
        if employees is None:
            self.employees = []
        else:
            self.employees = employees
    def add_emp(self,emp):
        if emp not in self.employees:
            self.employees.append(emp)
    def remove_emp(self,emp):
        if emp  not in self.employees:
            self.employees.remove(emp)

    def print_emp(self):
        for emp in self.employees:
            print '-->', emp.fullname()

#dev_1 = Developer('user1', 'test1', 50000)
#dev_2 = Developer('user2', 'test2', 60000)

dev_1 = Developer('user1', 'test1', 50000, 'java')
dev_2 = Developer('user2', 'test2', 60000, 'python')
```

#dev_1 first looks for init method in Developer class, its not there so it gets from Employee

#print dev_1.email

#print dev_2.email

#print dev_1.pay

#dev_1.apply_raise()

#print dev_1.pay

mgr_1 = Manager('manager1', 'test1', 90000, [dev_1])

mgr_2 = Manager('manager2', 'test2', 190000, [dev_2])

print mgr_1.email

print mgr_2.email

Modules:

Subprocess

```
  proc = subprocess.Popen(["rsh" , nodename CMD], stdout=subprocess.PIPE,
stderr=subprocess.PIPE)
  output,error = proc.communicate()
  print output
```

```
sys.path.append('modules')
sys.path.append("../..")
```

```
import os     to import the module
os.system() to use system function from module os
```