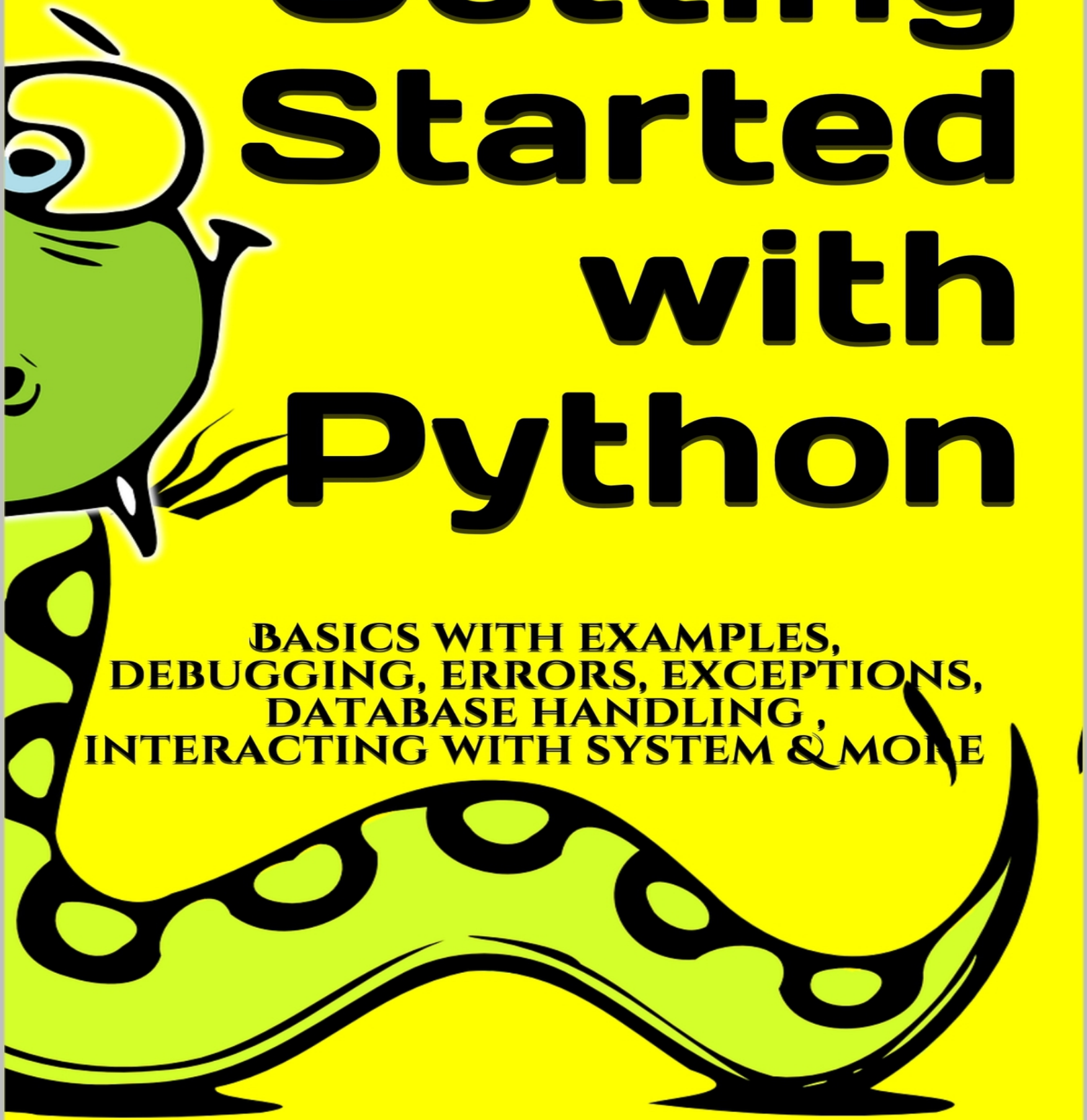
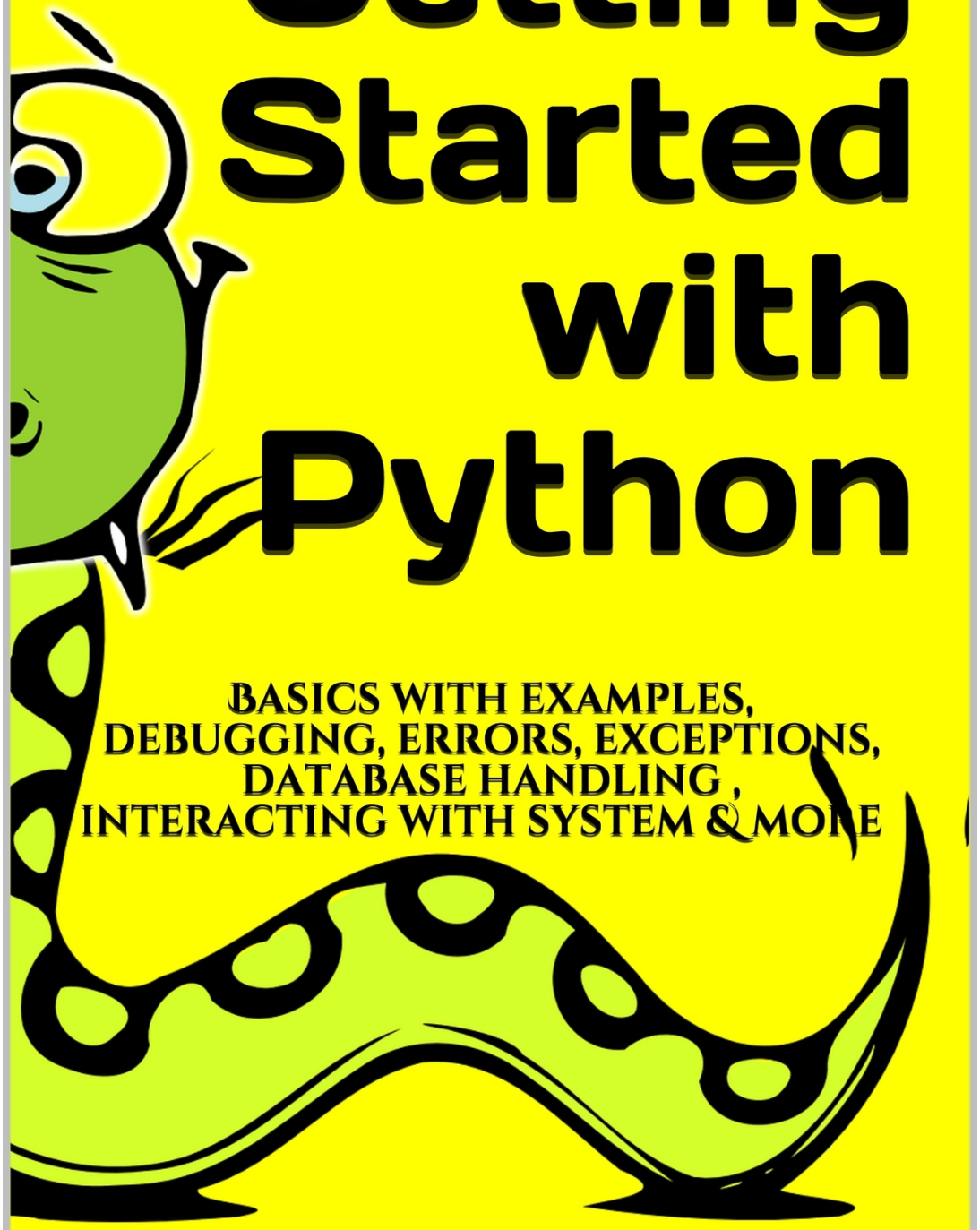
Getting Started with Python



BASICS WITH EXAMPLES, DEBUGGING, ERRORS, EXCEPTIONS, DATABASE HANDLING , INTERACTING WITH SYSTEM & MORE

Hemant Sharma

Getting Started with Python



BASICS WITH EXAMPLES,
DEBUGGING, ERRORS, EXCEPTIONS,
DATABASE HANDLING ,
INTERACTING WITH SYSTEM & MORE

Hemant Sharma

CONTENTS

PREFACE

Getting started with python is an effort to enable self learning by using small basic examples of the python building blocks. My belief is that if your foundations are good your can build many high rise building using that foundation.

Hemant Sharma

INTRODUCTION

Python is a high level general-purpose programing language which is often compared to Tcl, Perl, Java, JavaScript, Visual Basic or Scheme. Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines and used for

For real life usage Python is generally used to create in house testing and automation tools for deployment, monitoring system & environment building. There are not many commercial products created with python but it is more often used to supplement the main products created with java or c/c++ and other languages.

Python is written by Dutch computer programme, Guido Van Rossum in early 1991 with the following objectives

1. An easy and intuitive language
2. Open source
3. Easy to understand coe as plain English
4. Suitability for everyday tasks

The name python is given by Van Rossum inspired by a British sketch comedy series, Monty Python's Flying Circus, broadcast by the BBC from 1969 to 1974.

The Python Software Foundation (PSF), a non-profit corporation, holds the intellectual property rights of Python programming language. PSF manage the open source licensing for Python version 2.1 and later and own and protect the trademarks associated with Python.

PSF run the annual North American PyCon conferences, support other Python conferences around the world, and fund Python related development

FEATURES

Python comes with extensive standard library of built in modules. Some modules are written in C to support system level access such as I/O. Other modules written in python provides standard functionality like string operation using regular expression ,protocols like HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming , system calls , TCP/IP sockets . A number of third party modules are also available to supplement existing python modules.

RELEASES

There are two current versions of Python, 2.x and 3.x, Python 2.7.15 is the latest version for 2.x and python 3.6.5 is the latest version for 3.x as of now. 3.x version is cleaned up and enhanced version of python 2.x and does not have backward compatibility due to complete overhauling of 2.x version.

TOP 10 KEY DIFFERENCES BETWEEN 2.X AND 3X VERSIONS

1. Print command is statement in 2.x and a function in 3.x
2. The integer division is classical division in 2.x which truncates the decimal but true division in 3.x always returning the result with decimals.
3. 2.x has ASCII str() type and unicode function but no byte types; 3.x has Unicode(utf8).
4. xrange () function in 2.x returns a list to use in for loop like control structure, it is replaced by range() in 3.x which returns a iteratable structure.
5. Exception Arguments: 3.x requires exception arguments to be enclosed with parentheses while 2.x can accept a quoted notation.

6. Exception handling in 3.x requires keyword as to catch the standard exception in user defined variable and no ‘as’ keyword is required in 2.x
7. 3.x has only next() function to retrieve the next item from the list whereas 2.x can use next() or .next() method.
8. 3.x prevents loop variable and global variables mix, in 2.x variable can get mixed if the same variable is being used globally and in a for loop also.
9. 3.x rounds of decimal to nearest even number
10. input() function stores input as string in 3.x , user input can be other types also in 2.x and we raw_input() function is need to capture.

DOWNSIDE OF USING 3.X

python 3.x is not backward compatible with 2.x and much of the 2.x software doesn’t work on 3.x version yet. Features which are only available in 3.x releases will not be back ported to the 2.x series.

CHOOSING BETWEEN PYTHON 2 & PYTHON 3

Few things to consider when deciding a version for your use

1. If you have lot of python code already in use most likely it is in 2.x version and you should choose 2.x as 3.x is not backward compatible.
2. Many third party packages may not support 3.x so if you plan to use third party packages check if support for 3.x is available or you can port it easily to 3.x
3. Most of the Linux and other distributions come pre-installed with versions which are a few releases behind and your distribution may still have 2.x as python distribution.

CODE CONVERSION TOOLS

- 2to3 tool allows 3.x code to be generated from 2.x source code,
- 3to2 tool, allows 3.x code to convert back to 2.x code. In theory,

These tools are good if code doesn’t make use of 3.x features extensively.
Since there is no backward compatibility code will not convert successfully

1. INSTALLATION

python installation is simple and straight forwards but if you are working on a shared or company provided computer, it worthwhile to check if it is already installed. It can save some time in getting started with python

CHECKING FOR EXISTING INSTALLATION IN UNIX/LINUX, WINDOWS OR MAC

Open a terminal window for command prompt and type python. If python is already installed then an interactive python shell with details of version if already installed

> C:\> python
>
> Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec 5 2015, 20:32:19) [MSC v.1500 32 bit (Intel)] on win32
>
> Type "help", "copyright", "credits" or "license" for more information.
>
> $python
>
> Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)] on win32
>
> Type "copyright", "credits" or "license()" for more information.
>
> However if python is installed in a nonstandard location, you may have to go to that location to see what version is installed.

From python shell you can get help on a module, keyword, or topic by typing help() , which opens a help subshell , where you can type topics or keywords to get help , similar to man pages in Unix/Linux

```
>>> help ()   ### Launches help subshell to run help commands
help>
```

DOWNLOAD PYTHON

If python is not installed or you want to install a different version you can download latest Python binary or source distribution from python.org, at http://www.python.org/download/

You can download 2.7.x or 3.x version depending on what you decide based on your requirements.

INSTALLATION ON DIFFERENT PLATFORMS

WINDOWS

Download and extract zip arrive in a directory, go to command prompt and run following command

```
python setup.py install
```

UNIX /LINUX

Download source code in a directory and run following from the command prompt

```
#./configure
#make
#make install
```

README file in the root of the Python source has details of configuration options and caveats for specific Unix platforms.

make install can overwrite or masquerade existing python binary. make altinstall is therefore recommended instead of make install since it only installs exec_prefix/bin/pythonversion.

MAC INSTALLATION

Python comes pre-installed on Mac OS X, but often it is old version and best option is to upgrade by downloading and installing a newer version from python download location. You can choose between source code and 32/64 installer versions.

WORKING IN PYTHON SHELL

python command in the installation directory invokes the python shell and you can run command and learn all the functions inside the python shell. The shell is excited by pressing control C which clears all variables, imports etc. from the shell memory.

Throughout this book the example are run in the python shell.

PYTHON PROGRAM FILE

Python program files have extension as .py . You can create use any text editor to create python program file and save it with .py extension.

The first line is always used to define location of python as

#!/usr/local/bin/python in Linux and unix environments
This is particularly useful when you have multiple installations and you want to use a specific version.

something like

#!/usr/local/bin/python27

2. ENVIRONMENT

PATHS AND FILES IN PYTHON

The deployment process creates an installation directory structure to keep different types of files.

The top level directory location depends on the prefix (${prefix}) and exec_prefix (${exec_prefix}) supplied at the time of building and compiling python however if you don't provide specific options it is deployed under /usr directory in Linux and Unix. Default is c:\ for windows and

DIRECTORY STRUCTURE

Most subdirectories have their own README files and many text files have comments to describe contents.

Following is a typical python directory structure in Unix/Linux

- /usr/bin/python - location of python interpreter.
- /usr/lib/include - directories containing the standard modules.
- /usr/include/lib - directories containing the include files needed for developing Python extensions and embedding the interpreter.
- ~/.pythonrc.py - This is user-specific initialization file loaded by the user module; not used by default or by most applications.

A windows installation has following installation directory structure

- DLLs/
- Doc/
- Include/

- Lib/ - Module library with files in the form of .py files.
- Libs/ - Static Library files in the form of .lib
- Tcl/
- Tools/

ENVIRONMENT VARIABLES

Python environmental variables determine where to look for python libraries, modules and setting of some command line switches. These variables can be set on the command line or environment file in Unix/Linux

PYTHONHOME : Directory to look for python libraries, Default is /usr/local in Unix/Linux but can be changed to other location using this variable

PYTHONPATH: Default search path for python module files with similar format PATH variable. Search path can be changed from inside a program using variable sys.path

PYTHONSTARTUP: Specifies a file with startup commands. These commands are executed before the first prompt is displayed in interactive mode.

PYTHONY2K A non-empty string results in 4-digit years for time module.

Following variables if set to non-empty string becomes equivalent to specifying an command line option, helping in creating a custom environment for testing & debugging

PYTHONOPTIMIZE -O option
PYTHONDEBUG -d option
PYTHONUNBUFFERED -u option.
PYTHONVERBOSE -v option
PYTHONCASEOK ignores case in import
PYTHONDONTWRITEBYTECODE -B option
PYTHONHASHSEED If set to random, the -R option

PYTHONINSPECT -i option

These variable can also be modified by Python code using os.environ to force inspect mode on program termination.

INTERPRETER LOCATION IN PROGRAM FILE

When writing a program under Unix/Linux you have to mention the interpreter for the program on the top at first line of the program.

If you have multiple python installations and you want to use a specific version, you can define the location of your python version here

```
#!/usr/bin/python
```

ENCODING

Without encoding comment, Python's parser will assume ASCII. To define a source code encoding, a magic comment must be placed into the source files either as first or second line in the file, such as: # -*- coding: <encoding-name> -*-

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

3. DATA TYPES

Variables are declared to reserve the memory space. However you don't need to declare them explicitly in python as they are declared upon assignment.

STANDARD DATA TYPES

Python has five standard data types to store data in them.

- **Numbers** : A numeric
- **String** : String of letters
- **List** : A list of values stored in an indexed array
- **Tuple** : Number of values separated by comma 123,456,'abcd'
- **Dictionary** : A array storing keys and value pairs fruit=>apple;toy=>doll

VARIABLE ASSIGNMENT

Variables are assigned by using an equal sign (=) by putting name of variable on the left side and the values on the right side
For example

```
#!/usr/bin/python
cows = 10      # An integer assignment
length  = 100.8   # A floating point
city = "San Francisco"   # A string
```

MULTIPLE VARIABLE ASSIGNMENT

Python allows you to assign single values to a number of variables simultaneously. This can be a same value for all or different value for different variables.

```
a = b = c = 100       ## Assign same value to all variables
a, b, c = 10, 20, "Red" ##Assign 10 to a , 20 to b and Red to variable
c
```

SINGLE QUOTE

Single quotes are used for enclosing string literal which is printed as such and variables are not substituted with values. However special characters such as new lines tab /n /t are interpolated in single quotes.

Eggs = ‘One`

Eggs = ‘One and Two`

Eggs = ‘One Two and \t Three`

Eggs = ‘One Two and \n Three`

On printing these variables , first two examples prints the strings as such while the third one inserts a tab between the word “and Three”, a new line is inserted in the fourth example between the word “and Three”

DOUBLE QUOTE

Double quotes are used to preserve while space and variable substitution. Special characters such quote character has to be escaped with backslash (\) character

Eggs = “One \“Two\” and Three”

TRIPLE QUOTE

Python offers an easy way to escape long lines with multiple quotes and special characters. Enclosing long strings with triple string allows retaining special characters and quotes. The triple quote cans be single or double quotes.

String with white space, variables, new lines, Quotes

```
msg = '''Today is \t "Monday" "\nwhat is tomorrow " \n "Tuesday" '?'
'''
```

RAW STRING - THE STRING LITERALS PREFIXES

A long string with lots of special characters like quotes, slash etc. can simply be declared as raw string to avoid escaping each character individually

letter **'r'** **or** **'R'** called raw strings and this causes python not to interpreting special backslash sequences such as newline , tabs inside a string. This however protects the end of string backslash and new line characters. Line=r'some "random" text \n line 2 ' ## double quotes and \n are not interpreted as special characters.

STRING MANIPULATIONS

Strings are stored in indexed array and you can print the individual letters, a range from starting or backwards

Two strings can be combined with each other using single or double quotes and printed using positional reference starting from zero, for example :

word = 'HELLO'+'NEWYORK'

H	**E**	**L**	**L**	**O**	**N**	**E**	**W**	**Y**	**O**	**R**	**K**
0	1	2	3	4	5	6	7	8	9	10	11
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

a.) To print a character at particular position

```
>>>print word[4]    ## print 5th position , starts at 0
'L'
```

b.) Print a range

```
>>> word[0:2]    ## print first 2 position , starts at 0
'HE'
```

If first range is omitted first index defaults to zero above is same as

```
>>> word[:2]    ## print first 2 position , starts at 0
'HE'
```

```
>>> word[2:]    ## print last 2 position
'RK'
```

c.) Slice Operations

You can combine two slices at any point

```
>>> word[:5] + word[5:8]
'HELLONEW
```

4. OPERATORS

Once variables are defined you can use operators to do mathematical calculation, compare or test for true/false of bitwise testing. Below are the types of operators which are available in python.

MATH

Below are math functions in python with operation, usage and result for assigned values of two variables assigned as x=20, y=6,

The commands can be used directly in a python shell or used with print statement, after assigning variables inside a program.

Operation	Operator	Command	Results
Add	+	x + y	26
Subtract	-	x – y	14
Multiply	*	x * y	120
Divide (integer)	/	x / y	3
Remainder	%	x % y	2
Negate	-	-x	-20
Unchanged	+	+x	20
Absolute Value	abs()	abs(x)	20
Convert to Integer	Int()	int(x)	20
Convert to long integer	long()	long(x)	20L
converted to floating point	float()	float(x)	20.0
Div & remainder	divmod()	divmod(x,y)	(3, 2)
Raise to the power	pow()	pow(x, y)	64000000
Raise to the power	**	x ** y	64000000

COMPARISON OPERATOR

Comparison operations are used to compare values in numeric and string operations.

Multiple comparisons can be done simultaneously

a > b > c > d

The evaluation starts from left hand side and if first evaluation returns false rest are not evaluated. In example above if a > b is false d is not evaluated.

Let’s see how these work for a value of **x = 20** ;

Operator	Operation	Commands	Results

<	Less than	x < 30	True
<=	Less than or equal to	x <= 10	False
>	Greater than	x > 10	True
>=	Greater than or equal to	x >= 10	True
==	Equal	x == 20	True
!=	not equal	x != 30	True
Is	object identity	x is 20	True
Is not	negated object identity	x is not 30	True

COMPARING SEQUENCES AND OTHER TYPES

- Sequences can be compared to other sequences of same type.
- First two items are compared and if they are equal, next two items are compared and comparison continues until end of sequence.
- Comparison is done using alphabetical order, ordering for strings uses the ASCII ordering

BOOLEAN OPERATORS

Boolean operators are used to make a true/false decision between two values. **AND OR** and **NOT** are the Boolean operator to make this determination.

Here are some examples of their usage for to test the true/false for the assigned values of X=20 and y=6

Operation	Operator	Command	Result
Either is True	Or	x == 20 or y == 5	True
		x == 10 or y == 5	False
Both are true	And	x == 20 and y == 6	True
		x == 20 and y == 7	False
False if x is true	Not	not x == 10	True
		not x == 20	False

When working with Boolean operator, here are some special notes about how they actually operate

- In **or** operations, second argument is evaluated only if the first one is false. In first OR example above the result is true based on first value itself.
- Similarly for **and** operations, second argument is not evaluated if the first one is false, if first value is false it returns the results as false
- **not** has a lower priority than non-Boolean operators, which are evaluated first and then not condition tested . not x == 20 in the example above x == 20 is tested first and then not condition tested.

False value definitions

Following values are considered as false when evaluated against other values. These can be user defined values or values returned from functions or calls for example :

None: None
False: False
zero of any numeric type 0, 0L, 0.0, 0j.
any empty sequence: ", (), [].
any empty mapping: {}.

BITWISE OPERATIONS ON INTEGER TYPES

Bitwise operation is done on binary bits instead of decimal numbers and is useful for integer values. These operators have lower priority than numeric operators and higher priority than comparison operators. These are mainly used in machine based computations which are done at bit levels such as routers switches and other telecom & networking equipment.

Operation	Result
x | y	bitwise OR operation on x , y
x ^ y	bitwise exclusive or of x and y
x & y	bitwise and of x and y
x << n	x shifted left by n bits
x >> n	x shifted right by n bits
~x	the bits of x inverted

Examples

Following example tests the bitwise ‘OR’ condition for decimal 10 And 18 values

```
>>>x = 10 | 18
>>>print x
>>> 26
```

Here bits are compared and if either of the two bit is true it result is true

0	0	1	0	1	0	10
0	1	0	0	1	0	18
0	**1 (16)**	**1 (8)**	**0**	**1 (2)**	**0**	**26**

Value in braces is the decimal value of bit position as per binary math.

Using same values for AND condition,

Result gets the true value if both bits are true. For example :

```
>>>y = 10 & 18
>>>print y
```

>>> 2

0	0	1	0	1	0	10
0	1	0	0	1	0	18
0	**0**	**0**	**0**	**1 (2)**	**0**	**2**

Only second value bit corresponding to binary 2 is true as it is true for both numbers.

4. REGULAR EXPRESSIONS

Regular expressions provide effective way to manipulate a string like match, search, replace, modify or split. Most of the python regex are available as module level function or as compiled regex object. The compiled object offers more parameters to work on as compared to the modules.

Python provides regular expressions similar to perl and this support is provided by a python modules and ‘re’ module.

Python offers flexibility of regular expression and it offers standard built in requlare expression and ability to compile and create custom regular expression for specific your needs.

Pre Complied Regular Expressions

Python’s re module has some commonly used regular expression functions built in and pre compiled.

These functions can be used after importing re module using import re command and specifying function name as re.function.

The main advantage of these re modules are that you can use flags to change the behavior using flags and being pre compiled, these are faster.

Here are the flags and functions available as part of re module.

Flags in re module:

- I (ignore case),
- L (locale dependent),
- M (multi-line),
- S (dot matches all),
- U (Unicode dependent), and
- X (verbose), for the entire regular expression.
- These flags are used as flag=re.<flag> as part of regular expression.

Functions in re module:

SPLIT FUNCTION

re.split function is used to split a string by a specified pattern. You can split a string with values separated by comma, colon, semicolon or any other arbitrary field separator and get the results as a list, flags can be any flag mentioned earlier.

Syntax is re.split(pattern, string, maxsplit=0, flags=0) , maxsplit specifies number of splits needed.

Splitting a string by field :

p=”aa:bb:cc:dd:ff:gg:hh”

A simple split base on field separator value of ‘:’

```
>>> re.split(':',p)
['aa', 'bb', 'cc', 'dd', 'ff', 'gg', 'hh']  ## result is a list of elements
```

a.) Limit the split using maxsplit

If maxsplit is non zero the split occurs for maxsplit times from left and rest of the string is returned as single element.

```
>>> re.split(':',p,maxsplit=2)
['aa', 'bb', 'cc:dd:ff:gg:hh']  ## Result is a list of two separate elements and a rest of the string.
>>> re.split(':',p,maxsplit=2)
```

b.) Using flags with split

Consider following string with some lowercase and uppercase alphabets as separators

s=”12x34y56z7X8Y9Z”

Split the string based on lowercase alphabet range

```
>>> re.split('[a-z]',s)
['12', '34', '56', '7X8Y9Z']   ## only lowercase letters used for split
```

Used the same expression with flag of Ignore case.

```
>>> re.split('[a-z]',s,flags=re.I)
['12', '34', '56', '7', '8', '9', ''] ## Case is ignored and string is split completely.
```

c.) Zero length matches

re.split doesn't split a string when the regex matches a zero characters.

For example:

```
>>>re.split(r'\b', 'a b')
>>> ['a b']
```

The result ['a b'] instead of ['', 'a', ' ', 'b', ''] is correct as and known behavior for zero length matches

SEARCH FUNCTION

re.search function is used search a pattern in a string argument and return match Object instance using. The function takes the flags arguments to modify the search behavior.

Syntax is re.search(pattern, string, flags=0)

n="james joe, smith eric"

>> re.search('joe',n,flags=0)

<_sre.SRE_Match object at 0x01F210C8> ## string joe is found in the string

>> re.search('Joe',n,flags=0) ## string Joe is NOT found in the string

Using flag

Use the same expression as above with flag of Ignore case to match upper case Joe

>>> re.search('Joe',n,flags=re.I)

<_sre.SRE_Match object at 0x01D9CD40> ## string joe is matched in the string

Zero Length Match

Zero length matches do not return any object but only return position

>>> re.search('\b',"A ,B",flags=0) ## no object is returned as only position matches.

This is not a bug but a known behavior for zero length matches.

MATCH FUNCTION

re.match () function matches a zero or more characters at the beginning of string & return a matchObject for character match.

The difference between search and match is that search looks for pattern in whole string while match only looks at the beginning of the string for pattern match. Even in multiline mode, match will only match at the beginning of the string, not at the beginning of lines. Search can be forced to look for beginning of strings only by using '^' in front of the pattern such as '^aa' .

Syntax for re.match () is re.match(pattern, string, flags=0)

>>> s="aa:bb:cc:dd"

>>> re.match('aa',s,flags=0)

<_sre.SRE_Match object at 0x01DBCD40>

>>> re.match('bb',s,flags=0) ## No match is found , bb is not at the beginning of string

Using flag

Use the same expression as above with flag of Ignore case to match upper case

>>> re.match('AA',s,flags=re.I) ## match found even though pattern is in upper case

<_sre.SRE_Match object at 0x01DBCD40>

Consider using different flag in each example

Zero Length Match

Zero length matches do not return any object but only return position

>>> re.match('\b',"A ,B",flags=0) ## no object is returned as only position matches.

This is not a bug but a known behavior for zero length matches.

FINDALL FUNCTION

re.findall() searches a string and return all non-overlapping matches of pattern as a list of found strings. The string is scanned left-to-right, and matches are returned in the order found.

If one or more groups are present in the pattern, return a list of groups. Empty matches are included in the result unless they touch the beginning of another match.

Syntax is re.findall(pattern, string, flags=0)

Consider the following string made up of characters separated by ':' and use of findall() on this string

>>> p="aa:bb:cc:dd:ff:gg:hh:aa:bb:cc:DD:EE:FF"

Find all occurrences of 'bb'

>>> re.findall('bb',p,flags=0)

['bb', 'bb']

Find all occurrences of 'a:b'

>>> re.findall('a:b',p,flags=0)

['a:b', 'a:b']

Using Flags

See the results from following examples when we try to find a string value without any flag and a one with a ignore case flag

>>> re.findall('dd',p,flags=0) ## No flag specified

['dd']   ## only lowercase match is returned

>>> re.findall('dd',p,flags=re.I) ## Ignore case flag specified

['dd', 'DD'] ## All matches are match is returned

FINDITERATOR FUNCTION

re.finditer() searches a string and return all non-overlapping matches of pattern as a iterator list. This list is useful in iteration loops using for loop. An iterator method has a __iter__ method define as following

>>> a = [1, 2, 3, 4]

>>> a.__iter__

<method-wrapper '__iter__' of list object at 0x01F4D850>

```
>>> a = "1, 2, 3, 4"
>>> a.__iter__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute '__iter__'
```

The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups. Empty matches are included in the result unless they touch the beginning of another match.

```
re.finditer (pattern, string, flags=0
```

Let’s use a html page with some name and use python to strip html code and print only the name. The string used here is multiline string enclosed by triple quotes.

```
html = '''<table border="1" width=100px>
  <tr>
      <td width="50%">James</td>
  </tr>
  <tr>
     <td width="50%">Eric</td>
  </tr>
  <tr>
   <td width="50%">Gordon</td>
  </tr>
</table>'''
```

Define a pattern to strip off html tags and get the content, The whole string match is returned as one group and the string inside the tags is returned as another match group marked by parentheses .

```
>>>pat = r'<td.*?>([\w\W]*?)</td>'
```

Run the result through finditer() function and print results

```
>>>for match in re.finditer(pat, html):
    print match.group(1) ## group 0 is whole line
```

The iterator returns the following results

```
James
Eric
Gordon
```

SUB - SUBSTITUTE FUNCTION

re.sub is used for finding a pattern and substituting it with given pattern.

Syntax is re.sub(pattern, repl, string, count=0, flags=0)

In this following example ‘:’ in the string as field separator are replaced by tabs

```
>>>st = 'aa:bb:cc:dd'
>>>result=re.sub(':','\t',st,flags=0)
>>>print result
aa   bb   cc   dd ## colons are replaced by tabs
```

PURGE FUNCTION

re.purge() clear the re module cache. re module caches expressions and other data in the memory and if you have long running processes they may put huge cache re data and may result in high memory usage by the re module. re.purge() should be called to clear cache to avoid such issues

ERROR FUNCTION

Exception raised when a string is passed to one of the functions is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching.

In the following example a regular express is being built based on some dynamic input value. We can use re.error to evaluate for correct regular express and throw an appropriate error to capture the details about input and error

```
>>>patterns=”aa)”
>>> if pattern:
...     try:
...       pattern_re = re.compile(r"\A(?:" + patterns + r")\Z")
...     except re.error as e:
...       raise AssertionError('Regex {!r} failed: {}'.format(
...         pattern, e.args[0]))
...
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
AssertionError: Regex 'aa' failed: unbalanced parenthesis
>>>
>>> print patterns
aa)
>>>
```

However It is never an error if a string contains no match for a pattern

ESCAPE META CHARACTERS

Return string with all non-alphanumeric back slashed.

This is useful if you want to match a literal string with lots of met characters inside.

Syntax is re.escape(string)

Consider following example of a url and how re.escape modifies it

```
>>> url='http://www.python.org/community/'
>>> re.escape(url)
'http\\:\\/\\/www\\.python\\.org\\/community\\/'  ## all non-alphanumeric back slashed
```

Compiled Regular Expressions

If you want to use some particular regular expression multiple times then it is better to compile it and reference it. Compilation does not make execution faster but only loading becomes faster due to compiled object. The compiled versions of the most recent patterns passed to re.match(), re.search() or re.compile() are cached

Regular expression patterns can be compiled using re.compile command. Once object is created it can be referred in subsequent call.

Basic syntax for compiling re object is

```
re.compile( pattern, option1|option2|option3 )
```

Generally options are used for debug, ignoring case, locale, verbose etc.

Here is the complete list of compile options and expression in Python.

Compile options for Regular Expression	
re.DEBUG	Display debugs information about compiled expression.
re.I	re.IGNORECASE - case-insensitive matching
re.L	re.LOCALE – This makes \w, \W, \b, \B, \s and \S dependent on the current locale.
re.M	re.MULTILINE, '^' matches beginning of the string & line, '$' matches end of string and line.
re.S	re.DOTALL Dot '.' special character match any character, including a newline; Without this flag, '.' exclude newlines
re.U	re.UNICODE Makes \w, \W, \b, \B, \d, \D, \s and \S dependent on the Unicode character set.
re.X	re.VERBOSE, Allows Whitespace within the pattern to help write regular expressions formatted for easy reading

EXPRESSIONS TO CREATE REGULAR EXPRESSIONS

Range	
[ ]	Set of characters or range to match [a-z] will match all alphabets. [0-2][0-9] matches two digit numbers from 00 to 29, Special characters lose their special meaning inside sets. [(+*)] matches '(', '+', '*', or ')'.

Characters	
'\'	Escapes special characters. Raw strings ‘r’ can be used to escape all special characters inside a string instead of individual characters
\s	matches any whitespace character, this is Equivalent to [\t\n\r\f\v]
\S	matches any non-whitespace character, Equivalent to [^ \t\n\r\f\v]
\w	matches alphanumeric characters & underscore. Equivalent to [a-zA-Z0-9_].
\W	matches any non-alphanumeric character Equivalent to [^a-zA-Z0-9_].

Strings	
\A	Matches only at the start of the string.
\b	Matches the empty string at the beginning or end of a word, r'\bSmith\b' matches ' Smith ' but not SmithJr or 'MrSmith'.
\B	Opposite of \b ,Matches empty string when it is not at the beginning or end of a word. r'py\B' matches 'python', 'py3', 'py2', but not 'py', 'py.', or 'py!'.
\Z	Matches only at the end of the string.

Special Characters	
'.'	Any character except a newline.
'^'	Start of a string, or line
'$'	End of the string or line or newline (\n)
'*'	0 or more repetitions of the preceding RE,. xy* will matches x , xy , xyyy , matches as much text as possible
‘+’	1 or more repetitions of the preceding RE. ab+ matches ab or abbb but not just ‘a’ , matches as much text as possible

‘?’	0 or 1 repetitions of the preceding RE. ab? Will match either ‘a’ or ‘ab’. Match as much text as possible
***?, +?, ??**	Limit scope of *,+ and ? above to match as few characters as possible
{m}	Match m copies of the previous RE, w{7} matches seven ‘w’ characters
{m,n}	Match from m to n repetitions of the preceding RE, a{3,7} matches 3 to 7 'a' characters. a{,7} matches 0 to 7 characters. a{3,} matches 3 to infinite characters.
{m,n}?	Limits scope of {m,n}, a{3,5}? Matches only 3 characters.
‘^’	Match all except character followed by ^, - [^9] matches all except ‘9’.

>>>help(re) gives a regular expressions help page in python shell.

COMPILE A REGULAR EXPRESSIONS

You can compile regular expression for your need, Following are two samples with description in comments

```
import re # Import re module


thealphabets = re.compile('[a-z]+')  ## recognize Lowercase alphabets
thedigits = re.compile('\d+')   ## recognize Digits in a string
```

USING COMPILED REGULAR EXPRESSIONS

The compiled regular expression can be used now in conjunction with functions to match, search or find these patterns in a string. The results are stored as pattern group().

To match a string pass string as argument to match function by referencing pattern variable, p

Let’s consider a string made up of strings and digits and try to see how these functions works and what is returned as value.

fruits = “apples 12, oranges 15, bananas 24”

MATCH FUNCTION

```
>>> x= thealphabets.match(fruits)  ## Store object in a variable
>>> x.group()     ## first string , apple matches, print returned value using a group()
'apples'
```

SEARCH FUNCTION

```
>>> x= thealphabets.search(fruits) ## Find Alphabet match x.group()
>>> x.group()
'apples'
```

```
>>> x=thedigits.search(fruits) ## Find numeric match
>>> x.group()
'12'
```

Search and match difference

Match and search looks similar but differs in functionality. Match looks at the beginning of string while search looks at entire string and can match a pattern anywhere.

Let's see results of search & match on following two strings

```
firstline = "aa-11-bb-12-cc-13-dd-14"
secondline = "11-bb-12-cc-13-dd-14"
```

```
>>> thealphabets.match(firstline)
<_sre.SRE_Match object at 0x01E0CD40>
```

Match found in firstline, which begins with lowercase alphabets

A Match for line fails as alphabets are not in the beginning & match checks only beginning of string.

```
>>> thealphabets.match(secondline) ### Nothing returned, no match
```

Repeat the same thing with search and search is successful for both as search looks at entire string.

```
>>> thealphabets.search(firstline)
<_sre.SRE_Match object at 0x01F71090>
>>> thealphabets.search(secondine)
<_sre.SRE_Match object at 0x01E0CD40>
```

FINDALL FUNCTION

In our compiled match and search regular expressions returns the first matching lowercase string, but in order to find all the occurrences in a string, findall function is used.

findall returns a list of matches and does not need a group() to print as the value is stored in variable.

```
>>>fruits = "apples 12, oranges 15, bananas 24"
```

Get all string match for the pattern

```
>>> x=thealphabets.findall(fruits)
>>> x
['apples', 'oranges', 'bananas']  ## Don't need group() to display a list
```

Get all numeric match for the pattern

```
>>> thedigits.findall(fruits)          ## Find all numeric values
['12', '15', '24']
```

FINDITERFUNCTION

finditer function can be used to iterate over a list of items and print them on new line for each match instead of as list as in previous examples.

Use of iteration depends on your requirement and generally the results are assigned to variable for comparing and making a decision using if statement.

```
>>>fruitnames = thealphabets.finditer(“apples 12, oranges 15, bananas 24”)
>>>print iterator
>>>for match in fruitnames :
>>>print match.group()
…….
apples
oranges
bananas

>>>fruitcounts = d.finditer(“apples 12, oranges 15, bananas 24”)
>>>print fruitcounts
>>>for match in fruitcounts:
 >>>print match.group()
……
12
15
24
```

Grouping of regular expressions

Grouping of regular expressions provides additional functionality of specifying special instructions to a regular expression.

Groups are created enclosing regular expression and instruction in parantheses.

The following example creates two subgroups from alpha numeric metch of the argument.

```
>>> pattern = re.match(r"(\w+) (\w+)", "James Bond, spy")
>>> pattern.group(0)       # Matches whole group
'James Bond'
>>> pattern.group(1)       # The first subgroup.
'James'
>>> pattern.group(2)       # The second subgroup.
'Bond'
>>> pattern.group(1, 2)    # Return a tuple.
('James', 'Bond')
```

5. CONTROLS & LOOPS

Control structure and loops are used to go through a sequence of data and evaluate if certain conditions are true.

Python has traditional control structure and loops in the form of if, else, elif, while, & for functions with following differences from other common programming languages.

1. The control structure in python is marked by indentation as compared to parentheses and braces in other languages. Statements within a control structure are grouped by same amount of indentation.
2. A colon ‘:’ marks the end of line for execution block.
3. A blank line in the interactive parser mode indicates the end of control block and an Enter causes it to execute.

IF CONDITION

It evaluates a condition and if the condition is true the executes commands.If <expression> :

```
print “Something”
```

For example

```
if lastname == “smith”:
 print “Hello Mr. Smith “;
Hello Mr. Smith
```

ELSE CONDITION

else provides alternate statements or execution functions to the if clause

In this example condition is true so only first control block is executed to print message.

```
>>>lastname = “smith”
>>> if lastname == “smith”:
... print "Hello Mr. lastname "
... else:
... print "Hello Guest"
...
Hello Mr. smith
```

In this example condition is false and alternate control block, else, is executed to print message.

```
>>>lastname=”brown”
>>>lastname = “smith”
>>> if lastname == “smith”:
... print "Hello Mr. lastname "
... else:
... print "Hello Guest"
...
Hello Guest
```

ELIF CONDITION

elif is a short form for else if & it is useful to evaluate multiple if conditions within a if control block Following example evaluates the input color and provides the response accordingly using if and elfi

```
>>> color = "Yellow"
>>> if color == "Yellow":
... print "On Sale Today - Yellow flowers"
... elif color == "Blue":
... print "On Sale Today Blue Flowers"
... elif color == "Pink":
... print "On Sale Today Pink Flowers"
... else:
... print "Sorry ,” ,color,” flower are not on Sale today "
...
On Sale Today - Yellow flowers
>>>
```

WHILE LOOP

While evaluates a conditional loop and keeps on executing loop until the condition is true. If condition doesn't change to false inside the loop and stays true, the loop will run forever. To avoid infinite loops there should be condition checker inside the loop to modify the condition.

In the following example the while checks the condition and the body of the loop increments. While loop exists once the value of b reaches 5 and while condition of b <5 becomes false.

```
>>> b=1
>>> while b < 4:
... b=b+1
... print b
...
 2
3
>>>
```

FOR LOOP

For loop iterates through a sequence of strings or a list in the order they appear in the sequence. In its simplest form :

```
>>> seq=["AA","BB","CC","DD"]
>>> for i in seq :
... print i
...
AA
BB
CC
DD
>>>
```

Here are some important functions to use to provide sequences to for loop.

RANGE FUNCTION

Range function generates a range of numbers depending on the incremental step options. The result from a range function does not include the final specified number.

A simple range function with only a number gives a range of numbers from 0 to that number

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that number 10 is not part of the generated list

Here are some examples of range with different options

Intermediate range , not starting from 0

```
>>> range (5, 10)
[5, 6, 7, 8, 9]
```

A range with incremental steps of 100-500 range with 50 step increment

```
>>> range(100, 500, 50)
[100, 150, 200, 250, 300, 350, 400, 450]
```

Some Examples for loop usages

1. **Enumerate function:** The results can be enumerated with the help of enumerate function; also the starting number can be changed with start argument.

Consider the following example:

```
>>> a="a b c d e"
>>> for i in list(enumerate(a)):
... print i
...
(0, 'a')
(1, ' ')
(2, 'b')
```

```
(3, ' ')
(4, 'c')
(5, ' ')
(6, 'd')
(7, ' ')
(8, 'e')
```

Starting enumerating from number 2 onwards:

```
>>> for i in list(enumerate(a,start=2)):
... print i
...
(2, 'a')
(3, ' ')
(4, 'b')
(5, ' ')
(6, 'c')
(7, ' ')
(8, 'd')
(9, ' ')
(10, 'e')
>>>
```

2. **Zip Function : zip function can merge two values in variables and for loop can be used to zip a list of items**

```
>>> itemcolor=['Red','Blue','Green','Orange']
>>> itemnamename=['Shirts','Pants','Tops','Scarfs']
>>> for color, name in zip(itemcolor,itemnamename):
... print '{0} {1}.'.format(color, name)
...
Red Shirts.
Blue Pants.
Green Tops.
Orange Scarfs.
>>>
```

3. **Reversed function: This function** prints a range in reverse order.

```
>>> for i in reversed(range(0,9,3)):
... print i
...
 6
3
0
>>>
```

4. **Sorted Function :** Print a sorted list using sorted function

Alphabetic sort:

```
## string
>>> x=['AA','CC','BB','FF','DD']
>>> for i in sorted(x):
... print i
...
AA
BB
CC
```

Numeric Sort:

```
FF
>>>

## Number
    >>> n= [1,2,4,6,7,3,9,8,5]
    >>> for i in sorted(n):
    ... print i
    ...
    1
    2
    3
    4
```

```
5
>>>
```

5. iteritems() function: This function retrieves key values from a dictionary where items are arranged in key value pairs.

```
>>> names = {'Sky': 'Blue', 'Snow': 'White'}
>>> for a, n in names.iteritems():
... print a, n
...
Sky Blue
Snow White
>>>
```

BREAKING THE LOOP

break statement provides a way to break the loop. It has to be used with a if else clause so that it breaks only when some condition becomes true.

Below example shows a for loop set to print a range of 10 numbers but broken at number 8 using a break statement.

```
>>> for n in range(1, 10):
...  if n == 8:
...  print n, "reached max Quitting"
...  break
...  else:
...   print n
...
1
2
3
4
5
6
7
```

8 reached max Quitting

>>>

PASS STATEMENTS

Pass statement is not an operational statement, and nothing happens when it is executed. The only usages for pass is the places where a statement is needed to complete the syntax but there is nothing for execution

Some potential usage for pass statement are :

a.) As place holder for function which are a place holder for some new functionality:

```
 >>> def tbd():
... pass;
...
>>> tbd
<function tbd at 0x02D09AB0>
>>>
```

b.) To catch only one conditional evaluation and ignore the other condition

```
>>> if a==1:
... pass
... else:
... print “Mismatched”
...
Mismatched
>>>
```

c.) Ignore some exceptions:

```
>>> try:
... a = 100/0
... except ZeroDivisionError:
... pass
```

...
>>>

6. FILE OPERATIONS, INPUT OUTPUT

Python provides several function and arguments to work on text and binary files. Following paragraphs describes usages and examples to work with files, input and output of data

Python built in function open() opens a file provided as argument, default mode is read only if no other argument is provided

Basic syntax is :

```
open(file name, options)
```

MODES

A file can be opened in three modes - read (r) write (w) or append (a), r+ opens for read and write.

Examples

Following example opens a file in different modes and assigns it to a file handle f

```
>>> f=open('README.txt','r') )## Open a file in read only mode
  >>> f
  <open file 'README.txt', mode 'r' at 0x026E38B8>
  >>>
```

TEXT AND BINARY FILES

Python on Windows makes a distinction between text and binary so to open binary files in Windows binary (b) is added to operation mode of r, w a like binary file operation in windows

UNIX does not make a distinction between text and binary file but does not harm to use a 'b' to open binary files.

```
>>>
>>>
>>> f=open('README.txt','rb')
>>> f=open('README.txt','wb')
>>> f=open('README.txt','ab')
>>>
```

READING FROM FILE

Once a file is opened for reading you can read its contents in different ways, the file can be read as whole, it can be read line by line or you go to specific file position.

Read whole file

Python built in function read() is used to read a file , it takes an optional size argument and returns whole file content as string. If size is mentioned, chunk of data is read and returned as string and an empty string “” is returned at End of file .

It is advisable to mention the file size because if you have large files they may take up most of the memory causing system to slow or crash altogether.

Note that read() method only returns strings if there are numeric values they have to be passed to a function like int() , which takes a string as input and returns its numeric value. However to simplify the matter Python provides a built in function pickle/unpickle to take care of converting strings in to integers in an entire file with multiple numeric values.

More details about pickle at the end of this section.

Basic syntax is

<Filehandle>.read(size) or <Filehandle>.read()

```
>>> file = open('logfile.txt','r')
>>> content=file.read()
```

```
>>> print content
Line 1
Line 2
Line 3
Line 4
```

READ FILE BY LINES

readline function returns a single line ending with \n except if last line don’t end with a\n

End of file is marked by a empty string “”

Read One Line at a time

Blank lines are marked by \n

```
>>> fl = open('logfile.txt','r')
>>> fl.readline()
'Line 1\n'
>>> fl.readline()
'Line 2\n'
>>> fl.readline()
'Line 3\n'
>>> fl.readline()
'Line 4'
>>> fl.readline()
''
>>>
```

a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline

Reading all lines in a list

All lines from a file can be read in a list using readlines() function

```
>>> ff = open('logfile.txt','r')
>>> ff.readlines()
['Line 1\n', 'Line 2\n', 'Line 3\n', 'Line 4']
```

This list can be used inside a for loop to print each line separately as below:

```
>>> for i in ff:
... print i
...
Line 1

Line 2

Line 3

Line 4
```

List() function can be used as well to print file content in to list as below

```
>>> ff = open('logfile.txt','r')
>>> a=list(ff)
>>> print a
['Line 1\n', 'Line 2\n', 'Line 3\n', 'Line 4']
```

PRINT FILE CONTENT USING FOR LOOP

Using for loop for reading lines from a file object is memory efficient, fast, and simple to code

```
>>> ff = open('logfile.txt','r')
>>> for row in ff:
... print row
... ff.close()

Line 1

Line 2

Line 3

Line 4
```

Extra new lines in the output

Notice the extras new lines, these are originating from print statement, one way to suppress them is to use write function to stdout using sys module with an import command

```
>>>Import sys
>>> ff = open('logfile.txt','r')
>>> for row in ff:
... sys.stdout.write (row)
...
Line 1
Line 2
Line 3
Line 4>>>
```

FILE SEEK

File seek returns an integer giving the file object's current position in the file. The position is measured in bytes from the beginning of the file.

Syntax for file seek is

f.seek(offset, from_what)

f is file object , offset is the position to seek and from what is a reference point , defaults to beginning of file if not specified .

0 , from what value measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. From what defaults to 0 , beginning of file If not mentioned. See examples below for more clarity

```
>>> f = open('workfile', 'w+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)
>>> f.read(1)
'5'
>>> f.read(1)
```

```
'6'
>>> f.read(1)
'7'
>>> f.read(1)
'8'>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

f.tell()

f.tell() returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file.

For example create a file named as workfile with these letters as content

```
0123456789abcdef
```

Now open & read the file selectively and the end use f.tell to find the current position

```
>>> f = open('workfile', 'w+')
>>> f.seek(5)
>>> f.read(1)
'5'
>>> f.read(1)
'6'
>>> f.read(1)
'7'
>>> f.read(1)
'8'
>>> f.read(1)
'9'
>>> f.seek(-3, 2)
>>> f.read(1)
'd'
>>> f.seek(-3, 2)
>>> f.tell()
13L
>>> f.seek(-2, 2)
```

```
>>> f.tell()
14L
```

WRITING TO FILE

Once a file is opened in Write or read/write mode, content can be written into it using write function

Syntax is fileObject.write(“content”)

Write a string

```
f.write(string)
```

or to create a file with a string

```
>>>f.write('0123456789abcdef') ### Writes the contents of string to the file,
```

Write a non string

To write something other than a string, it needs to be converted to a string first:

```
>>> f.write('This is a test\n')
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
>>> f.close()
```

CLOSING FILE WITH CLOSE() FUNCTION

Open files needs to be closed when all the operation is done to free up any system resources taken up by the open file. with keyword allows files to as soon as file operations finishes. Even if there is an exception it closes the file after exit and you don’t have to write extra code to close files and exception handling.

```
>>> with open('workfile', 'r') as f:    ### Use with to open file
...   read_data = f.read()
>>> f.closed   ### Test if file is closed
```

```
True
>>>
```

7. LISTS

List in python are values separated by commas and enclosed in square brackets like [a,b,c,d] .

Lists can have compound data types of strings and numbers and can be rearranged or sliced.

The elements in a list are indexed starting from zero and can be accessed using index values.

CREATING A LIST

A list can be created by assigning elements to a variable in following format.

Create a list of 5 elements with numbers and strings

```
>>> a = ['Apple','Banana','Coconut',10, 20, 30]
>>> print a
['Apple', 'Banana', 'Coconut', 10, 20, 30]
```

ACCESSING THE LIST ELEMENTS

U se index value (starting from 0) to access the elements of a list and print them:

```
>>> a[0]
'Apple'
>>> a[1]
'Banana'
>>> a[2]
'Coconut'
>>> a[3]
```

```
10
>>> a[4]
20
>>> a[5]
30
```

Also Values can be accessed from the end of list as well, -1 being end of list

```
>>> a[-1] ### Last value of the list
 30
>>> a[-6] ### First value of the list , 6th value from the end
'Apple'
```

You can specify a range to extract from list

Range of 2nd element to end of list, If end range is not mentioned it is taken as end of list

```
>>> a[1:]  ## 0 is Apple
['Banana', 'Coconut', 10, 20, 30]
```

if starting range is not specified it is taken as 0 , beginning

```
>>> a[:2]
['Apple', 'Banana']
```

Range of 1st element from beginning to second last

```
>>> a[0:-1]
['Apple', 'Banana', 'Coconut', 10, 20]
```

Range of 2nd element from beginning to second last

```
>>> a[1:-1]
['Banana', 'Coconut', 10, 20]
```

Some example of ranges

```
>>> a[2:-1]          ### Extract 3 elements
['Coconut', 10, 20]
```

```
>>> a[2:-2]          ### Extract 2 elements
['Coconut', 10]
>>> a[2:-3]          ### Extract 1 elements
['Coconut']
```

MODIFYING A LIST

A list can be changed to modify the individual elements of a list:
To insert elements at specific positions

```
>>> a[:2]+['Mango',46] ### Add elements after 2nd position , (0-2 )
['Apple', 'Banana', 'Mango', 46]
```

Modification of list with Arithmetic operations:
Elements can be multiplied added or subtracted with arithmetic operators.

For example:

Multiply a range 0-4 by two times

```
>>> 2*a[:4]
['Apple', 'Banana', 'Coconut', 10, 'Apple', 'Banana', 'Coconut', 10]
```

Multiply a range 0-4 by two times & add a element after that

```
>>> 2*a[:4]+['Orange']
['Apple', 'Banana', 'Coconut', 10, 'Apple', 'Banana', 'Coconut', 10,
'Orange']
>>>
```

LIST METHODS

Elements can be added, deleted moved around with the help of following methods.

Append

Append method is used to adds element at the end of the list.

Syntax is

<listname>.append(<element>)

Following example appends numbers 4 & 5 to an existing list of mynumbers.

```
>>> mynumbers=[1,2,3]    ### Existing List
>>> mynumbers.append(4)  ### Append number 4
>>> mynumbers          ### List after append
[1, 2, 3, 4]
>>> mynumbers.append(5)  ### Append number 5
>>> mynumbers          ### List after append
[1, 2, 3, 4, 5]
>>>
```

extend

Extend method in a list extends an existing list by appending all the items from another list.

Append is an element addition while extend is a list addition.

<listname>.extend(<another list>)

Following example illustrates the appending of an existing list with a new list

```
>>> mynumbers           ### Existing List
[1, 2, 3, 4, 5]
>>> b=['AA','BB','CC']          ### Another list
>>> mynumbers.extend(b)  ### Extend mynumbers list with new list b
>>> mynumbers
[1, 2, 3, 4, 5, 'AA', 'BB', 'CC'] ### Final list after append
>>>
```

Insert

Insert method inserts an item at a given position in a list.

Syntax is

<listname>.insert (pos,value)

Where list is name of an existing list, pos is the index position of the element to be inserted and value is the value to be inserted.

```
>>> mynumbers
[1, 2, 3, 4, 5, 6]
```

Insert word ZERO at the beginning of list . index value = 0

```
>>> mynumbers.insert(0,'ZERO')
>>> mynumbers
['ZERO', 1, 2, 3, 4, 5, 6]
```

Insert word FIVE at the index value of 5

```
>>> mynumbers.insert(5,'FIVE')
>>> mynumbers
['ZERO', 1, 2, 3, 4, 'FIVE', 5, 6]
```

remove

Remove method removes a defined item from the list

Syntax is

<listname>.remove (value)

Remove the first item from the list whose value is x. It is an error if there is no such item.

Lets use the same list as in insert and try to remove inserted values from the list in following examples

```
>>> mynumbers    ### Current list values
['ZERO', 1, 2, 3, 4, 'FIVE', 5, 6]
```

Remove string FIVE from the list

```
>>> mynumbers.remove('FIVE')
>>> mynumbers   ### Check Current list values
['ZERO', 1, 2, 3, 4, 5, 6]
```

Remove string ZERO from the list

```
>>> mynumbers.remove('ZERO')
>>> mynumbers ### Check Current list values
[1, 2, 3, 4, 5, 6]
```

Error is thrown id the item to be removed is not in the list

```
>>> mynumbers.remove('FIVE')
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>>
```

POP

Pop method removes an item from a given position and returns its value.

Syntax is :

<listname>.pop(i)

Where I is the index value, index value is optional. If Index is not mentioned it defaults to last value of the list

Let’s use pop method in our list of numbers and remove odd numbers from the list.

```
>>> mynumbers  ### Currrent values in the list
[1, 2, 3, 4, 5, 6]
```

Remove element at index 0

```
>>> mynumbers.pop(0)
1                          ### Value of removed element
>>> mynumbers    ### Remaining values in the list
[2, 3, 4, 5, 6]
```

Remove element at index 1

```
>>> mynumbers.pop(1)
3                           ### Value of removed element

>>> mynumbers
```

```
[2, 4, 5, 6]                ### Remaining values in the list
```

Remove element at index 2

```
>>> mynumbers.pop(2)
5                           ### Value of removed element
>>> mynumbers
[2, 4, 6]                   ### Remaining values in the list
```

If no index is specified pop removes and returns the last item in the list.

```
>>> mynumbers.pop()
6                           ### Value of last element
>>> mynumbers
[2, 4]                      ### Remaining values in the list
>>>
```

DELEING ITEM/S FROM THE LIST

Delete method removes an element, range of element or delete an entire list

Syntax is:

```
del <listname[range]>
```

The listname is an existing list and range of one or many elements denoted by index value.

Note that del method does not return value of deleted items unlike pop.

Lets see the operations using our original list values

```
>>> mynumbers=[1, 2, 3, 4, 5, 6]
>>> mynumbers
[1, 2, 3, 4, 5, 6]
```

Delete element at index value 0

```
>>> del mynumbers[0]   ### Nothing is returned
>>> mynumbers
[2, 3, 4, 5, 6]             ### Remaining values in the list
```

Delete a range of elements

```
>>> del mynumbers[2:4]  ### Delete range of 2 to 4 index value
>>> mynumbers
[2, 3, 6]                ### Remaining values in the list
```

Entire list can be deleted by not specifying start and end range:

```
>>> del mynumbers[:]
>>> mynumbers
[]                  ### list is empty, with just list name
>>>
```

Delete the variable holding list values:

You can delete the variable name itself to remove all existence of list

```
>>> del mynumbers
>>> mynumbers
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'mynumbers' is not defined
>>>
```

LIST OPERATIONS

List operations are done on the entire list as described below

Index

It returns the index value of the first item in a list. Returns an error if there is no such item.

Syntax is

<listName>.index(x)

Where x is the value for which index needs to be find out

```
>>> mystring=['aa','bb','cc','dd','ee'] ]   ### create a new List
>>> mystring.index('cc')   ### Index value of ‘cc’
2
>>> mystring.index('ee')  ### Index value of ‘ee
```

```
4
>>> mystring.index('aa')   ### Index value of ‘aa’
0
>>>
```

count

count return the number of times a list element appears in the list.

Syntax is

mynumbers**.count** (x)

Where x is the list element

Let’s create a list and find the count:

```
>>> mystring=['aa','bb','cc','dd','ee','cc'] #### List with two ‘cc’ element
>>> mystring.count('cc')
2               ### Count of cc elements
>>> mystring.count('aa')
1               ### Count of aa elements
>>> mystring.count('c')
0                       ### Non element match count is zero
>>>
```

sort

This is in place sort and this operation is used to sort the items of the list. The operation does not return the sorted list but sort the original list.

Syntax is :

<listname>.sort()

Lets create a new list to see sort operation

```
>>> mynumbers=[1,3,5,7,2,4,6]   ### new unsorted list
>>> mynumbers.sort()          ### In place sort of list , nothing returned
>>> mynumbers
```

```
[1, 2, 3, 4, 5, 6, 7]              ### In place sort of list , original list is
sorted
>>>
```

Reverse

reverse operation reverses the elements of the original list.

This operation does not return the reversed list but reverses the original list.

Syntax is :

```
 <listname>.reverse()
```

```
>>> mynumbers
[1, 2, 3, 4, 5, 6, 7]              ### Values in the list
>>> mynumbers.reverse()    ### In place reverse of list , nothing
returned
>>> mynumbers
[7, 6, 5, 4, 3, 2, 1]              ### Reversed list
>>>
```

Using Lists as Queues

Lists can also be used as queue where elements are added and retrieved in the order in which they arrive, first in first out. However, lists are not efficient for this purpose because doing inserts or pops from the beginning of a list is slow. Any pop or insert causes all elements in the list to be shifted one by one.

LIST COMPREHENSIONS

List comprehensions allow creating lists based on expressions and conditions. A list comprehension consists of an expression followed by a for clause then followed by zero or more for or if clauses.

mylist = [expr for <condt>:for <loop: if <condt>>

Examples

Create a list of squares

Following a is a simple list comprehension to find a list of squared for numbers from 0 to 10 using expr & for loop

```
>>> mysquares = [x**2 for x in range(10)]
>>> mysquares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Or

get both values, original and squared

```
>>> mysquares = [(x,x**2) for x in range(10)] ## for range 10 print org. and sq. values
>>> mysquares
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
>>>
```

Using conditions to create list with range 0-10 but only from number less than 6

```
>>> mysquares = [(x,x**2) for x in range(10) if x<6]
>>> mysquares
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)] ### List with squares from 0 to 5
```

Create list with range 0-10 but only from number greater than 6

```
>>> mysquares = [x**2 for x in range(10) if x>6]
>>> mysquares
[(7, 49), (8, 64), (9, 81)]   ### List with squares greater than 6
```

Create list with range 0-10 but excluding number 6

```
>>> mysquares = [(x,x**2) for x in range(10) if x!=6]
>>> mysquares
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (7, 49), (8, 64), (9, 81)] ## No number 6,36
```

8. TUPLES & SETS

Tuples are looks similar to list but they are quite different. A tuple is number of values separated by commas such (1, 2, 3, 4, 5).

1. Elements in a list can be changed; you can’t change elements of a Tuple.
2. List has homogenous elements accessed by iterating the list, Tuples usually contain an heterogeneous sequence of elements that are accessed via unpacking, indexing or by attribute.

CREATING A SIMPLE TUPLE

This is a simple tuple made up of numbers and a name, heterogeneous elements.

```
>>> tp = 1234, 56789, 'george' ### Also known as packing
```

Values are indexed in arrays and can be accessed by index key

```
>>> tp
(1234, 5678, 'george')
>>> tp[0]
1234
```

CREATING A NESTED TUPLE

You add a existing tuple to new one with multiple objects.

```
nt = tp,(1,2,3,4)
>>> nt
  ((1234, 5678, 'george'),(1, 2, 3, 4, 5))
```

Output is enclosed in parenthesis to protect the object identity.

Tuples are immutable

Trying to assign a new value to existing value fails

```
>>> tp[0]
1234 ## Current Value
>>>tp[0] = 0123 ## new value assignment
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Empty Tuples

Empty Tuples are constructed by an empty pair of parentheses;

```
>>> tpempty = ()
>>> len(tpempty)
0
```

Single Item Tuples

A tuple with one item is constructed by following a value with a comma. Without comma it is just a variable holding single assigned value.

```
>>> tpsingle = 'word',  # <-- note trailing comma
>>> len(word)
1
>>> tpsingle
('word',)
```

UNPACKING TUPLES

Assigning values to create a tuple is called packing and there is a process to retrieve elements from tuple , which is called unpacking

```
>>> tp = 1234, 56789, 'george' ### Also known as packing
```

Unpacking involves assigning same number of variables as in tuple

```
>>> a,b,c=tp
```

Now check the value of assigned variables

```
>>> a
1234
>>> b
6789
>>> c
'george'
>>>
```

SETS

A set is a collection of unordered elements, without any duplicate.
The sets are mostly used for eliminating duplicate entries & membership testing between two sets..

Sets can be created with curly braces or the set() function

Examples of eliminating duplicate

```
>>> letters = ['AA', 'BB', 'CC', 'CC', 'DD', 'AA']
>>> myletters = set(letters)      ### create a set without duplicates
>>> myletters
set(['AA', 'CC', 'DD', 'BB'])        ### Unordered collection
```

Membership testing

Membership testing involves testing if a element belongs to a given set or not. Results are in true /false statement

```
>>> AA in myletters
True
>>> 'AB' in myletters
False
```

SET OPERATIONS

Sets support mathematical & logical operation, they can be added, subtracted or compared as in following examples.

Mathematical Operations on sets:

Subtracting

You can subtract one set from another to get the remaining objects

```
>>> a=('a','b','c','d','e')
>>> b=('a','b','c','d','e','f','g')
>>> sa=set(a)
>>> sb=set(b)
>>> sa-sb
set([])
>>> sb-sa
set(['g', 'f'])
```

Logical OR Operation

Find unique letters in either of the sets

```
>>> sa|sb
set(['a', 'c', 'b', 'e', 'd', 'g', 'f'])
>>>
```

Logical AND operation

Find unique letter in both sets

```
>>> sa&sb              # letters in both a and b
set(['a', 'c', 'b', 'e', 'd'])
```

Logical negate operation

Find unique letter in one of the sets which are not in both sets

```
>>> sa^sb             # letters in a or b but not both
set(['g', 'f'])
>>>
```

9. DICTIONARIES

Dictionaries are a set of key: value pairs, with unique keys similar to associative arrays in perl.

The main operations on a dictionary is storing and retrieving keys & their value.

Relationship between dictionaries, Tuples and Lists:

In dictionary - A key value pair can be deleted using del command but it cannot be changed, new key values can be added but exiting keys cannot be modified

Tuples can be used as keys if they don’t have a mutable object, even indirectly.

Lists can’t be used as keys as lists can be modified using index, slice, append and extend

DICTIONARY OPERATIONS

Create a new dictionary

A pair of curly ,{}, braces creates an empty dictionary. To create a dictionary with key, values assign name of dictionary to key values seated by a : and enclosed by { } braces. Printing a dictionary also appears in the same format.

```
>>>a={}  ### Creates a empty dictionary
>>> d={'John':100,'George':200,'James':300}
```

Get dictionary listing using its name.

```
>>> d
{'James': 300, 'John': 100, 'George': 200}  ### Values in dictionary ‘d’
```

Find value of a key, James

Use dictionary names and key to its value.

```
>>> d['James']
300
>>>
```

Find all the keys in dictionary ‘d’ using key() method. keys() method returns a list of all the keys used in the dictionary in unsorted original order.

```
>>> d.keys()
['James', 'John', 'George']          ### Keys in unsorted format
```

Sorted function to sort keys
Use dictionary name & keys() method to get the keys and then use sorted to get the keys sorted

```
>>> sorted(d.keys())
['George', 'James', 'John']          ### Keys in sorted format
>>>
```

Assign a new key: value
Use dictionary & key name to assign a new value.

```
>>> d['Eric']=400                    ### Add a new key value , Eric =
400
>>> d
{'James': 300, 'John': 100, 'George': 200, 'Eric': 400} ### New Values
in the dictionary
>>>
```

Delete a Key
del method deletes the key & its value from dictionary.

```
>>> del d['John']                    ### Delete key name John
>>> d
{'James': 300, 'George': 200, 'Eric': 400}    ### New dictionary values,
no John here
>>>
```

Test Membership

You can test if a key exists in a dictionary or not, the result are in True/False statement

```
>>> 'Eric' in d
True
>>> 'John' in d
False
>>>
```

DIFFERENT WAYS TO CREATE DICTIONARIES

Besides assigning a key value pair, there are some other ways to create dictionaries as described below.

1. **Using sequences**

The dict() function can build dictionaries directly from key-value sequences pairs as in the example below

```
>>> dd=dict([('AA', 100), ('BB', 200), ('CC', 300)])
>>> dd
{'AA': 100, 'CC': 300, 'BB': 200}
>>>
```

2. **Using expressions**

Dictionaries can be created using expressions to create keys and value pairs

In the following example x in the range of 2 to 5 is squared and the value of x and its square is printed as dictionary

```
>>> a= {x: x**2 for x in range(2,5)}
>>> a
{2: 4, 3: 9, 4: 16}   ### Created Dictionary with key value pair
>>>
```

10. PICKLE UNPICKLE

Pickling & unpickling is the process of preserving the python object by way of serialization and deserialization. Pickling converts an input object in to byte stream object. The object can be stored transmitted or converted back to the input object using unpickle.

USAGES

Two main usages of pickle/unpickle is

1. Picked object can be stored or transmitted
2. Integer values can be written to the pickled object as you are writing it as byte stream.

There are some security concerns with pickled objexts as a pickled object can be hacked and packed with malicious code, opening a pickled object from unknown source poses a security risk.

Simplest way to pickle the object is to import pickle module and call the pickle function

```
>>import pickle
```

PICKLE FUNCTIONS

pickle.dump(‘DATA’,‘FILE) , write pickled data, DATA in to a file,FILE)

pickle.load(f) - Reads pickled data from file and presents in unpickled

SOME EXAMPLES

Write an integer value in a file

```
>>> f = open('data.txt','wb')   ### open a file for writing , Note the binary mode for writing non strings
```

```
>>> pickle.dump(94555,f)    ### Write number 94555 in file object ,
f, using pickle.dump()
>>> f.close()              ### Close file after writing
```

Retrieve the value from the file using pickle.load

```
>>> f = open('data.txt','r') ### Open file for reading
>>> x = pickle.load(f)     ### retrieve value using pickle.load()
>>> x
94555
>>>
```

Another example to show use of pickle to save mixed data and retrieval using pickle

```
>>> l = ['Bob', 'Roy','Chris',50, 75, 200] ## a list of numbers and
strings
>>> f = open('data.txt','wb')         ## Open file for binary write
>>> pickle.dump(l,f)                ## Write using pickle.dump
>>> f.close()
>>> f = open('data.txt','r') ## Open file for reading
>>> x=pickle.load(f)      ## retrieve value from file using pickle ,
>>> x
['Bob', 'Roy', 'Chris', 50, 75, 200]
>>>
```

11. FUNCTIONS

Functions are written to perform a specific function in a program. Generally functions are called with parameters as input to the function and may return a value or just the exit status.

BUILT IN FUNCTIONS

These are part of Python library and are always available. Some of the functions have been used in this book and a complete list can be obtained by using dir() function at python console prompt.:

>> dir(__builtins__)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',

'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']

>>>

To get syntax and more detail of a function, use

help(function_name), like help for sum function below.

```
>>> help(sum)

Help on built-in function sum in module __builtin__:

sum(...)

  sum(sequence[, start]) -> value

  Return the sum of a sequence of numbers (NOT strings) plus the value

  of parameter 'start' (which defaults to 0). When the sequence is empty, return start.
```

CREATING A CUSTOM FUNCTION

A function is created using def statement with the name of function parenthesis and colon.

def name_of_function():

Body of the function is formed by statements indented by the same amount. Unlike braces in most of the programming languages to enclose function statements, python depends on indentations to include statements in the function body, same statement levels has to be at the same level of indentation.

Any statement not indented is considered as normal line not as part of function

Example: Let’s create a function to print some names

```
>>> def names():          ### Create function called ‘names’
  ... print 'James'        ### Function body statements starts
  ... print 'George'
  ... print 'Harry'
...                ### Function body statements ends with end in
indentation
>>> names              ### Checks for existence og function object
<function names at 0x01FE5970>
```

EXECUTING A FUNCTION

Functions are executed by calling function names with parentheses. Parentheses in the function call causes function to be executed.

```
>>> names()            ### Execute function created earlier to get
output
James
George
Harry
>>>
```

EXECUTING MULTIPLE FUNCTIONS

Multiple functions can be defined and called in the program later, the functions needs to be defined first before calling for execution

```
>>> def function1():
... print 1
...
>>> def function2():
... print 2
...
>>> def function3():
... print 3
...
>>> def function4():
... print 4
...
```

```
>>> function4()
4
>>>
```

These functions can be called for execution in multiple ways

CALL ONE FUNCTION AT A TIME AS IN THE EXAMPLE BELOW

```
>>> function1()
1
>>> function2()
2
>>> function3()
3
>>> function4()
4
```

CALL BY GROUPS

You can create a function group to execute multiple functions in one call.

Let’s make two function groups to print even and odd numbers from functions we created earlier.

evenfunction() prints the even numbers.

```
>>> def evenfunction():
... function2()
... function4()
>>> evenfunction()
 2
4
```

evenfunction() prints the odd numbers.

```
>>> def oddfunction():
... function1()
... function3()
...
>>> oddfunction()
```

```
1
3
 >>>
```

CALL EVEN AND ODD FUNCTIONS ONE BY ONE

```
>>> evenfunction()
2
4
>>> oddfunction()
1
3
>>>
```

CALL BY MAIN FUNCTION

Define a main function to execute all the functions

```
>>> def main():
  ... function1()
  ... function2()
  ... function3()
  ... function4()
  ...
>>> main()
1
2
3
4
>>>
```

FUNCTION PARAMETERS & VARIABLES

Function parameters are used to supply additional inputs to a function for processing. Parameters are defined as a placeholder while defining a

function and multiple parameters can be passed to the function inside parentheses.

To execute, a function is invoked with the parameters value. The value is passed on to the function and gets substituted in the variable name inside the function.

```
>>> def fruit(name):          ### Function is defined with a parameter called name.
  ... print name,"is Delicious!"   ### Function body statement, prints parameter value & text.
...
>>> fruit('Apple')            ### Function Execution
Apple is Delicious!          ### Parameter value is substituted, message is printed
>>>
>>>
```

FUNCTION SCOPE

Scope of functions is hierarchical in nature; the top level function has a global scope for the functional defined below its level. Anything defined at upper level function will be available to the below level functions.

MULTIPLE PARAMETERS

The parameters in function call are evaluated from left to right and associated with variables in that order.

Following example defines three parameters and calculated the volume of the object.

```
>>> length=2
>>> width=4
>>> height=5
>>> def getvol(length,width,height):
... print “volume is “, a*b*c
...
```

```
>>>
>>> getvol(length,width,height)
volume is 40
>>>
```

USER INPUT

If you working on a interactive program which need user input, it can be defines as in following example:

```
>>
>>> height = input("Enter Height ")
Enter Height 3
>>> width= input("Enter Width ")
Enter Width 4
>>> length= input("Enter Length ")
Enter Length 5
>>>
>>> getvol(length,width,height)
volume is 60
```

USING DATA FROM ANOTHER FUNCTION

As mentioned earlier, function data sharing is hierarchical to share data from one function to another, define the called function with parameter, it can be any name, doesn't have to be same name as shared.

In the example below a topfunc() is define which sets the value of x, second function midfunc() is define inside which takes x as parameter and print its value.

```
>>> def topfunc():
  ... x = 10                        ## set the value of x
  ... def midfunc(x):       ## second function takes x as argument
  ... print "printing from mid function", x
... midfunc(x)               ## call to mid function to print value
```

```
...
>>> topfunc()
printing from mid function 10
>>>
```

A more convinent way do this is to use functions under the scope of main function.

FUNCTION VALUE RETURN

Function can return a value using return statement in a function. Python does not let any function without a return value, even if a function doesn’t return anything. You will see “None” as returned value.

In all the function examples above a value is returned when a function is called. This value can be used for display or further processing in the program.

```
>>> def test(a,b):
... c=a*b
... return 'DONE'
... print c
...
>>>
>>> test(a,b)
'DONE'
>>>
```

Only return value is printed as it returned before the print command, anything to print or return value must be stated before return statement.

SPECIALIZED BUILT IN FUNCTIONS – FILTER, MAP AND REDUCE

filter()

filter is a built in function which prints the values based on a condition. It takes two arguments

1. Function – which evaluate a condition and return true or false status.
2. Iterator , a range , list or tuple to go over

The function processes the range of data using function and returns true or false, filter function then prints only the true returns only.

Define a function, myodds, with argument as x, to remove even numbers from the range 2 to 20

```
>>> def myodds(x):
>>> return x % 2 != 0
```

Invoke builtin function filter() with parameters of function myodds and a range

```
>>> filter(myodds, range(2, 10))
[3, 5, 7, 9]
```

Filter prints only values which return true , satisfying the condition in the function.

This would have been output without using filter function, see how filter only printed for true conditions.

```
>>> for i in range(10):
... print i
... myodds(i)
...
0
False
1
True
2
False
3
True
4
```

```
False
5
True
6
False
7
True
8
False
9
True
>>>
```

map()

Builtin function map() filters a sequence based on a function definition and returns a list of the return values. All values of the iterator list are run through function and results are displayed as a list.

Define a function add which takes two arguments and returns sum of the arguments and define a range of numbers. Map will process all the numbers of the sequence and give results in a list.

```
>>> def add(x, y): return x+y    ## Define a function to add two numbers.
>>> myseq = range(8)          ## Define a sequence
>>> map(add, myseq,myseq)    ## Invoke map function with two parameters
  [0, 2, 4, 6, 8, 10, 12, 14]
```

As the range of numbers are processed, each are added as
(0+0 1+1 2+2 3+3 4+4 5+5 6+6 7+7) and displayed as list .

reduce()

Reduce function is used to consolidate the returned values from a function and return only a single value.

```
>>> myseq=range(8)     ## define a number sequence
>>>myseq
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
>>> reduce(add, myseq)  ## Reduce the numbers to one consolidated
number
28
```

The result is calculated by adding 0+2+3+4+5+6+7 = 28.

Unless you have a very specific requirement to use reduce, there may be a builtin function to get the same results for example in this case sum() is an easier function to achieve the same result

```
>>> sum(myseq)
 28
>>>
```

12. CLASSES

Classes are used to create class objects based of a standard template. Multiple class objects in different class state can be created and accesses independently of each other. Classes provide features of Object Oriented Programming in python.

Notable features of a class are

1. A class is defined by a capitalized name
2. A class may have a comment enclosed in quotes.
3. The functions inside class are called method.
4. init method is first method and is used to create a initial state for the class

A sample class template may look like:

```
Class SampleClass:
    “””Comment sample functions “””
     __init__
      [Function 1]
      [Function2 ]
      [Variable1 ]
      [Variable2]
```

CLASSES OPERATIONS

Classes create class instance object by instantiation. Class objects support attributes reference and instantiation operation.

Instantiation is the class instance object creation by assignment

x=SampleClass()

Attribute reference is referencing functions/variables in the object

Function1.x , Variable1.x

CLASS METHODS

__init__() Method

Init method is used to create a class in a specific initial state. It initializes the arguments passed on during creating a class instance and establishes instance object as self.

self, used as first argument creates a class reference object. It is not a reserved keyword for python, it can be any other words as well but ‘self’ is used as generally accepted practice.

Let’s create a new class which can take two arguments and has a function to

```
>>>
>>> class GetArea:
... """Get Area from the parameters """
... def __init__(self,x,y):
... self.x = x
... self.y = y
... def area(self):
... return self.x * self.y
...
>>>
```

INSTANTIATION

In order to use GetArea it needs to be initialized and an object created for operation, this is called instantiation.

For example

```
>>x = GetArea (7,4)
```

Once created the functions or variables in the classes can be accessed using object reference

```
>>>print x.area()
28
```

Comments can be accesses using __doc__ and variables can be changed by assignment.

```
>>> x.__doc__
'Get Area from the parameters '
>>>
```

Creating different class states

Once a class object is created for class GetArea it can be used to create different class states based on input parameters.

For example to the area of rooms in house, you can simply use this class with parameters, without having to write a new function.

```
>>> diningroom=GetArea (7,4)
>>> kitchen=GetArea (7,8)
>>> livingroom=GetArea(10,10)
>>>
>>>
>>> print diningroom.area()  ## diningroom class state has 7 and 4
parameters.
28
>>> print kitchen.area()      ## kitchen class state has 7 and 8
parameters.
56
>>> print livingroom.area()   ## livingroom class state has 10 and 10
parameters.
100
```

All the class states can now be used to get the total area

```
>>> total=diningroom.area() + livingroom.area() + kitchen.area()
>>> print total
184
>>>
```

If you want to make a correction in any class parameter, instantiate it with new parameter

```
>>> diningroom=GetArea (9,4)
>>> print diningroom.area()
36
>>> total=diningroom.area() + livingroom.area() + kitchen.area()
>>> print total
192
>>>
```

CLASS INHERITANCE & DERIVED CLASSES

Classes can be inherited or may inherit methods and variable from other classes. A derived class can be created from a single or many base classes.

The syntax of declaring a derived class with a single base class is

```
class DerivedClass(BaseClass)
```

Multiple base classes can be specified which are scanned from left to right for accessing class resources.

```
class DerivedClass(BaseClass1, BaseClass2, BaseClass3)
```

Methods from base class can be referenced directly or overwritten in the Derived class

Let’s take example of of the class GetAraa we create earlier and use that as base class in new class perimeter, notice how the parimeter class only need function definition, rest all is inherited/derived from the GetArea class.

```
>>> class GetArea:        ## Class to get parameters and calculate area
... def __init__(self,x,y):
... self.x = x
... self.y = y
... def area(self):
... return self.x * self.y
```

```
...
>>>
>>> class parimeter(GetArea):  ##  Inherit GetArea class,
  ... def par(self):
... return (self.x + self.y)*2    ## Use values from GetArea() to get
parimeter.
...
>>>
>>>
>>>
>>> a=GetArea(2,7)         ## Instantiate GetArea Class with
parameters
>>> p=parimeter(2,7)        ## Instantiate parimeter Class with
parameters
>>>
>>> print a.area()          ## Area from GetArea class
14
>>> print p.par()           ##  Perimeter from perimeter class
18
>>> p=perimeter(8,5)        ## instantiate with new parameters to get
new values
>>> print p.par()
26
>>>
```

EXCEPTION CLASS

Exception class is a built in python class All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

You can create your own custom exceptions by using built in Exception class as base class and displaying a custom message in the derived class as

For example:

```
>>> class MyCustomError(Exception):
```

Learn more about exceptions and custom exceptions in the exception section.

13. MODULES

Key points about Python Modules

- Modules are written to provide additional functionality to a program.
- A module essentially contains executable code in the form of one or several functions.
- Any program can import modules and these functions become available to the program.
- Each module has its own private global symbol table for the functions defined inside that module.
- During development and for debugging purpose modules can be run as script on the command line and in interactive mode.
- Modules are compiled to produce a compiled object , .pyc files.
- Modules are included in the program using import statement. Python looks first for built in module then for a file with .py in the run directory and then look in the path defined in PYTHONPATH

Python has two types of module based on their availability - Standard Modules and Custom Modules.

STANDARD MODULES

Standard modules come packaged with standard Python installation as part of module library. These modules are written in C to provide system level access and other common functions.

Although these modules are built in python yet you have to import them to use it. After import you can use the functions available within the modules as

module_name.function_name

Complete list of available modules in python can be found using help command

```
>>>help('modules')   ### Displays list of modules available to import
Please wait a moment while I gather a list of all available modules...
BaseHTTPServer   antigravity   ihooks      setuptools
Bastion          anydbm        imageop     sgmllib
CGIHTTPServer    argparse      imaplib     sha
```

Help on a particular module provide details about the modules & the file name or built in type . A snippet for the help sys modules

```
>>>help('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.
```

Names defined in the modules

dir() is a built in function in python and it is used to get the names that a imported module defines. The output is a sorted list of strings.

Without arguments, dir() lists the names you have defined currently in the current python shell

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', '_subprocess',
'_winreg',
'add', 'div', 'minus', 'mult', 'mymath', 'nt', 'plus', 'sys']
>>>
```

With module name as an argument dir() gives list of functions in the module.

To demonstrate the module usage lets take one of the importamt module - sys module which process functions to provide system path, system prompt, ps1 & pst2 along with stdout, stdin and stderr etc.

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__',
'__package__', '__s
tderr__', '__stdin__', '__stdout__', '_clear_type_cache',
'_current_frames', '_g
etframe', '_mercurial', 'api_version', 'argv', 'builtin_module_names',
'byteorde
r', 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle', 'dont_
write_bytecode', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
'exec_prefix
', 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckint
erval', 'getdefaultencoding', 'getfilesystemencoding', 'getprofile',
'getrecursi
onlimit', 'getrefcount', 'getsizeof', 'gettrace', 'getwindowsversion',
'hexversi
on', 'last_traceback', 'last_type', 'last_value', 'long_info', 'maxint',
'maxsiz
e', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_c
ache', 'platform', 'prefix', 'ps1', 'ps2', 'py3kwarning', 'setcheckinterval',
's
etprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subve
rsion', 'version', 'version_info', 'warnoptions', 'winver']
```

Changing System prompt:

Using ps1 and ps2 functions in sys module you can change the primary and secondary prompts.

```
>>>import sys          ## import to load module
>>> sys.ps1           ## Show current primary prompt
```

```
'>>> '
>>> sys.ps2                          ## Show current secondary prompt
'... '
>>> sys.ps1 = 'Enter> '     ## Change primary prompt to Enter>
Enter>
Enter> sys.ps2 = '--> '      ## Change secondary prompt to -->

Enter>a=4
Enter> if a==4:           ## New primary prompt
--> print ‘Even Number’
-->                    ## New secondary prompt
Even Number
Enter>
```

Print & change system path

The variable sys.path is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH or from a built-in default if PYTHONPATH is not set. You can modify it using standard list operations such as append in the example below.

```
>>> sys.path
['', 'C:\\Python27\\python27.zip', 'C:\\Python27\\DLLs',
'C:\\Python27\\lib', 'C
:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Python27',
'C:\\P
ython27\\lib\\site-packages']
Enter>

>>> sys.path.append('c:\home\james')
Enter> sys.path
['', 'C:\\Python27\\python27.zip', 'C:\\Python27\\DLLs',
'C:\\Python27\\lib', 'C
:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Python27',
'C:\\P
ython27\\lib\\site-packages', 'c:\\home\\james']
Enter>
```

Similar to sys module, you can import other built in modules, list names and get help using dir functions, and start using them by <module name>.<function Name>

'BUILT-IN' MODULES

built-in functions and variables are not listed by dir() , these are defined in a standard module __builtins__ module. Being a module __builtins__ needs to be imported first to access its functions. __builtin__>>>import __builtin__

>>>dir(__builtin__)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']

>>>

CUSTOM MODULES

You can create a custom module to serve specific purposes. Any custom application will have its own custom modules developed for it.

Create a custom module

You can create custom modules and keep them in your own directory. As python searches modules only in the defined path, you have to check if directory path for module exists

```
>>> import sys
>>> print(sys.path)
['', 'C:\\Python27:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk',
'C:\\Python27', 'C:\\Python27\\lib\\site-packages']
        >>>
```

If the path is not there, it can be added using following command

```
>>> sys.path.append("C:\Python27\mymodules")
>>> print(sys.path)
['', 'C:\\Python27\\python27.zip', 'C:\\Python27\\DLLs',
'C:\\Python27\\lib', 'C
:\\Python27\\lib\\plat-win', 'C:\\Python27\\lib\\lib-tk', 'C:\\Python27',
'C:\\P
ython27\\lib\\site-packages', 'C:\\Python27\\mymodules']
>>>
```

To permanently add new path, add path in your OS profile file such as .profile file for the user or add it to user environment in Windows.

Lets create and use a custom module called mymath.py to do some math operations of plus , minus ,divide and multiply

Create a file and name it as mymath.py and put the following entries

```
def plus(x, y): return x+y
def minus(x, y): return x-y
def div(x, y): return x/y
def mult(x, y): return x*y
```

Load the module using import command

```
>>> import mymath
>>> mymath.plus(2,2)
4
>>>
```

Execute module functions with <module_name>.<function> (parameters)

```
>>> mymath.div(6,2)
3
>>> mymath.mult(6,2)
12
>>> mymath.plus(6,2)
8
>>> mymath.minus(6,2)
4
>>>
```

Reload module with reload command if you change something in mymath.py

```
>>> reload(mymath)
<module 'mymath' from 'mymath.py'>
```

Note that running import command just reloads already created module object so to reflect changes use reload().

Selective Import of functions - Using syntax - from <module> import <function name> , import two functions, plus & minus from mymath module.

```
>>> from mymath import plus, minus
>>> plus(4,5)
```

Import all functions using wild card * . Using syntax - from <module> import *

```
 >>> from mymodule import *
>>> addition(4,5)
```

14. PACKAGES

A collection of module for a common purpose is called a package. For example a payroll package or inventory management package may contain various required modules.

A package defines a package name, initializes package name and defines module directories. Multiple modules can be imported using a package name

__init__.py file

If you decide to keep modules in your own directory then python requires a specially named file , __init__.py , to be present in that directory. This file tells python to treat directory as python package directory containing modules.

__init__.py is loaded first when a import command is issued to load module from a specific directory, it can be an empty file or it can contain instruction to load only select sub modules instead of whole module. __all__ defines a list of modules to import if you issue an import using a wild card.
__all__=
['plusmodule.py','divmodule.py"minusmodule.py','multmodule.py']

Package Path

When importing the package, Python searches through the directories on sys.path looking for the package subdirectory. If package directory is not in path , you have to add it using sys.path.append

For example you can have a finance package with several modules arranged in following directory structure. __init__.py for each subdirectory may be either empty or has list of exposed modules or instructions to load specific functions only from a module.

```
finance/
   __init__.py

receipts/
        __init__.py
        vendors.py

  payables/
        __init__.py
        vendors.py
        customers.py
        ...
  audit/
        __init__.py
        yealyreports.py
        currentreport.py
   tax/
        __init__.py
        state.py
        federal.py
        sales.py
```

Each module directory have their own __init__.py file and
finance/payable//__init__.py

May have following entries as

```
__all__ = ["vendors", "customers"]
```

As stated earlier the __init__.py files are required to make Python treat the directories as containing packages. __init__.py can be an empty file or it can set the __all__ variable or execute custom import of functions.

As you may have noticed the import * command looks at __all__ declaration to get a list of modules to import. If you create new modules but don't declare in __all__ import command doesn't import new modules. Practically it is not good idea to use import * due to this reason.

PACKAGES IN MULTIPLE DIRECTORIES

__path__ is additional variable supported by packages and it holds the name of the directory containing __init__.py when __init__.py is executed. The __path_ variable can be modified inside __init__.py file to extend the module search path to the sub directories .

__path__.insert(/apps/finance/modules)

IMPORT MODULES FROM A PACKAGE

Specify complete package path along with module name to import

>>Import finance.payables.vendors

Alternatively select modules or functions can be imported as

>>>from finance.payables import vendors

Or for specific function, monthly, from payables.vendors module

>>>from finance.payables.vendors import monthly

General syntax for the above is

>>>from <package/module> import <module/function>

from has to be a package or module name and import can have a module or function name.

You should always use absolute imports because the name of the main module is always defined as "__main__", module in the symbol table.

IMPORTING ALL MODULES FROM A PACKAGE USING WILD CARD *

A package can be imported using following command

from finance.payables import *

However it is not considered as good practice because, unless updated in __all__ declaration of __init__.py new modules in a package will not be picked up .References

When there is a need to import submodules of sibling packages you can use absolute or relative path to import module. For relative path dot represents the current directory.

```
>>>from finance.payables import vendors
>>>from . import customers
>>>from ..tax import state
```

EXTERNAL MODULES

External modules, developed by others, can be installed using a python setup command file , setup.py . Modules packaged & built with Distutils can be installed with a simple syntax of :

```
$python setup.py install
C:/python/ python setup.py install
```

15. DATABASE

Python provides connectivity to different databases like Oracle, mysql , DB2 , MS access, MS SQLsybase etc using db modules.

The db modules are available at sourceforge download locations; you can download and install it on your local system such as https://sourceforge.net/projects/mysql-python/

SUPPORTED DATABASES

Following databases are supported using db modules, Ingres, MySQL, Oracle, PostgreSQL, Firebird, Informix, SAPDB, SQLServer, Access, Sybase
asql, GadFly, SQLite, ThinkSQL, MetaKit, ZODB, BerkeleyDB, KirbyBase, Durus, atop, buzhug,4Suiteserver, Oracle/Sleepycat, DBXML, Neo4j, buzhug, SnakeSQL

Python db modules are based on DB-API 2.0 specification described by (http://www.python.org/dev/peps/pep-0249/)

BUILT IN SQLITE MODULE

Python provides a built in sqlite3 module, compliant with the DB-API 2.0 specification. Sqlite3 uses nonstandard SQL and disk based database. Sqlite is a good way to get started with DB programming when you don't have access to other databases. You can later port the code to Database of your choice.

SQLLite mostly follows SQL syntax but have some additions and omission from standard SQL. A complete list of commands and syntax is available online at :
SQL As Understood By SQLite at http://www.sqlite.org/lang.html

INSTALLING AN EXTERNAL DB MODULE IN UNIX/LINUX

This example shows mysql db module installation but installation is similar for all other db modules.

Download MySQL-python-1.2.2.tar.gz from
http://sourceforge.net/projects/mysql-python/

```
$ gunzip MySQL-python-N.N.N.tar.gz < N is required/latest version builds )
$ tar -xvf MySQL-python-N.N.N.tar
$ cd MySQL-python-N.N.N
$ python setup.py build
$ python setup.py install
```

WORKING WITH DATABASE

Database operation basically consists of four steps as mentioned below.

First two steps are dependent on the database you are using and last two steps are common as they use the definition created in earlier steps

1. Import DB module.
2. Create a connection with the database.
3. Use SQL statements to get the data
4. Close the connection

Let us discuss these steps in details.

1. Import Module

Import modules for the database you want to work with, following are some examples of modules you can import

```
>>>import sqlite3    ## To work with Sqlite
>>>import cx_Oracle  ## To work with Oracle
```

```
>>>import MySQLdb  ## To work with Mysql
```

2. Create a Connection Object

Create a connection object, using available Database information such as user name, password, database host & name of the database

Sqlite

Creating connection object for Sqlite is simple , you need to provide either a filename to store data or mention memory to store data in the memory.

```
conn = sqlite3.connect('example.db') ## create & use example.db file to store data
conn = sqlite3.connect(':memory:')   ## Use memory to store data
```

Oracle

The connection information is provided in the format of - DB_username/password@DB Host/DB_name; 127.0.0.1 is local hosts and orcl is db name in the following example.

```
con = cx_Oracle.connect('DBUSER/DBPasswd@127.0.0.1/orcl')
```

MySQL

The connection information is provided in the format of host= DB Host, user= DB_username, passwd= DB_passwd, db= DB_name

For Example

```
con = MySQLdb.connect(host= DBHOST, user= DB_username, passwd= DB_passwd, db= DB_name )
```

Create a Cursor object

Once connection object is created, it can be used to create a cursor object for executing and database operation

```
cur = conn.cursor() ## create a cursor
```

Using cursor for database operation using execute

3. Create table & insert data

```
cur.execute('''CREATE TABLE stocks (date text, trans text, symbol
text, qty real, price real)''')
```

A. Insert a row of data

```
cur.execute("INSERT INTO stocks VALUES ('2006-01-
05','BUY','RHAT',100,35.14)")
```

B. Insert multiple rows of data

```
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),('2006-04-
05', 'BUY', 'MSFT', 1000, 72.00),('2006-04-06', 'SELL', 'IBM',
500, 53.00),] ## define input data


cur.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)',
purchases)
```

Note the use of **executemany** function to insert multiple rows of data

C. Save (commit) the changes

```
conn.commit()
```

D. Close the connection

```
conn.close()
```

4. Get the data from table

Reading data from the tables is also uses execute function and query to get data from the table

A. Define the database query

```
q = “SELECT * FROM stocks ”
```

B. Execute the defined query.

```
>>>cur.execute(q)
```

5. Extracting and using Cursor Data

Depending on the requirements, data from cursor object can be retrieved and printed using different functions

fetchone() to get a single matching row as in print c.fetchone()
getrows = cur. fetcone()

fetchmany() method returns a list of tuples, numRows parameter specifies number of rows to be returned

getrows = cur.fetchmany(numRows=10)

fetchall() to get all the matching rows as in print c.fetchall

```
>>> getrows=cur.fetchall()
>>> print getrows
>>> [“'2006-01-05','BUY','RHAT',100,35.14”]
>>>[(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14), (u'2006-03-28', u'BUY', u'IBM',1000.0,45.0), (u'2006-04-05', u'BUY', u'MSFT', 1000.0, 72.0), (u'2006-04-06', u'SELL',u'IBM',500.0, 53.0)]
```

Use **for iterator** to print rows of retrieved data as in following example

```
>>> for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
    print row

(u'2006-01-05', u'BUY', u'RHAT', 100, 35.14)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSFT', 1000, 72.0)
```

If you need to initiate a rollback in a script, the conn.rollback() method can be used.

SQLITE DATABASE

Functions, constants and connection object.

Here is the over view of the sqLite functions, constants, connections and objects to get started with sqlite if you want to know more about sqlite or don’t have oracle , mysql or any other database available.

Function Constants

- sqlite3.version
- sqlite3.version_info
- sqlite3.sqlite_version
- sqlite3.sqlite_version_info
- sqlite3.PARSE_DECLTYPES
- sqlite3.PARSE_COLNAMES
- sqlite3.connect(database[, timeout, detect_types, isolation_level, check_same_thread, factory, cached_statements])
- sqlite3.register_converter(typename, callable)
- sqlite3.register_adapter(type, callable)
- sqlite3.complete_statement(sql)
- sqlite3.enable_callback_tracebacks(flag)

Connection Objects

- class sqlite3.Connection

Connection attributes and methods:

- isolation_level
- cursor([cursorClass])
- commit()
- rollback()
- close()
- execute(sql[, parameters])
- executemany(sql[, parameters])
- executescript(sql_script)

- create_function(name, num_params, func)
- set_authorizer(authorizer_callback)
- set_progress_handler(handler, n)
- row_factory
- iterdump

Cursor Objects

- class sqlite3.Cursor

Cursor attributes and methods.

- execute(sql[, parameters])
- executemany(sql, seq_of_parameters)
- fetchone()
- fetchmany([size=cursor.arraysize])
- fetchall()
- rowcount
- lastrowid
- description

Row Objects

- class sqlite3.Row

16. DEBUG

Python pdb module provides an interactive debugger to debug python programs

The debugger allows you to

1. Set up conditional break points in the source code lines
2. Inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame.
3. Supports post-mortem debugging which can be called under program control.

DEBUG-MODE VARIABLES

Setting these variables only has an effect in a debug build of Python, that is, if Python was configured with the --with-pydebug build option.

PYTHONTHREADDEBUG

If set, Python will print threading debug info.

PYTHONDUMPREFS

If set, Python will dump objects and reference counts still alive after shutting down the interpreter.

PYTHONMALLOCSTATS

If set, Python will print memory allocation statistics every time a new object arena is created, and on shutdown.

Important points about debugger Commands & arguments

1. Commands are recognized by one or two letters, help , h , continue , c and arguments are separated by white space Optional arguments are enclosed in square brackets.

2. Entering a blank line repeats the last command entered. Exception: if the last command was a `list` command, the next 11 lines are listed.
3. Commands that the debugger doesn't recognize are assumed to be Python statements and are executed in the context of the program being debugged.
4. Python statements can also be prefixed with an exclamation point (`!`). This is a powerful way to inspect the program being debugged; it is even possible to change a variable or call a function. When an exception occurs in such a statement, the exception name is printed but the debugger's state is not changed.
5. Multiple commands may be entered on a single line, separated by `;;` . (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string.
6. The debugger supports aliases. Aliases can have parameters which allows one a certain level of adaptability to the context under examination.
7. If a file `.pdbrc` exists in the user's home directory or in the current directory, it is read in and executed as if it had been typed at the debugger prompt. This is particularly useful for aliases. If both files exist, the one in the home directory is read first and aliases defined there can be overridden by the local file.

USING PYTHON DEBUGGER, PDB

Python debug module can be used in two ways

1. **In interactive mode through import module**

pdb.run function of debugger allows to run script or statements under debugger control.

Debugger executes the statements or scripts until the point of crash. You get the option at the crash either to step line by line to see which line is causing the problem or you can continue running program further if errors are not fatal.

Syntax is : pdb.run(statement[, globals[, locals]])

Typical usage:

```
>>> import pdb
>>>pdb.run(“myscript.py”)
```

2. As Command line option

pdb.py can also be invoked as a script to debug other scripts. For example:

```
$python -m pdb myscript.py
```

on invoking pdb as script pdb it automatically enters the debugging mode if program exits abnormally.

If there is abnormal exit for program, pdb restarts the program to preserve the breakpoints and state of debugger.

PDB ENVIRONMENT

The default environment for pdb is the dictionary of __main__ module however additional environment can be specified using global and locals arguments while invoking pdb.run.

File .pdbrc stores to load run time environment and aliases every time you start a pdb shell. .pdbrc can be in user home directory or local directory.

A .pdbrc may look like this.

```
import rlcompleter
import pdb
pdb.Pdb.complete=rlcompleter.Completer(locals()).complete
import os
import sys
os.system("stty sane")
```

execfile(os.path.expanduser("~/.pdbrc.py"))

PDB COMMANDS

Pdb shell supports a number of commands which can be invoked as single letter or full command name
A question mark (?) , displays the available commands in pdb.

(Pdb) ?

(Pdb)

EOF	Bt	Cont	enable	Jump	Pp	run	unt
A	C	Continue	exit	L	Q	s	until
Alias	Cl	D	H	List	quit	step	up
Args	Clear	Debug	help	N	R	tbreak	w
B	Commands	Disable	ignore	next	restart	u	whatis
Break	Condition	Down	J	p	return	unalias	where

Miscellaneous help topics:
==========
exec pdb

Undocumented commands:
==========
retval rv

NAVIGATION COMMANDS

Commands used to move around the program being executed in the pdb, first or few first letter shows the commands below.

- **S** (step) - Step and stop at next line of current function or called function if next line calls a function

- **n** (next) – Continue execution of the next line , if a function is called on the next line it is executed.
- **unt** (until) - Continue execution until the a line number greater than the specified line number.
- **r** (return) - Continue execution until the current function returns.
- **c** (continue)) - Continue execution and stop only if a breakpoint is found.
- **j** (jump) lineno - Execute code by jumping back or forward to a line number. Some condition apply and you can't jump in arbitrary manner for example in the middle of for loop.

STATUS COMMANDS

Command telling about current status

- **l** (fist) [first[, last]] - List the 11 lines of code or 11 lines around a line specified , or a range of lines mentioned in argument
- **a** (args) - Print the argument list of the current function.
- **w** (where) - Print a stack trace . The most recent frame appears at the bottom and an arrow points to the current frame
- **d** (down) - Move the current stack trace frame to one level down to a new frame.
- **u** (up) - Move the current stack trace frame one level up in the stack trace to an older frame).

Other command in pdb

b(reak) [[filename:]lineno | function[, condition] - Set a break in the current file using lineno argument or at the first executable statement of a function . A break can be set in file other than current file and can be specified by file:linenum

```
(Pdb) b 3
Breakpoint 1 at c:\python27\myscript.py:3
(Pdb) b 5
Breakpoint 2 at c:\python27\myscript.py:5
```

Without argument it lists the detail s such as breakpoints, number of times it has been hit, the ignore count and associated condition.

```
(Pdb) b
Num Type    Disp Enb Where
1 breakpoint keep yes at c:\python27\myscript.py:3
2 breakpoint keep yes at c:\python27\myscript.py:5
(Pdb)
```

tbreak [[filename:]lineno | function[, condition]]
temporary break , A onetime use temporary breakpoint can be set with same arguments as break , it is removed automatically when at its first hit .

```
Pdb) tbreak 4
Breakpoint 3 at c:\python27\myscript.py:4
```

```
(Pdb) b
Num Type    Disp Enb Where
1 breakpoint keep yes at c:\python27\myscript.py:3
2 breakpoint keep yes at c:\python27\myscript.py:5
3 breakpoint del yes at c:\python27\myscript.py:4
(Pdb)
```

Note del in the second column to delete it after first use.

disable [bpnumber [bpnumber ...]]

Disables single or multiple breakpoints. Multiple breakpoints are specified as space separated list. Disabling does not remove the breakpoint only it does not break at the set point and can be enabled later if needed.

```
 (Pdb) disable 1
(Pdb) b
Num Type    Disp Enb Where
1 breakpoint keep no  at c:\python27\myscript.py:3
2 breakpoint keep yes at c:\python27\myscript.py:5
3 breakpoint del yes  at c:\python27\myscript.py:4
(Pdb)
```

Note Enb flag set to no above.

enable [bpnumber [bpnumber ...]]

Enables single or multiple breakpoints disabled by disable command.
To enable breakpoint 1 disabled inn earlier example:

```
(Pdb) enable 1
(Pdb) b
Num Type    Disp Enb Where
1 breakpoint keep yes at c:\python27\myscript.py:3
2 breakpoint keep yes at c:\python27\myscript.py:5
3 breakpoint del yes at c:\python27\myscript.py:4
(Pdb)
```

Note Enb flag set to yes above.

ignore bpnumber [count]

Ignore a breakpoint for a specified number of counts. The count is decremented on each crossing and breakpoint becomes active when count becomes zero.

```
 (Pdb) ignore 2 2

Will ignore next 2 crossings of breakpoint 2.
(Pdb)
```

condition bpnumber [condition]

A condition can be applied to a breakpoint. The condition should be true to execute that breakpoint

```
(Pdb) condition 2 < 0
(Pdb) b
Num Type    Disp Enb Where
1 breakpoint keep yes at c:\python27\myscript.py:3
2 breakpoint keep yes at c:\python27\myscript.py:5
    stop only if < 0
```

ignore next 2 hits
3 breakpoint del yes at c:\python27\myscript.py:4

commands [bpnumber]

A list of commands can be specified for a breakpoint. A new com (command) shell is opened to enter the command and a line with ‘end’ marks the end of commands.

An example:

(Pdb) commands 1
(com) print "BREAK 1 ----"
(com) end
(Pdb)

run [args ...]

it restarts a debugged Python program keeping History, breakpoints, actions and debugger options. If arguments are supplied they are used as new arguments in sys.argv
restart is an alias of run.

q(uit)

Quit from the debugger by aborting the program and going back to command prompt.

SETTING A BREAKPOINT INSIDE A PROGRAM

pdb.set_trace() is used to hard-code a breakpoint at a given point in a program. Typical usage to break into the debugger from a running program is to insert pdb.set_trace()

Let’s see this by modifying our script and putting a trace start. The program will enter debug mode only if trace is set before the line causing program to crash.
myscript.py

```
import pdb;
print "line 1" ;
```

```
print "line 2" ;
print "line 3" ;
pdb.set_trace()
print "line 4" ;
print name ;
print "line 5" ;
print "line 6" ;
```

If pdb.set_trace() is set at “line 5” it won’t enter debug mode as program would already came out before that due to exception.

Output

```
C:\Python27>python myscript.py
line 1
line 2
line 3
> c:\python27\myscript.py(6)<module>()
-> print "line 4" ;
(Pdb) step
line 4
> c:\python27\myscript.py(7)<module>()
-> print name ;
  (Pdb)
NameError: "name 'name' is not defined"
```

at the location you want to break into the debugger. You can then step through the code using cont or step following this statement, and

INSPECT A CRASHED PROGRAM

pdb.pm() function enables post-mortem debugging of the trace back .

Let’s run our script inside interactive shell and then do a post mortem using pdb

The run* functions and set_trace() are aliases for instantiating the Pdb class and calling the method of the same name. If you want to access further features, you have to do this yourself:

```
>>> import pdb
```

```
>>> import myscript
line 1
line 2
line 3
line 4
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "myscript.py", line 6, in <module>
  print name ;
NameError: name 'name' is not defined
                ## Program crashed and returned to prompt
>>> pdb.pm()          ## Run pdb postmortem
> c:\python27\myscript.py(6)<module>()
-> print name ; ##Location where program crashed
(Pdb) c                 ## c or step will take to prompt as program has
already crashed
>>>
```

Let’s look at the output in detail

Program executed up to line 4 and printed output

line 4

It failed on next line throwing an exception

```
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "myscript.py", line 6, in <module>
  print name ;
NameError: name 'name' is not defined      ## Program crashed and
returned to prompt
>>>
```

Run pdb post mortem to find out where program crashed - check

```
>>> pdb.pm()                              ### Run pdb postmortem
> c:\python27\myscript.py(6)<module>()
-> print name ;                ### Location where program crashed
(Pdb) c   ## c or step will take to prompt as program has already
crashed
```

>>>

POST MORTEM USING TRACE BACK OBJECT

If you have set a trace back in program, trace back object can be given as an argument to start post-mortem . However If no trace back is given, it uses the one of the exception that is currently being handled

Syntax is **pdb.** post**_mortem** ([traceback])

```
trbk = sys.exc_info()
pdb.post_mortem(trbk)
```

MORE USAGES OF PDB

1. Evaluate the expression using p **db.runeval and** returns the value of the expression

Syntax: pdb.runeval(expression[, globals[, locals]])

It evaluates the expression (given as a string) under debugger control. When runeval() returns, it returns the value of the expression. Otherwise this function is similar to run().

2. Call the function - It Call the function returns whatever the function call returned , syntax is

pdb.runcall (function[, argument, ...])

Call the function (a function or method object, not a string) with the given arguments. When runcall() returns, it returns whatever the function call returned. The debugger prompt appears as soon as the function is entered.

17. PYTHON & JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format based on a subset of JavaScript syntax. Python provides ready to use functions to encode python data to json and decode json data back to python.

Following are the main commands to handle json in python.

1. Json.dumps : Used for json encoding in Python , It takes the python list enclosed in [ ] , python dictionary enclosed in { } as input and convert it in to json object.
2. Json.dump reads the input directly from file for json encoding.
3. Json.loads : Used for decoding json object and convert it into python list or dictionary and to access data.
4. Json.load reads the input directly from file for json decoding.

Bothe the commands looks similar but notice that singular names of commands work on files and plural versions, json.load, json.dumps works on input objetxs

JSON ENCODING

json.dumps converts lists or dictionaries in to a json object.

Here is some key point to note for json encoding

1. Keys in key/value pairs of JSON are always of the type string.
2. On conversion to JSON, all the keys of the dictionary are converted to strings.
3. After multiple conversions the dictionary may not be equal to original one.
4. There is no serialization like pickle so multiple

imports can make json file invalid.

Table below shows the convention used when changing data from json to python and reverse, according to following convention.

JSON	**Python**
Object	Dict
Array	List
String	Unicode
number (int)	int, long
number (real)	Float
True	True
False	False
Null	None

Json.dump command Syntax:

json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)

If ensure_ascii is False would lead to have non ascii chacaters resuting in a unicode output.

Example

Here is a simple example of json.dumps function taking python dictionary data as input and returning a JSON string.

json.dumps converts json dictionary data in to string returning the dictionary object as a string object as in the example below.

```
>>> a = {'cars':5,’bikes’:10}
        >>> b=json.dumps(a)
        >>>b
        '{"cars": 5, "bikes": 10}'
        >>>
```

Json.dumps is to create the object and to retrieve the data you have to use json.dump as describe under decoding json section.

Lets confirm the Object type, if it is string or not with following command, a, b and c are the objects to test

```
>>> isinstance(a, basestring)
False
>>> isinstance(b, basestring)
True
```

PRETTY PRINTING

Pretty printing is used to show output json in a readable format. For big json files it becomes very difficulty to glance over the data and its structure. Pretty printing helps in by providing a readable format o json data.

Consider this sample json string.

```
>>>d={"menu":{"id":"file","value":"File","popup":{"menuitem":
[{"value":"New","onclick":"CreateNewDoc()"},
{"value":"Open","onclick":"OpenDoc()"},
{"value":"Close","onclick":"CloseDoc()"}]}}}
```

Simple json.dumps prints the string as it in the following example.

```
>>> print json.dumps(d)

{"menu": {"popup": {"menuitem": [{"onclick": "CreateNewDoc()",
"value": "New"}, {"onclick": "OpenDoc()", "value": "Open"},
{"onclick": "CloseDoc()", "value": "Close"}]}, "id": "file", "value":
"File"}}
```

Pretty Prining, - With additional options you can print the json in a pretty readable format**.**

```
>>> print json.dumps(d, sort_keys=True, indent=3, separators=(',', ':
'))
{
  "menu": {
   "id": "file",
```

```
    "popup": {
     "menuitem": [
       {
        "onclick": "CreateNewDoc()",
        "value": "New"
       },
       {
        "onclick": "OpenDoc()",
        "value": "Open"
       },
       {
        "onclick": "CloseDoc()",
        "value": "Close"
       }
     ]
    },
    "value": "File"
   }
}
>>>
```

The default item separator is ', ' , the output might include trailing whitespace when indent is specified. You can use separators=(',', ': ') to avoid this.

DECODING JSON OBJECTS

Json.loads convers json object to python dictionary and allows you to access the values of keys. Using the example we used earlier

```
>>> a = {'cars':5,'bikes':10}
  >>> b=json.dumps(a)
  >>>b
  '{"cars": 5, "bikes": 10}'
  >>>c=json.loads(b)      ### Convert json object to python dictionary
  >>> c
  {u'cars': 5, u'bikes': 10}   ## printing shows python data
```

```
>>>c[“cars”]          ## you can access the value of keys now
5
>>>
>>>
>>> d={"menu":{"id":"file","value":"File","popup":{"menuitem":
[{"value":"New","onclick":"CreateNewDoc()"},
{"value":"Open","onclick":"OpenDoc()"},
{"value":"Close","onclick":"CloseDoc()"}]}}}
>>>
>>> n=json.dumps(d)        ### convert to json string object
>>> nav=json.loads(n)        ### convert to python dictionary to
retrieve the keys/values
>>>
>>> nav["menu"]["id"]        ### key/values can be retrieved using
keynames & hierarchial notation.
u'file'
>>> nav["menu"]["popup"]["menuitem"]
[{u'value': u'New', u'onclick': u'CreateNewDoc()'}, {u'value': u'Open',
u'onclick': u'OpenDoc()'}, {u'value': u'Close', u'onclick':
u'CloseDoc()'}]
>>>
```

Json.dump() and json.load()

There functions get input fron file and write input to file. We wil use variable d to write to a file and then read from file:

```
>>>with open("sample.json","w") as f:    ### Open file to write data
. . . json.dump(d, f)                ### Wtite data d to the file f
>>>with open("sample.json","r") as f:    ### Open file to read data
. . . d = json.load(f)                ### Read data from the file f
```

Key value retrieval is same as discussed before using keys and hierarchical structure.

18. ERRORS AND EXCEPTIO NS

Programs often crash when they encounter an unhandled exception or syntax errors in the statements and functions used in the program. Let's see how python handles Syntax errors and other exceptions.

SYNTAX ERRORS

Syntax is the arrangement of words and phrases to create a well-formed sentence in a language. Python has its own grammar and rules to create statements which its interpreter can understand and execute.

- Syntax errors are most common errors and can results due to any of the following reasons Missing comma, full stop or operators
- Missing or misspelled keywords
- Sentence is not correct grammatically as per language rules

For a syntax error file name, line number and a pointer (^) to error location is provided

For example consider this simple declaration of a array n= [10,20,30,40,50,60,70,80,90]

```
>>> n= [10,20,30,40,50,60,70,80,90]
>>> print n
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

If you mistype one of the zeros with letter 'o' in 90 , it will result in syntax error .

```
>>> n= [10,20,30,40,50,60,70,80,9o]
File "<stdin>", line 1
 >>> n= [10,20,30,40,50,60,70,80,9o]
```

```
                                ^
  SyntaxError: invalid syntax
>>>
```

In the above statement numbers are defined properly but 9o is not a number , it is not a quoted string and neither a variable as it is not defined .

Corrective actions can be

1. **Define it as number**

```
n= [10,20,30,40,50,60,70,80,90]
```

2. **Define 9o as string**

```
n= [10,20,30,40,50,60,70,80,’9o’]
```

3. **Define o as variable**

```
o=”aaa”
>>> n= [10,20,30,40,50,60,70,80,o ]
>>> print n
[10, 20, 30, 40, 50, 60, 70, 80, 'aaaa']
>>>
```

19. EXCEPTIONS

An exception is thrown when statements are grammatically correct but operation is not permitted. These may or may not be fatal and programs needs to have code written to handle them.

Python has built exception to handle different types. Following diagram shows types & hierarchy for system exceptions and warnings

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
    |  +-- BufferError
    |  +-- ArithmeticError
    |  |  +-- FloatingPointError
    |  |  +-- OverflowError
    |  |  +-- ZeroDivisionError
    |  +-- AssertionError
    |  +-- AttributeError
    |  +-- EnvironmentError
    |  |  +-- IOError
    |  |  +-- OSError
    |  |     +-- WindowsError (Windows)
    |  |     +-- VMSError (VMS)
    |  +-- EOFError
    |  +-- ImportError
    |  +-- LookupError
    |  |  +-- IndexError
    |  |  +-- KeyError
```

```
| +-- MemoryError
| +-- NameError
| |  +-- UnboundLocalError
| +-- ReferenceError
| +-- RuntimeError
| |  +-- NotImplementedError
| +-- SyntaxError
| |  +-- IndentationError
| |    +-- TabError
| +-- SystemError
| +-- TypeError
| +-- ValueError
|    +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
   +-- DeprecationWarning
   +-- PendingDeprecationWarning
   +-- RuntimeWarning
   +-- SyntaxWarning
   +-- UserWarning
   +-- FutureWarning
         +-- ImportWarning
         +-- UnicodeWarning
         +-- BytesWarning
```

When encountered in a program, exception will show one of the above classifications with a brief description, file name and line number.

For example

A common exception you can encounter is ZeroDivisionError when a calculated value in a program reached zero value and program still attempt to divide the divisor by zero .

How many items you can buy with ten dollars, in this small program you can do calculation with decreasing value of a range, the range ultimately becomes zero and program exists with Zero division error.

```
>>> for i in reversed(range(10)):
... print 'With $',i,'You can buy ', 10/i,'items'
...
With $ 9 You can buy 1 items
With $ 8 You can buy 1 items
With $ 7 You can buy 1 items
With $ 6 You can buy 1 items
With $ 5 You can buy 2 items
With $ 4 You can buy 2 items
With $ 3 You can buy 3 items
With $ 2 You can buy 5 items
With $ 1 You can buy 10 items
With $ 0 You can buy
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

Another common exception is name error when you try to use some undefined value as variable.

```
>>>
>>> print zz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'zz' is not defined
>>>
```

Besides these exceptions you can encounter any of the exception mentioned in the exception tree.

Custom Exceptions

Besides built in system exception you can create your own custom exception as class objects. These can be called in a similar fashion as system generated exception

HANDLING EXCEPTIONS

Python offers a good way to handle exception using *while* and *try* functions.

Basic operation involves trying some operation and catching specific exception and print response to the exception.

In the example below

Start a while loop to keep on asking for a number until you get a number, then break out of the while loop.

If the value entered in not the type number then loop continues by printing the custom exception message in the body and continues the loop by asking for number again.

While is a infinite loop which exists only when condition is true or it gets a user interrupted in the form of Control C or control break or exception.

```
>>while True:
  try:
    x = int(raw_input("Enter a number: "))
    break
  except ValueError:
    print "Error: Only numbers are accepted. "
```

The error message in the above example is the custom error message specified by you.

To capture system message assign it to a variable and print it along with your custom message as except <exceptionType> as <variable to capture system error message >:

```
>>> while True:
...   try:
...     x = int(raw_input("Enter a number: "))
...     break
...   except ValueError as errordetails:
...     print "Error: Only numbers are accepted. ", errordetails
```

...
Enter a number: e
Error: Only numbers are accepted. invalid literal for int() with base 10: 'e'
Enter a number: abc
Error: Only numbers are accepted. invalid literal for int() with base 10: 'abc'
Enter a number:

HANDLING MULTIPLE EXCEPTIONS

Multiple exceptions can be handled in two ways either by using multiple exception directives in a try clause or using a catch all exception handlers to print a custom error message

Multiple exceptions can be handled in a single line as :.

except (SystemError, TypeError, ValueError)
print “An error has occurred. Try again “

Multiple exceptions can be used separately in a try clause to catch and print custom message according to type of exception. Note that the exceptions mentioned are valid only for the try clause it is contained in.

except <exception type >:
  print "msg for exception 1”
except <exception type >:
   print "msg for exception 1”
except:
print "Unexpected error” ## catch all if none of the previous exeption matches.

try can be used with an else clause in the format below similar to if , then , else clauses to print or do something is there is no exception.

try
<do Something >
exception
<do something if an exception>
else
<do something if successful run>

Exception handling for called functions can be handled by parent try clause. If a function don’t have a exception handling in itself, calling try clause is able to handle and print the error message

Following example we have a function with an exception and being called in by a later try clause.

```
>>> def justerror():
...   x = 1/0
...
>>> try:
...   justerror()
... except ZeroDivisionError as detail:
...   print ‘Got it from justerror function :', detail
...
Got it from justerror function: integer division or modulo by zero
```

RAISING EXCEPTIONS

So far we have seen that we can capture exceptions occurring in a program. What if we have a a custom program and that does not trigger a built in exception but we want it to trigger an exception. Raise statement is useful in raising an exception or rearising an existing exception.

For example if a certain environmental variable is not setup for your program you can raise your own EnvironmentError exception.

```
>>> raise EnvironmentError ('Environment not setup correctly')
Traceback (most recent call last):
 File "<stdin>", line 1, in?
EnvironmentError: 'Environment not setup correctly'
```

A simple raise statement after exception clause will re-raise the earlier exception with same custom message

```
>>> try:
...   raise EnvironmentError ('Environment not setup correctly')
... except EnvironmentError
...   print 'Environment not setup correctly'
...   raise
```

```
...
Traceback (most recent call last): ## Most recent exception
 File "<stdin>", line 2, in?
EnvironmentError: 'Environment not setup correctly'
```

USER-DEFINED EXCEPTIONS

You can create & name your own exceptions as class objects derived from the class Exception, Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

For example:

```
>>> class MyError(Exception):
...   def __init__(self, value):
...    self.value = value
...   def __str__(self):
...    return repr(self.value)
...
>>> try:
...   raise MyError(2*2)
... except MyError as err:
...   print 'My exception occurred, value:', err.value
...
My exception occurred, value: 4
>>> raise MyError('my custom error msg') ## Call custom
exception with your
                                    own message to print
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
__main__.MyError: 'my custom error msg'
```

DEFINING CLEAN-UP ACTIONS

Try statement has an option clause ‘finally ‘ which is executed whenever try statement exists normally or abnormally. Generally it it is used to close files and connections opened during the execution of try.

```
>>> try:
```

```
...   raise NameError
... finally:
.. .  print “Executing finally, Got name error , exiting ! “
```

WITH STATEMENT AS WRAPPER

with statement works as a wrapper around try, except, finally or any other action. It results in loading a context manager and invoking entry and exist methods at start and end irrespective or normal or abnormal exit.

For example:

```
with open("myfile") as f:
  for line in f:
    print line,
```

Here file handle f defined with ‘with ‘ statement works as wrapper , whenever the for statement loop closes and comes out , the file handle is also closed.

20. INTERACTING WITH SYSTEM

Python provides lots of functions and modules to interact with operating system. This is very useful if your program need to work with OS to collect the data or work with files and processes. Built in *OS module* provides most of the OS level functions.

Let's discuss OS and other related modules in detail.

OS MODULE

OS & signal modules allow you to interact with operating system & its processes. Using functions in OS module you can execute OS commands from within python program.

In order to access functions in these module, you have to import these functions

```
>>>import os
```

OS level functions are dependent on the platform so most of them may work on Unix/Linux platforms and some may not work in windows platforms.

Functions are executed as os.<FUNCTION NAME>(<ARG/S>). Some don't need arguments as they return status or information like path, current working directory, process id etc., for example :

```
os.getcwd()  - Returns Current working Directory name
os.pid()     - Returns process id
os.getuid()  - Returns User Id
```

Other OS functions require a single or more arguments depending on the operation

Few examples:

os.rmdir(path) - Need a directory name to remove, /tmp/test

```
>>>os.rmdir('/tmp/test’)
```

os.renames(old, new) - Need old and new file names as oldfile and newfile in example

```
>>>os.renames('/tmp/oldfile', '/tmp/newefile')
```

os.kill(pid, sig) - Need a pid and signal to send as 2781, signal.SIGTERM in example

```
>>>os.kill(2781, signal.SIGTERM)
```

os.fchmod(fd, mode) - Need a file name and mode as newfile , 755 in example

```
>>>os. fchmod ('/tmp/newfile',755)
```

os.fchown(fd, uid, gid) - Need filename, userid , groupid as newfile,George,Others in example

```
>>>os. fdchown ('/tmp/newfile',200 ,15)
```

-1 is used to keep the current value for user and group id

```
>>>os. fdchown ('/tmp/newfile',400 ,-1 )  only user is changed to 400
and group id is unchanged.
```

LIST OF FUNCTIONS AVAILABLE IN OS MODULE

abort, access, altsep, chdir, chmod, chown, chroot, close, closerange, confstr, confstr_names, ctermid, curdir, defpath, devnull, dup, dup2, environ, errno, error, execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, extsep, fchdir, fchmod, fchown, fdatasync, fdopen, fork, forkpty, fpathconf, fstat, fstatvfs, fsync, ftruncate, getcwd, getcwdu, getegid, getenv, geteuid, getgid, getgroups, getloadavg, getlogin, getpgid, getpgrp, getpid, getppid, getresgid, getresuid, getsid, getuid, initgroups, isatty, kill, killpg,

lchown, linesep, link, listdir, lseek, lstat, major, makedev, makedirs, minor, mkdir, mkfifo, mknod, name, nice, open, openpty, pardir, path, pathconf, pathconf_names, pathsep, pipe, popen, popen2, popen3, popen4, putenv, read, readlink, remove, removedirs, rename, renames, rmdir, sep, setegid, seteuid, setgid, setgroups, setpgid, setpgrp, setregid, setresgid, setresuid, setreuid, setsid, setuid, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, stat, stat_float_times, stat_result, statvfs, statvfs_result, strerror, symlink, sys, sysconf, sysconf_names, system, tcgetpgrp, tcsetpgrp, tempnam, times, tmpfile, tmpnam, ttyname, umask, uname, unlink, unsetenv, urandom, utime, wait, wait3, wait4, waitpid, 0walk, write

SIGNAL MODULE

Signals are one of the ways to communicate with running process. Signal functions are implemented with Signal module and used in conjunction with os module function – os.kill to send signal to a process.

Basic syntax of os.kill is os.kill(PID SIGNAL)

For example to kill a process id of 2772 :

```
>>>import os
>>>import signal
>>>os.kill( 2772, signal.SIGTERM)
```

Here are the signals described in the original POSIX.1-1990 standard which are common for many Unix / Linux implementations.

Signal	Value	Description
SIGHUP	1	Hang-up Signal
SIGINT	2	Interrupt Signal
SIGQUIT	3	Quit Signal
SIGILL	4	Illegal Instruction Signal
SIGABRT	6	Abort signal
SIGFPE	8	Floating point exception Signal
SIGKILL	9	Kill signal

SIGSEGV	11	Invalid memory reference Signal
SIGPIPE	13	Broken pipe signal
SIGALRM	14	Timer Alarm signal
SIGTERM	15	Termination signal
SIGUSR1	30,10,16	User-defined signal 1
SIGUSR2	31,12,17	User-defined signal 2
SIGCHLD	20,17,18	Child stopped or terminated Signal
SIGCONT	19,18,25	Continue if stopped Signal
SIGSTOP	17,19,23	Stop process Signal
SIGTSTP	18,20,24	Stop typed at terminal Signal
SIGTTIN	21,21,26	Terminal input for background process Signal
SIGTTOU	22,22,27	Terminal output for background process Signal

The signals SIGKILL and SIGSTOP cannot be caught, blocked or Ignored.

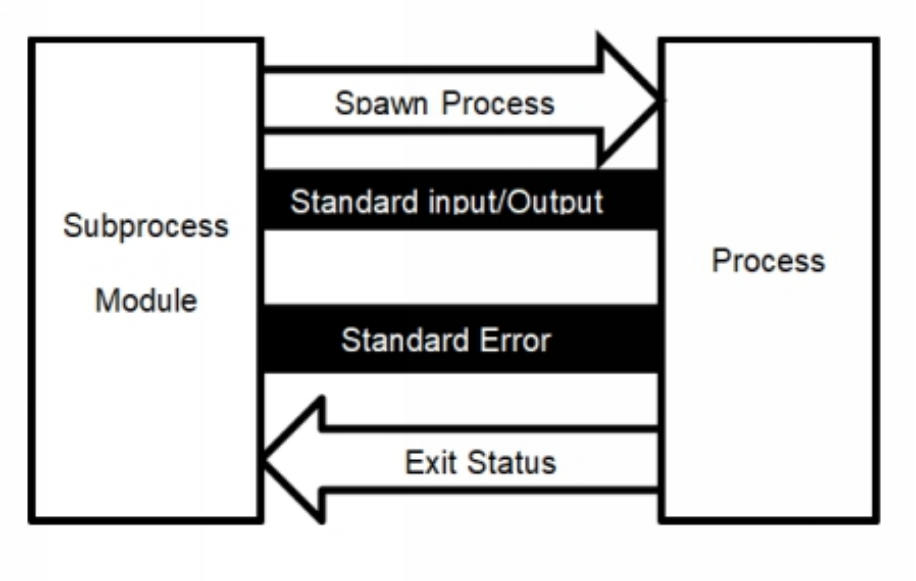
SUBPROCESS MODULE

The subprocess module allows you to interact with system by letting you run commands and capturing their output, errors and exist status. Functions in subprocess module are available after importing subprocess module

```
>>>import subprocess
```

This is how a subprocess module functions

1. Module Spawns process
2. Connects to its input, Output and Error Module
3. Gets the Exist Status of process



This module has several functions to interact with system which can be accessed using module and function name.

```
>>> dir(subprocess)
['CalledProcessError', 'MAXFD', 'PIPE', 'Popen', 'STDOUT',
'_PIPE_BUF', '__all__', '__builtins__', '__doc__', '__file__',
'__name__', '__package__', '_active', '_args_from_interpreter_flags',
'_cleanup', '_demo_posix', '_demo_windows', '_eintr_retry_call',
'_has_poll', 'call', 'check_call', 'check_output', 'errno', 'fcntl', 'gc',
'list2cmdline', 'mswindows', 'os', 'pickle', 'select', 'signal', 'sys',
'traceback', 'types']
>>>
```

Here are some important functions along with some examples

SUBPROCESS.CALL

This is used to run a command and specify arguments

Syntax to call a command:
subprocess.call(cmd , args , stdin=None, stdout=None, stderr=None, shell=False/True)

Command & arguments needs to be in the form of list, enclosed by [ ] and items separated by comma. For Items in the list first item is considered as executable command and rest as command line arguments.

```
>>> subprocess.call(["ls" , "-l" ]);
```

Another way is to specify command with arguments as single entity and make shell true, which is interpreted by shell and executed correctly

```
>>> subprocess.call('ls -l', shell=True);
```

However in the example above if shell is not defined then the function tries to look for a executable by the name of 'ls –l' , being the first item in the list , and the call fails to execute.

Some important points about shell=True

- On Unix the shell defaults to /bin/sh.
- On Windows COMSPEC environment variable defines the default shell , normally cmd.exe
- On windows Shell=True is needed only to run built in commands and not the external or batch commands.
- shell=True is a potential security loop hole if input commands are being accepted without validations. Malicious commands can be injected and executed using this method by any untrusted party.

SUBPROCESS.CHECK_CALL

This function checks for call , if successful , executes it and display the results , if fails raise CalledProcessError(retcode, cmd) exception returning command & exit status code.

```
>>> subprocess.check_call(["lsl", "-l" ], shell=True)
-l: lsl: command not found
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
```

```
 File "/usr/lib64/python2.7/subprocess.py", line 542, in
check_call
  raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['lsl', '-l']' returned
non-zero exit status 127
```

SUBPROCESS.CHECK_OUTPUT

This function run command with arguments and returns the output of command as a byte string.

If the return code was non-zero it raises a CalledProcessError. The CalledProcessError object will have the return code in the return code attribute and any output in the output attribute.

Following examples shows a successful and error condition execution of check_output

```
>>> subprocess.check_output("ls", “-l”, shell=True)

>>> subprocess.check_output("dir", shell=True)   ## on Windows
```

POPEN CLASS

Popen Class in subprocess module allows you to create new child processes, use them by joining in pipe , redirecting standard input /outputs/errors to PIPEs, checking status and sending signals to kill a process

Syntax:

Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=False, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0)

In its simplest form, Popen can be used to run commands in shell and send output to a PIPE .

```
>>> from subprocess import PIPE,Popen
>>> a=Popen("ls", shell=True,stdout=PIPE)
```

The object can be read using a for loop or communicate() method described below.

Here are methods of Popen class used in process management.

popen.communicate( )

popen.communicate Interacts with process and performs the following functions :

- communicate() function communicates with the process & returns a tuple (stdoutdata, stderrdata).
- Send data to stdin. Optionally you can send a input data as string to the child process using input option.
- Read data from stdout and stderr, until end-of-file is reached.
- Wait for process to terminate.
- Use it carefully, stdout / stderr data is stored in memory so it may cause problems if data size is huge.

It is invoked as <process object>.communicate() and examples are covered in following sections.

Popen.stdout

Popen.stdout method provides output of the child process if stdout argument is PIPE

```
>>> a=Popen("ls",stdout=PIPE)
>>> a.communicate()
('anaconda\naudit\nboot.log\nbtmp\nbtmp-
20160313\nchrony\nXorg.0.log.old\nXorg.1.log\nXorg.1.log.old\nyu
m.log\nyum.log-20141127\nyum.log-20150325\nyum.log-
20160126\n', None)
>>>
```

Note that communicate() return a tuple of standard output and standard error, assigned to [0] & [1] element of communicate () array.
If you just want to capture standard output use

```
>>> a.communicate()[0]
'anaconda\naudit\nboot.log\nbtmp\nbtmp-
20160313\nchrony\nXorg.0.log.old\nXorg.1.log\nXorg.1.log.old\nyu
m.log\nyum.log-20141127\nyum.log-20150325\nyum.log-
20160126\n'
>>>
```

POPEN.STDIN

Popen.stdin function provides input to a process if stdin argument is PIPE. This is generally used to pass the output of one process as input of other processes using pipe.

The is the sequence of activity:

Process 1 -> output ->Input ->Process 2

Following example shows how two processes , a & b, are piped together to perform a common function by taking output from first process and processing that output in the second function.

```
>>> a=Popen("ls",stdout=PIPE)
>>> b=Popen(["grep","Xorg"],stdin=a.stdout, stdout=PIPE )
>>> b.communicate()
('Xorg.0.log\nXorg.0.log.old\nXorg.1.log\nXorg.1.log.old\n', None)
>>>
```

POPEN.STDERR

Popen.stderr function provides output of the child process errors if stderr argument is PIPE

```
>>> a=Popen('ls -l z* ',shell=True , stderr=PIPE,stdout=PIPE)
>>> a.communicate()
('', 'ls: cannot access z*: No such file or directory\n')
>>>
```

communicate return a tuple of standard output and standard error, Notice the result tuple has empty stdout & populated stderr in this case.

SPECIFY A ENVIRONMENT

External environment can be specified using env keyword in Popen construction.

```
>>> a=Popen("ls", shell=True, env={"PATH": "/tmp"})
```

OTHER FUNCTIONS IN POPEN CLASS

PROCESS STATUS

Popen.poll() - Checks if child process has terminated & then set, return returncode.

```
>>> a.poll()
0
```

Popen.wait() - Waits for child process to terminate & set, return code .

```
>>> a.wait()
0
```

Popen.detail - gets the process ID of the child process or spawned shell if

shell is True.

```
>>> a.pid
3191
```

Popen.returncode - Gets the child return code set by poll() & wait

STOPPING /SIGNALING PROCESS

Popen.send_signal(signal) - Sends the signal to the child. More on signals in OS module.

```
>>> a.send_signal(signal.SIGTERM)
```

Popen.terminate() - Terminate child process

Popen.kill() - Kills the child process.

HELPFUL RESOURCES :

python language home for latest news, updates and downloads
https://www.python.org/

python download page
https://www.python.org/downloads/

python FAQ for 2.7 version
https://docs.python.org/2.7/faq/

python FAQs for 3.7 version
https://docs.python.org/3.7/faq/

A free weekly newsletter featuring curated news, articles, new releases, jobs etc related to Python.
https://www.pythonweekly.com/

PyCon - largest annual gathering for the community using and developing python.
https://us.pycon.org/2019/

Repositories related to the Python Programming language
https://github.com/python

PyPI - The Python Package Index - repository of software for the Python programming language. you can find and install software developed and shared by the Python community.
https://pypi.org/

ABOUT THE AUTHOR

The Author Hemnat Sharma has been working in the technology industry for more than 20 years. As a Linux and Unix expert he has written many articles online and published in the magazines. As a strong believer in helping and sharing his knowledge with community, founded the website https://www.adminschoice.com which he is managing currently.