Python Documentation

Release 1.0

Tuxfux Team

September 18, 2015

CONTENTS

CHAPTER

ONE

UNIT 1: WALK WITH LINUX

1.1 Help in Linux

Linux provides so many commands and because Linux commands provide so many possible options, you can't expect to recall all of them. To help us, Linux provides lots of help options, which let you access a help database that describes each command and its options.

1.1.1 whatis

```
whatis <command name >
```

whatis is a command which will give idea about the abbreviation of the command. Sometimes in a few command this utility dont work so you can run "make whatis" to create the database.

1.1.2 help

```
Syntax: <command> --help
```

This is one command which will provide the various options of the command.

1.1.3 Info

```
Syntax: info <command>
```

This is called the information page. This provides the following info about the command as a whole.

1.1.4 man

```
Syntax: man <command>

This is called manual ( man )
```

This will provide lots of input about the command. Name Synopsis Description Author Reporting bug Copyrights see also

1.1.5 Redhat Documentation

We can also get info about the redhat doc from file:///usr/share/doc. This is very much like a manual and will contain the complete information about the redhat docs.

1.1.6 Documents available on the site.

Please refer internal site for more documentation and Release notes.

1.1.7 Some basic commands

For example

```
ls
ls -l
ls -l -a
ls -la
```

The above example shows few commands like "ls" with and without options.

1.2 Browsing the filesystem

Few principles about the linux filesystem is:

- Everything is file.
- Linux is a single rooted inverted tree structure.
- Names are case sensitive.
- Paths are delimited by /.

1.2.1 Linux Directory hierarchy

```
/ -> root
+ /root => Home directory of root.
+ /bin  => Binary commands. ( Commands for both user and root)
+ /sbin => Binary commands. ( commands which can be run only by root)
+ /proc => Virtual filesystem.
+ /home => home directory of the normal users.
+ /etc/ => configuration files.
+ /tmp  => Temporary directory.
+ /var/ => Log files.
+ /srv  => Server data.
+ /sys  => System Information.
+ /lib  => shared libraries.
+ /usr/lib  => shared libraries.
+ /usr/local/lib  => shared libraries.
```

These are few of the important files under the "/" (root) . Note : / (root) and /root (Home dir of root).

1.2.2 Changing directories

Command for changing the directories is the cd command.

```
Syntax: cd <directory name>

cd        : Home directory of the user.
cd .      : Current working directory.
cd ..     : Parent working directory.
cd        : Moving to the home directory.
cd -      : Swithching between the directories.
```

- . : Represents the current working directory.
-

1.2.3 Absolute and Relative path

In linux every thing is file and starts from root (/).

Suppose consider a directory with path as

```
/aa/bb/cc/dd/ee/ff

/ => root
+ aa
+ bb
+ cc
+ dd
+ ee
```

The path from / to ee is called the absolute path or the complete path.The complete path always starts with a "/".If I am in location "cc" I have to get to directory called "ee".

```
cd dd/ee
```

This will take us to the location "ee". This way of moving around directories is called as the relative path.

1.3 Exercise

1 .Write a program that accepts a comma separated sequence of words as input and prints the words in a comma-separated sequence after sorting them alphabetically. Suppose the following input is supplied to the program: without,hello,bag,world Then, the output should be: bag,hello,without,world

Hints: In case of input data being supplied to the question, it should be assumed to be a console input.

2 .Write a program that accepts sequence of lines as input and prints the lines after making all characters in the sentence capitalized. Suppose the following input is supplied to the program: Hello world Practice makes perfect Then, the output should be: HELLO WORLD PRACTICE MAKES PERFECT

3 .Write a program that accepts a sequence of whitespace separated words as input and prints the words after removing all duplicate words and sorting them alphanumerically. Suppose the following input is supplied to the program: hello world and practice makes perfect and hello world again Then, the output should be: again and hello makes perfect practice world

Hints: In case of input data being supplied to the question, it should be assumed to be a console input. We use set container to remove duplicated data automatically and then use sorted() to sort the data.

4 .Write a program which accepts a sequence of comma separated 4 digit binary numbers as its input and then check whether they are divisible by 5 or not. The numbers that are divisible by 5 are to be printed in a comma separated sequence. Example: 0100,0011,1010,1001 Then the output should be: 1010 Notes: Assume the data is input by console.

Hints: In case of input data being supplied to the question, it should be assumed to be a console input.

CHAPTER

TWO

UNIT 2: EDITOR IN LINUX

2.1 Basic VI concepts

There are various editors in linux .

```
* Emacs
* vi
* pico
* vim
* ed
* ex
```

Now lets get back to the vi editor. In most of the operating system , vi and vim are synonomous. vi is basically an alias of vim.

```
alias vi='vim'
```

2.1.1 Modes of vim/vi

There are basically three modes in vi editor.

```
Modes
+ Command modes
+ Insert modes
+ Execution modes
```

2.1.2 Command Mode

We can do lots of thing in the command mode. Lets list a few now.

- Movement of cursor.
- Cut/Delete
- Copy
- Undo/Redo
- Paste
- Change modes
- Search/Replace

Movement of cursor

```
j : Move the cursor to the top
k : Move the cursor to the left
h : Move the cursor to the bottom
l : Move the cursor to the right
w : Move by a word
b : Word back
( : Move sentence back
) : Move sentence forward
{ : Paragraph above
} : Paragraph below
```

Cut/Delete

```
dd : Cutting or deleting a line.
dl : Cutting or deleting a letter.
dw : Cutting or deleting a word.
d) : Cutting or deleting a sentence ahead.
d( : Cutting or deleting a sentence behind.
d{ : Cutting or deleting a paragraph above.
d} : Cutting or deleting a paragraph below.
```

Copy or Yank

```
yy : Copy/yank a line.
yl : Copy/yank a letter.
yw : Copy/yank a word.
y) : Copy/yank a sentence ahead.
y( : Copy/yank a sentence behind.
y{ : Copy/yank a paragraph above.
y} : Copy/yank a paragraph below.
```

Change

```
cc : Change a line.
cl : Change a letter.
cw : Change a word.
c) : Change a sentence ahead.
c( : Change a sentence behind.
c{ : Change a paragraph above.
c} : Change a paragraph below.
```

Paste

The behaviour of paste is different for a line/sentence and word/letter.

```
P   : (shift P) -> copy above the line.
p   :copy below the line.
P   :copy a word before cursor.
P   :copy a word after cursor.
```

Undo/Redo

```
u       : undo recent changes.
U       : undo all changes.
ctrl-r  : redo last "undone" change.
```

Search

If we want to search a word from top to bottom follow the below.

```
/<word> : For searching a word from top to bottom.
n      : For searching a word from top to bottom.
```

If we want to search a word from bottom to top follow the below.

```
? : For searching a word from bottom to top.
n : For searching a word from bottom to top.
```

2.1.3 Insert Mode

There are diffent ways by which we can enter into the insert mode.

```
a : append after the cursor.
i : insert before the cursor.
o : open a line below.
A : Append to the end of the line.
I : Insert at the beginning of the line.
O : Open a line above.
```

2.1.4 Execution mode

To enter execution mode you have to enter ":" . But before that you need to be in command mode.

```
:w                 : To save a file.
:w <file name >    : To save as a file name.
:x                 : To save a file.
:X                 : To encrypt a file.
:1                 : To move to the starting of the file.
:$                 : To move to the end of the file.
:n                 : To move to the nth poistion in the file.
:1,5w <filename>   : To save from line number 1,5 in a filename.
:set nu            : To set number lines.
:set nonu          : To remove number lines.
:nohl              : To remove highlighting of text.
:q                 : To exit.
:q!                : Forced exit.
:w!                : Forced save.
```

2.1.5 Miscelleneous Mode

```
ctrl-w,s splits the screen horizontally.
ctrl-w,v splits the screen vertically.
ctrl-w,arrow moves between windows.
```

CHAPTER

THREE

UNIT 3: WHY PYTHON

3.1 Welcome students

The prerequisite for the training is linux.

Topics needed:

- Vim editor.
- Browsing the linux filesystem.

3.2 Why python

Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java. There are also low-level languages, sometimes referred to as "machine languages" or "assembly languages." Loosely speaking, computers can only run programs written in low-level languages. So programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.

The advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct.

Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

3.2.1 Interpreter and Compiler

Two kinds of programs process high-level languages into low-level languages: interpreters and compilers.

An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately readinglines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this context, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.

Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: interactive mode andscript mode. We will understand about the interactive mode and script mode in the future topics.

3.2.2 Features of Python

The Features of python.

Lets discuss them one by one:

- Simple.

  Python is a very simple language. It more like reading a English novel.The Pseudo-code of python is one of its greatest strengths.

- Ease to learn

  Python is extremely easy to learn.

- Open source product.

  Python is a open source product. It means you have the freedom to distribute the copies of the softwares. Make changes to the code/read the source code. we can reuse the code and start using it for our day out needs.

CHAPTER

FOUR

UNIT 4: SETTING UP PYTHON IN LINUX

4.1 Installation of python

we can run the yum install command to install python. since our training is more on redhat linux , i am concentrating on installation on redhat.

```
yum install python
```

we can query by running the rpm command.

```
[root@RHEL6 ~]# rpm -qa|grep -i ^python
python-matplotlib-0.99.1.2-1.el6.i686
python-lxml-2.2.3-1.1.el6.i686
python-netaddr-0.7.5-4.el6.noarch
python-slip-0.2.11-1.el6.noarch
python-rhsm-0.96.15-1.el6.noarch
python-libs-2.6.6-29.el6.i686
python-ethtool-0.6-1.el6.i686
python-setuptools-0.6.10-3.el6.noarch
python-dmidecode-3.10.13-1.el6.i686
python-nose-0.10.4-3.1.el6.noarch
python-paramiko-1.7.5-2.1.el6.noarch
python-iniparse-0.3.1-2.1.el6.noarch
python-dateutil-1.4.1-6.el6.noarch
python-simplejson-2.0.9-3.1.el6.i686
python-gudev-147.1-4.el6_0.1.i686
python-kerberos-1.1-6.2.el6.i686
python-pycurl-7.19.0-8.el6.i686
python-iwlib-0.1-1.2.el6.i686
python-urlgrabber-3.9.1-8.el6.noarch
python-meh-0.11-3.el6.noarch
python-2.6.6-29.el6.i686
python-ldap-2.3.10-1.el6.i686
python-crypto-2.0.1-22.el6.i686
python-nss-0.11-3.el6.i686
[root@RHEL6 ~]#
```

check the version of python:

```
[root@RHEL6 ~]# python --version
Python 2.6.6
[root@RHEL6 ~]#
```

Location of the Python binary:

```
[root@RHEL6 ~]# which python
/usr/bin/python
[root@RHEL6 ~]#
```

So we are set :)

4.2 Indentation

Most important part of python scripting is indentation.

I basically setup following features in .vimrc file

```
santosh-PI945GCM ~ # cat .vimrc
syntax on
set nu
set tabstop=2
set expandtab
```

The **.vimrc** file is basically created in the home directory of the host.

Lets try to understand each and every option that we have setup in our .vimrc files.

- syntax on

  This sets the *syntax on* for code in your vim editor.

- set nu

  This sets the number line the moment you login to your vi editor.

- set tabstop=2

  This option controls the number of space characters that will be inserted when the tab key is pressed. The above option inserts 2 spaces for a tab.

- set expandtab

  This options allows us to insert the space characters whenever a tab is pressed.

CHAPTER

FIVE

UNIT 5: HELP

5.1 Python Help

5.1.1 Using python command line

you can use python to get help:

```
santosh-PI945GCM ~ # python
Python 2.7.4 (default, Apr 19 2013, 18:28:01)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>>
```

Help in python interpreter

we can type the help() command in the python interpreter to get the help prompt.

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

To come out of the help prompt:

```
help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
```

```
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

To get help on particular topics type:

- modules
- keyword
- topics

5.2 Introduction to Ipython

Ipython is just the same as python command prompt. But there are few features of the Ipython, lets go via few of them.

5.3 Using ipython

you can also use ipthon to get the help:

```
santosh-PI945GCM vapython # ipython
Python 2.7.4 (default, Apr 19 2013, 18:28:01)
Type "copyright", "credits" or "license" for more information.
IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

5.4 Features:

5.4.1 Tab completion

Tab completion, especially for attributes, is a convenient way to explore the structure of any object . Simply type object_name followed by a <TAB> to view the object's attributes.Besides Python objects and keywords, tab completion also works on file and directory names.

5.4.2 Exploring your objects

Typing object_name? will print all sorts of details about any object, including docstrings, function definition lines (for call arguments) and constructor details for classes.

5.4.3 Running and Editing

The %run magic command allows you to run any python script and load all of its data directly into the interactive namespace. Since the file is re-read from disk each time, changes you make to it are reflected immediately .

```
In [4]: !cat /tmp/pa1.py
x = 3
y = 5

if x < y:
  print x , "is less than" , y
elif x > y:
  print x ,"is greater than", y
else:
  print x , " and ", y , "are equal"
```

I have used the system command *!* to display the file. Now let me use the run operator to execute the commands.

```
In [3]: run /tmp/pa1.py
3 is less than 5
```

%run has special flags for timing the execution of your scripts (-t), or for running them under the control of either Python's pdb debugger (-d) or profiler (-p).

To run the script in the debugging mode.

```
In [5]: run -d /tmp/pa1.py
Breakpoint 1 at /tmp/pa1.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> /tmp/pa1.py(1)<module>()
1---> 1 x = 3
    2 y = 5
    3

ipdb> c
3 is less than 5
```

To check the timer on the python script.

```
In [6]: run -t /tmp/pa1.py
3 is less than 5

IPython CPU timings (estimated):
User   :       0.00 s.
System :       0.00 s.
Wall time:       0.00 s.
```

The %edit command gives a reasonable approximation of multiline editing, by invoking your favorite editor on the spot. IPython will execute the code you type in there as if it were typed interactively.

Type Edit to open an editor of your choice.

```
In [9]: edit
```

To open multiple lines for an editor.

```
In [10]: edit 7:8
```

5.4.4 History

IPython stores both the commands you enter, and the results it produces. You can easily go through previous commands with the up- and down-arrow keys, or access your history in more sophisticated ways.

Input and output history are kept in variables called `In` and `Out`, keyed by the prompt numbers, e.g. `In[4]`. The last three objects in output history are also kept in variables named _, ___ and ____.

You can use the `%history` magic function to examine past input and output. Input history from previous sessions is saved in a database, and IPython can be configured to save output history.

5.4.5 System shell command

We can run the system shell commands by simple prefix it with ! .

Lets have a look at the below example to understand how the nslookup comamnd is run.

```
In [27]: !nslookup www.google.com
Server:   127.0.1.1
Address:  127.0.1.1#53

Non-authoritative answer:
Name: www.google.com
Address: 74.125.236.178
Name: www.google.com
Address: 74.125.236.176
Name: www.google.com
Address: 74.125.236.179
Name: www.google.com
Address: 74.125.236.177
Name: www.google.com
Address: 74.125.236.180
```

I use them frequently , we will be using few more options in the comming sections.

5.5 pypy

Python has a huge community . There is a contribution from every corner of the world about the various modules. you can get a glance of every module in the site below.

http://pypy.org/

CHAPTER

SIX

UNIT 6: VARIABLES EXPRESSIONS AND STATEMENTS

6.1 Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A variable is a name that refers to a value.

Lets work on assiging values to the variable. Lets take few quick examples to understand the variables syntaxes and needs.

```
In [8]: school = "De paul"

In [9]: another_school = "saint vincent"

In [10]: city = "small township"

In [11]: beach = "small beach"

In [12]: time_of_bus = 12.00
```

we can use all the below named variable in sentence formation. Lets try to set some quick code to see the examples.

```
In [14]: print "my school name is:" , school
my school name is: De paul

In [15]: print "another school near to my", school , "is" , another_school
another school near to my De paul is saint vincent

In [16]: print "our school", school , "and" , another_school,"are in ", city
our school De paul and saint vincent are in  small township

In [17]: print "our city" , "has a", beach
our city has a small beach

In [18]: print "we daily come back home by ",time_of_bus
we daily come back home by  12.0
```

we will be covering more example on print statements soon.

```
In [23]: my_age = 25

In [24]: my_height = 172

In [25]: my_weight = 160

In [26]: my_language = 'telugu'
```

```
In [27]: my_dress_color = 'pink'

In [28]: my_hair='curly'

In [29]: my_name ='anusha'
```

Lets start using the above variable as print statements.

```
In [30]: print "my name is %s" %(my_name)
my name is anusha

In [31]: print "my name is %s and age %d " %(my_name,my_age)
my name is anusha and age 25

 In [33]: print "The colour of my dress is %s , and my hair is %s "  %(my_dress_color,my_hair)
The colour of my dress is pink , and my hair is curly
```

6.1.1 Variable names and keywords

Programmers generally choose names for their variables that are meaningful they document what the variable is used for.

Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. It is legal to use uppercase letters, but it is a good idea to begin variable names with a lowercase letter.

Lets see some examples which bomb out:

```
In [10]: 123variable = "hello"
File "<ipython-input-10-59012fb1363d>", line 1
123variable = "hello"
          ^
SyntaxError: invalid syntax
```

Lets see one more example on how to not give the variable name.

```
In [15]: case@ = "hello"
 File "<ipython-input-15-8944707581b0>", line 1
case@ = "hello"
    ^
SyntaxError: invalid syntax
```

the above function failed as @ is a illegal character.

If you verify in the above case, the variable name started with the number. That is the reason it failed.

```
In [11]: _variable = "hello"

In [13]: print _variable
hello
```

we can see that we can use the _ value as part of the variable.

6.2 Understanding variables

In python we can identify if a value is integer,string or a float.

lets try out few examples in the python interpreter:

The below example shows the examples of string.

```
In [1]: type("hello collegues")
Out[1]: str

In [2]: type('hello collegues')
Out[2]: str
```

The below examples show the examples of intergers and float.

```
In [3]: type(20)
Out[3]: int

In [4]: type(20.0)
Out[4]: float
```

But a catch here.If we put single quotes are double quotes with the number it become strings. Lets see the examples here.

```
In [5]: type('20')
Out[5]: str

In [6]: type('20.0')
Out[6]: str
```

6.3 Playing with numbers

Every programming language has some mathematics associated with it lets work a bit with it.

Lets understand some math symbols and try to use in in few of our examples.

- + plus
- - minus
- / slash
- asterisk
- % percent
- < less-than
- > greater-than
- <= less than equal to
- >= greater than equal to

Lets start typing the code on our *ipython* editor and see what the outputs are.

```
In [1]: print "lets do some calculation"
lets do some calculation

In [2]: print "division", 25 + 25 / 2
division 37

In [3]: print "division", ( 25 + 25 ) / 2
division 25

In [4]: print "division : 25 + 25 /2 "
```

```
division : 25 + 25 /2

In [5]: print 3 > 2
True

In [1]: print "is 3 > 2"
is 3 > 2

In [2]: print "is 3 > 2" , 3 > 2
is 3 > 2 True

In [5]: print "what is 3 + 2 = ", 3 + 2
what is 3 + 2 =  5

In [32]: print "fraction", 8.0 / 5
fraction 1.6

In [33]: print "fraction", 8.0 // 5
fraction 1.0
```

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%)

Some examples on the modulus operator:

```
In [21]: num1 = 10

In [22]: num2 = 2

In [23]: num1 % num2
Out[23]: 0
```

We can assign the reminder value to the variable also

```
In [24]: value = num1 % num2

In  [25]: print value
0

In [26]: type(value)
Out[26]: int
```

6.4 Playing with strings

Lets start playing a bit with strings.

In the below example lets try to conver few examples.

```
In [36]: 'good eggs'
Out[36]: 'good eggs'

In [37]: 'let's not'
File "<ipython-input-37-3fdc45959b16>", line 1
  'let's not'
       ^
SyntaxError: invalid syntax
```

If you notice below to avoid the issue of syntax error i have used a “” backslash.

```
In [38]: 'let\'s not'
Out[38]: "let's not"
```

Also we can save the day by using the double quotes.

```
In [39]: "let's not"
Out[39]: "let's not"
```

or we can aso go for the below method

```
In [41]: "'let's' not"
Out[41]: "'let's' not"
```

Lets carry on with printing some strings . See in the below print command i have not assinged any variables.

```
In [46]: print "i take %s classes " % 'perl'
i take perl classes
```

i want to print the name **tuxfux** ten times

```
In [51]: print " tuxfux " * 10
 tuxfux  tuxfux  tuxfux  tuxfux  tuxfux  tuxfux  tuxfux  tuxfux  tuxfux  tuxfux
```

Lets try couple of more examples:

```
In [58]: a = "t"

In [59]: b = "o"

In [60]: c = "d"

In [61]: d = "a"

In [62]: e = "y"

In [63]: f = "i"

In [64]: g = "s"

In [65]: h = "s"

In [66]: i = "u"

In [67]: j = "n"

In [68]: k = "d"

In [69]: l = "a"

In [70]: m = "y"

In [75]: print a + b + c + d + e
today

In [76]: print "%s + %s + %s + %s + %s" %(a,b,c,d,e)
t + o + d + a + y

In [77]: print "%s  %s  %s  %s  %s" %(a,b,c,d,e)
t  o  d  a  y
```

```
In [79]: print "%s%s%s%s%s" %(a,b,c,d,e)
today
```

Lets work now on few more examples.

```
In [82]: formate = "%r %r %r %r"

In [83]: print formate % (1,2,3,4)
1 2 3 4

In [84]: print formate % ("one","two","three","four")
'one' 'two' 'three' 'four'

In [86]: print formate % ( True,False,False,True)
True False False True

In [95]: print formate % (
"today is wednesday"
, "tomorrow is thursday",
"yesterday was tuesday",
"day after tomorrow is friday")
'today is wednesday' 'tomorrow is thursday' 'yesterday was tuesday' 'day after tomorrow is friday'

In [92]: print formate % ( formate, formate , formate, formate )
'%r %r %r %r' '%r %r %r %r' '%r %r %r %r' '%r %r %r %r'
```

6.5 More fun with strings

Lets work on few more example of printing.

```
In [97]: days = "mon tue wed thu fri sat sun"

In [98]: months = "jan\nfeb\nmar\napr\nmay\njun\njul\naug\nsep\noct\nnov\ndec"

In [99]: print "here are the days %s" %(days)
here are the days mon tue wed thu fri sat sun

In [100]: print "here are the days %r" %(days)
here are the days 'mon tue wed thu fri sat sun'

In [101]: print "here are the months %s" %(months)
here are the months jan
feb
mar
apr
may
jun
jul
aug
sep
oct
nov
dec

In [102]: print "here are the months %r" %(months)
here are the months 'jan\nfeb\nmar\napr\nmay\njun\njul\naug\nsep\noct\nnov\ndec'
```

If i want to print a paragraph there is a way of doing it.

```
In [103]: print """
 .....: Today i learned to type in python
 .....: I am really happy that its working for me
 .....: lets hope i continue to do working similarly in python
 .....: """

Today i learned to type in python
I am really happy that its working for me
lets hope i continue to do working similarly in python
```

Lets try to extract some charaters out of a string. Lets go with some examples below.

```
In [44]: word = 'python'
```

Now lets try to extract some characters out of the following word 'python'.

```
In [45]: word[0]
Out[45]: 'p'

In [50]: word[-1]
Out[50]: 'n'

In [51]: word[-2]
Out[51]: 'o'

In [49]: word[0:2]
Out[49]: 'py'
```

Lets play with few more examples.

```
In [52]: word[:2]
Out[52]: 'py'

In [53]: word[2:]
Out[53]: 'thon'

In [54]: word[:2] + word[2:]
Out[54]: 'python'
```

But one thing we need to keep in mind is that strings are non-mutable.

Few examples to support the above case

```
In [58]: word
Out[58]: 'python'

In [59]: word[2]
Out[59]: 't'

 In [60]: word[2] = 'y'
 ---------------------------------------------------------------------------
 TypeError                                 Traceback (most recent call last)
 <ipython-input-60-b75148953cd2> in <module>()
 ----> 1 word[2] = 'y'

 TypeError: 'str' object does not support item assignment
```

If you see in the above example we can never assign values to the string.

6.6 comments

Commenting is very important as far as any programming language is concerned. In python also we can go ahead and comment.

```
1 #!/usr/bin/env python                    # she-bang
2 # This is our first program.
3 # I love python as its more english like.
4
5 print "Welcome to tuxfux \n" # this is basic print statement.
```

If you notice the # symbol it used to comment in python. we can comment parts of the code or provide some hints or answers about code we have written.

we can execute the code in two ways.

- using the python binary

```
python comment_1.py
Welcome to tuxfux
```

- Running it as executable

```
# chmod +x comment_1.py
# ls -l comment_1.py
-rwxr-xr-x 1 root root 156 Nov 25 23:37 comment_1.py
# ./comment_1.py
Welcome to tuxfux
```

Going forward in all the tutorials make it a point to run the script in anyof the following ways.

6.7 Accepting Input

Now lets work on accepting the inputs from the user. Lets go via few of the examples and we will understand how its going to work.

```
In [109]: print "enter your name"
enter your name

In [110]: name=raw_input()
tuxfux

In [111]: type(name)
Out[111]: str
```

If you notice in the above example, we have put a question asking for name. Later we used **raw_input()** to fetch for the name of the user. if you notice, *type(name)* is actully displaying it as string.

```
In [112]: name = raw_input("Please enter your name:")
Please enter your name:tuxfux

In [113]: print name
tuxfux

In [114]: type(name)
Out[114]: str
```

If you notice in this above example, i have used the print code within **raw_input**.

Now lets try to understand the difference between **raw_input()** and **input()** .

Case I : using **raw_input()**

```
In [115]: age = raw_input("please enter your age:")
please enter your age:16

In [116]: print age
16

In [117]: type(age)
Out[117]: str
```

Case II: using the **input()**

```
In [118]: age = input("please enter your age:")
please enter your age:16

In [119]: print age
16

In [120]: type(age)
Out[120]: int
```

if you note very closely,

- **raw_input()** takes values as string.
- **input()** takes the values as input.

CHAPTER

SEVEN

UNIT 7: CONTROL STATEMENTS

7.1 Introduction

Till now, there has always been a series of statements and Python faithfully executes them in the same order. What if you wanted to change the flow of how it works?For example, you want the program to take some decisions and do different things depending on different situations such as printing 'Good Morning' or 'Good Evening' depending on the time of the day.

There are three control flows:

- if conditions
- for
- while

we will be covering the looping in the other concepts.

7.1.1 Boolean expressions

A boolean expression is an expression that is either true or false. The following examples use the operator == , which compares two operands and produces True if they are equal and False otherwise.

```
In [29]: 5 == 5
Out[29]: True

In [30]: 5 == 6
Out[30]: False
```

If we notice the type on the *True* and *False* values we will notice that the following values are both boolean types.

```
In [32]: type(True)
Out[32]: bool

In [33]: type(False)
Out[33]: bool
```

The == operator is just one of the relational operators we have other operators too.

Operator	Explanation
!=	not equal to
>	greater than
<	less than
>=	greater than
<=	lesser than

Lets see one more example on above operators.

```
In [35]: 5 != 5
Out[35]: False

In [36]: 5 >= 5
Out[36]: True

In [37]: 5 <= 5
Out[37]: True
```

Note: = is for assignment and == is for comparison.

7.1.2 Logical Operators

There are three logical operators: “and” , “or” and “not”.

Lets take few examples to understand these operators.

```
In [38]: 3 == 3 or 3 != 3
Out[38]: True
```

The above value is True since one of the values need to be true for the above asssumtion to be true.

```
In [39]: 3 == 3 and 3 != 3
Out[39]: False
```

The above value is false since either of the values need to be true for the above asssumtion to be true.

not operator negates a boolean expression.

```
In [40]: not True
Out[40]: False

In [41]: not False
Out[41]: True
```

If you see above the **not** of True is False. **not** of False is True.

7.2 If execution

The if statement is used to check a condition and if the condition is true, we run a block of statements (called the if-block), else we process another block of statements (called the else-block). The else clause is optional.

Syntax:

```
if <condition> :
  (if-block)
else:
  (else block)
```

Here else is optional we can also write the statements as below.

Syntax:

```
if <condition> :
  (if block)
```

Lets consider some examples below to undestand completely how it works.

```
In [45]: if 2 < 5:
....:     print " 2 is not greater than 5"
....:
2 is not greater than 5
```

If you note above the condition statement is True, so it will execute the if block.

Lets see one more example on what will happen if it see the values if false.

```
In [43]: if 2 > 5:
....:     print "2 is not greater than 5"
....:
```

7.2.1 pass

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements . In that case, you can use the pass statement, which does nothing.

Example on the *pass*

```
In [46]: if 2 < 5:
....:     pass
....:
```

if you don't want to execute any statement in if block you can use the *pass* statement.

7.3 If else

Lets first understand the syntax of if .. else

Syntax:

```
if <condition> :
  (if block)      # run if block if condition True
else
  (else block)    # run else block if condition False
```

Example:

```
In [48]: if 2 < 5:
   ....:     print "2 is less than 5"
  ....: else:
  ....:     print "2 is not less than 5"
  ....:
        2 is less than 5
```

Example:

```
In [51]: if 2 > 5:
  print "2 is greater than 5"
else:
  print "2 is not greater than 5"
 ....:
2 is not greater than 5
```

7.4 if .. elif .. else

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a chained conditional

Syntax:

```
if <condition>:
   <ifblock>
elif <condition>:
   <elif block>
else
   <else block>
```

Example:

```
1 x = 3
2 y = 5
3
4 if x < y:
5   print 'x is less than y'
6 elif x > y:
7   print 'x is greater than y'
8 else:
9   print 'x and y are equal'
```

Lets see what we get on execution.

```
santosh-PI945GCM vapython # python /tmp/pa1.py
x is less than y
```

If you notice we have got the **x is less than y** as output. Its True. But the variables have not expanded.

Lets see how to expand the variables.

```
1 x = 3
2 y = 5
3
4 if x < y:
5   print x , "is less than" , y
6 elif x > y:
7   print x ,"is greater than", y
8 else:
9   print x , " and ", y , "are equal"
```

Lets see what we get on execution

```
santosh-PI945GCM vapython # python /tmp/pa1.py
3 is less than 5
```

7.5 Exercises

- WAP to find the min of the two numbers ?
- Wap to find the max of the three numbers ?
- Wap to find out if a given character is a vowel or consonant ? Note: 'a','e','i','o','u' are the vowels.
-

CHAPTER

EIGHT

UNIT 8: RECURSIONS

8.1 Introduction

Sometimes we need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

Python programming language provides following types of loops to handle looping.

- while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

- for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

- nested loops

You can use one or more loop inside any another while, for or do..while loop.

8.2 For

Python **for** iteration iterates over the items of any sequence (a list or string), in the order that they appear in sequence.

Lets take a list as a quick example. We will be convering a lot on the list in the next topics.

```
week = ('sunday','monday','tuesday')
```

Lets use the week array and try to iterate over them.

Example:

```
1 #!/usr/bin/python
2
3 week = ('sunday','monday','tuesday')
4
5 for a in week:
6   print a
```

Output:

```
santosh-PI945GCM loop # python for.py
sunday
monday
tuesday

~
```

8.3 While

The while statement allows us to repeatedly execute a block of statement as long as a condition is true. A while statement is a example of what is called as looping statement.A while statement can have optional else statement.

Example:

```
1 #!/usr/bin/python
2
3 number = 7
4 test = True
5
6 while test:
7   guess = int(raw_input('Enter an integer :'))
8
9   if guess == number:
10     print 'you gussed the right number !! '
11     test= False
12   elif guess < number:
13     print 'No, the number is a little higher than that'
14   else:
15     print 'No, the number is a little lower than that'
16 else:
17    print 'The while loop is over.'
18
19
20 print 'Done'
21
```

Output:

```
santosh-PI945GCM loop # python while.py
Enter an integer :2
No, the number is a little higher than that
Enter an integer :2
No, the number is a little higher than that
Enter an integer :23
No, the number is a little lower than that
Enter an integer :7
you gussed the right number !!
The while loop is over.
Done
```

8.4 range

If you do need to iterate over a sequence of numbers, the built-in function **range** comes in handy.

Lets take a quick example:

Example

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

Suppose we want to go from a number *5* to *10* we can go as below.

```
In [2]: range(5,10)
Out[2]: [5, 6, 7, 8, 9]
```

Similary, if we want to print even numbers from *0* to *10* we can do the same.

```
In [3]: range(0,10,2)
Out[3]: [0, 2, 4, 6, 8]
```

8.5 Break

The break statement is used to break out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has been completely iterated over.

An important note is that if you break out of a for or while loop, any corresponding loop else block is not executed.

Lets take a quick example to understand the above example

Example:

```
1 #!/usr/bin/python
2
3 while True:
4   s = raw_input('Enter something:')
5   if s == 'quit':
6     break
7   print 'Lenght of the string is:', len(s)
8
9 print 'Done'
```

Output:

```
santosh-PI945GCM loop # python break.py
Enter something:hai
Lenght of the string is: 3
Enter something:quit
Done
```

8.6 continue

The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

Example:

```
1 #!/usr/bin/python
2
3 for i in range(10):
4   if i == 2:
5     continue
```

```
6   print i
7
```

Output:

```
santosh-PI945GCM loop # python for_continue.py
0
1
3
4
5
6
7
8
9
```

CHAPTER

NINE

UNIT 9: DATA STRUCTURES

9.1 Lists

9.1.1 What is lists

Like a string, lists are sequence of values. The values in a list are called elements or sometimes Items.

There are severeal ways to create a list, lets go via few of the example.

```
[10,20,30,40,50]
['10','20','30','40',50']
[10,20,30,'syam','marcus']

In [1]: list = [10,10,30,40,50]
In [2]: list
Out[2]: [10, 10, 30, 40, 50]

In [4]: list1 = [ '10','20','30','40','50','60' ]
In [5]: list1
Out[5]: ['10', '20', '30', '40', '50', '60']

In [6]: list1 = [ 10,20,30,'syam','marcus' ]
In [7]: list1
Out[7]: [10, 20, 30, 'syam', 'marcus']
```

9.1.2 Mutable Lists

Accessing the list variables

The variables of the list can be accessed in the given way.

```
In [7]: list1
Out[7]: [10, 20, 30, 'syam', 'marcus']

In [8]: list1[0]
Out[8]: 10

In [9]: list1[3]
Out[9]: 'syam'
```

The array values can be assesed using the index values for the items in the array.

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

Lets take a quick example to understand it.

Example:

```
In [10]: newlist = [1 ,2 ,3]
In [11]: newlist[0] = 4
In [12]: newlist
Out[12]: [4, 2, 3]
```

If you notice in the above example, you can see that we have assigned a value **4** to the newlist variable.

Just for a recap lets check on string variables.

```
In [13]: string = "hello"

In [14]: string[0]
Out[14]: 'h'


In [16]: string[0] = 'w'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-d69ac2e43889> in <module>()
----> 1 string[0] = 'w'

TypeError: 'str' object does not support item assignment
```

In operator

We can use the **in** operator on the list

```
In [17]: items = [ 'fair' ,'and' , 'lovely' ]

In [18]: 'lovely' in items
Out[18]: True

In [19]: 'an' in items
Out[19]: False
```

9.1.3 Traversing a list

The most common way of traversing a list is using the for loop.

Lets take a quick example:

```
In [17]: items = [ 'fair' ,'and' , 'lovely' ]

In [22]: for item in items:
 ....:     print item
 ....:
fair
and
lovely
```

9.1.4 List operations

1.

The **+** operator concatenates lists.

```
In [27]: a
Out[27]: ['1', '2', '3', '4', '5', '6']

In [28]: b
Out[28]: ['a', 'b', 'c', 'd']

In [29]: a + b
Out[29]: ['1', '2', '3', '4', '5', '6', 'a', 'b', 'c', 'd']
```

If you notice in the below example, both the arrays are concatinated.

1.

The ***** operator repeats a list a given number of times.

```
In [30]: a = [1,2,3]

In [31]: a * 3
Out[31]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

9.1.5 Slices

We can use slice on array variable too. Lets take some quick example on the following.

```
In [33]: a = ['a','b','c','d','e']

In [34]: a[1:3]
Out[34]: ['b', 'c']

In [35]: a[:3]
Out[35]: ['a', 'b', 'c']

In [36]: a[3:]
Out[36]: ['d', 'e']

In [37]: a[:]
Out[37]: ['a', 'b', 'c', 'd', 'e']
```

Slices can also used for assignement.

```
In [39]: a
Out[39]: ['a', 'b', 'c', 'd', 'e']

In [40]: a[3:] = ['f','g']

In [41]: a
Out[41]: ['a', 'b', 'c', 'f', 'g']
```

If you notice we have assigned values **f,g** in position of **d,e** .

9.1.6 Deleting list values

There are various ways of deleting elements from a list.

pop Function

If we know the index of the element we can quickly go around and pop that element out.

```
In [42]: a
Out[42]: ['a', 'b', 'c', 'f', 'g']

In [44]: a.pop(2)
Out[44]: 'c'

In [45]: a
Out[45]: ['a', 'b', 'f', 'g']
```

if you see **pop** is one of the functions to remove the element from the list.

del Function

You can use the **del** operator to delete the element in an list.

```
In [46]: a = ['a','b','c','d','e']

In [47]: del a[3]

In [48]: a
Out[48]: ['a', 'b', 'c', 'e']
```

we can also remove slice of elements using the **del** operator.

```
In [53]: a = ['a','b','c','d','e']

In [55]: del a[3:4]

In [56]: a
Out[56]: ['a', 'b', 'c', 'e']
```

remove function

you can use the **remove** function to delete the elements in a list.

```
In [49]: a = ['a','b','c','d','e']
In [51]: a.remove('d')

In [52]: a
Out[52]: ['a', 'b', 'c', 'e']
```

9.1.7 Lists and Strings

In this topic we will learn about converting the

1. Lists to String.

2. Strings to list.

Lists to String

Lets take some quick example

```
In [2]: a = 'hello'

In [3]: b = list(a)

In [4]: b
Out[4]: ['h', 'e', 'l', 'l', 'o']
```

If you notice , we have split the string **hello** into list variable.

split

We can use the split variable to split a sentence into array variable. If we use a word it will be taken as one array variable.

```
In [9]: b = a.split()

In [10]: b
Out[10]: ['hello']

In [13]: a = "hello today is good"

In [14]: b = a.split()

In [15]: b
Out[15]: ['hello', 'today', 'is', 'good']
```

If you notice , we have converted the string to variable.

There is one more quick way of conveting the strings into the arrays.

```
In [16]: a = "hello:today:is:good"

In [20]: b = a.split(':')

In [21]: b
Out[21]: ['hello', 'today', 'is', 'good']
```

if you notice here the delimiter is **:**

Strings to list.

we can convert the strings to lists. Lets take a quick example to understand this.

```
In [27]: b
Out[27]: ['hello', 'today', 'is', 'good']

In [28]: limiter=' '

In [29]: c = limiter.join(b)
```

```
In [30]: c
Out[30]: 'hello today is good'
```

If you notice , i have converted the list variable **b** to string value **c** .

9.1.8 Aliases

Consider we have a both variable refering to same object. That is called aliasing.

```
In [42]: a = ['hello','today']

In [43]: b = a

In [44]: b is a
Out[44]: True

In [45]: a is b
Out[45]: True
```

This concept is called aliasing. If you note the **is** operator used to verify if both the variable are refering to same object.

Now lets consider one more example where we have different object.

```
In [46]: c = ['hello' ,'today']
In [47]: a = ['hello','today']

In [48]: c is a
Out[48]: False

In [49]: a is c
Out[49]: False
```

If you see in the above example its very clearly understood that both the variables are different objects.

9.1.9 Functions

These are some of the array function, which will come in handy.

sort

we can use the sort function to sort the elements of the array.

```
In [68]: b = ['jan','feb','mar','apr']

In [69]: b.sort()

In [70]: b
Out[70]: ['apr', 'feb', 'jan', 'mar']
```

reverse

we can use the reverse function to reverse the elements of the array.

```
In [70]: b
Out[70]: ['apr', 'feb', 'jan', 'mar']

In [72]: b.reverse()

In [73]: b
Out[73]: ['mar', 'jan', 'feb', 'apr']
```

Insert

we can use the insert function to insert new elements into the array.

```
In [77]: b
Out[77]: ['mar', 'jan', 'feb', 'apr']

In [78]: b.insert(3,'may')

In [79]: b
Out[79]: ['mar', 'jan', 'feb', 'may', 'apr']
```

remove

Removes the first occurance of the value.

```
In [79]: b
Out[79]: ['mar', 'jan', 'feb', 'may', 'apr']

In [80]: b.remove('jan')

In [82]: b
Out[82]: ['mar', 'feb', 'may', 'apr']
```

append

we can use the append function to append the value.

```
In [84]: b
Out[84]: ['mar', 'feb', 'may', 'apr']

In [85]: b.append('jan')

In [86]: b
Out[86]: ['mar', 'feb', 'may', 'apr', 'jan']
```

pop

Pop removes the last element.

```
In [91]: b
Out[91]: ['mar', 'feb', 'may', 'apr', 'jan']

In [92]: b.pop()
Out[92]: 'jan'
```

```
In [93]: b
Out[93]: ['mar', 'feb', 'may', 'apr']
```

count

count variable is used to find the occurance of the value in array.

```
In [97]: a = [1,2,3,1,4,5,6,7]

In [98]: a.count(1)
Out[98]: 2

In [99]: a.count(2)
Out[99]: 1
```

9.2 Tuples

9.2.1 what is tuples

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable.

Examples:

```
In [54]: a = ('a','b','c','d','e')

In [55]: type(a)
Out[55]: tuple

In [56]: a = 'a','b','c','d','e'

In [57]: type(a)
Out[57]: tuple
```

Note if we take just one value, it acts as string.

```
In [58]: a = 'a'

In [59]: type(a)
Out[59]: str

In [60]: a = ('a')

In [61]: type(a)
Out[61]: str
```

we can also convert a string to a tuple.

```
In [63]: a = tuple('strings')

In [64]: a
Out[64]: ('s', 't', 'r', 'i', 'n', 'g', 's')
```

Accessing a tuple

Most of the list operators also work on tuples.

```
In [64]: a
Out[64]: ('s', 't', 'r', 'i', 'n', 'g', 's')

In [65]: a[0]
Out[65]: 's'

In [66]: a[1]
Out[66]: 't'
```

We can also use the slice operator to select a range of elements.

```
In [67]: a[0:3]
Out[67]: ('s', 't', 'r')

In [68]: a[:3]
Out[68]: ('s', 't', 'r')

In [69]: a[3:]
Out[69]: ('i', 'n', 'g', 's')

In [70]: a[3:4]
Out[70]: ('i',)

In [71]: a[3]
Out[71]: 'i'
```

9.2.2 Tuple is immutable

If you notice in the below example, we cannot assign values to the tuple variables.

```
In [72]: a[3] = ('j')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-72-84c5654ef43e> in <module>()
----> 1 a[3] = ('j')

TypeError: 'tuple' object does not support item assignment
```

9.2.3 Tuple Assignment

Sometimes we need to swap couple of variables. Lets see how to do it.

```
In [74]: a = 10

In [75]: b = 20

In [76]: c = a

In [77]: a = b

In [78]: b = c
```

```
In [79]: a
Out[79]: 20

In [80]: b
Out[80]: 10
```

we can simple do this over by a easy method as tuple assignment .

```
In [81]: a = 10

In [82]: b = 20

In [83]: (a,b) = (b,a)

In [84]: a
Out[84]: 20

In [85]: b
Out[85]: 10
```

unpack

we can also unpack the value using the tuples.

```
In [87]: a = (1,2)

In [88]: (c,d) = (1,2)

In [89]: c
Out[89]: 1

In [90]: d
Out[90]: 2
```

Lets take a quick example to split an email address into username and domain.

```
In [92]: email = 'tuxfux.hlp@gmail.com'

In [93]: (mail,domain) = email.split('@')

In [94]: mail
Out[94]: 'tuxfux.hlp'

In [95]: domain
Out[95]: 'gmail.com'
```

If you notice here , we have used a quick example of split option and unpacked the email address into **domain** and **address**.

9.2.4 Comparing tuples

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

Example:

```
In [29]: (0,1,2) > (0,0,3)
Out[29]: True

In [30]: (0,1,2) > (0,1,3)
Out[30]: False

In [31]: (1,1,2) > (0,1,3)
Out[31]: True

In [32]: (0,1,2) > (1,1,3)
Out[32]: False
```

9.2.5 Lists and Tuples

zip is a built-in function that takes two or more sequences and "zips" them into a list of tuples where each tuple contains one element from each sequence.

Lets consider a zip example in string and a list.

Example:

```
In [34]: a = 'abc'

In [35]: t = [1,2,3]

In [36]: zip(a,t)
Out[36]: [('a', 1), ('b', 2), ('c', 3)]
```

If we take the same number of elements on the both side, it works fine. Lets take an example where the values are not similar on both sides.

```
In [37]: a = 'abcd'

In [38]: t = [1,2,3]

In [39]: zip(a,t)
Out[39]: [('a', 1), ('b', 2), ('c', 3)]
```

we can use tuple assignment in a for loop to traverse a list of tuples:

```
In [40]: for l,n in zip(a,t):
 ....:     print l,n
 ....:
a 1
b 2
c 3
```

9.2.6 Functions

Lets look at few of the functions of the tuples.

index

index is used to return first index of the value.

```
In [107]: b = ('jan','feb','mar','apr')

In [108]: b.index('mar')
Out[108]: 2
```

count

count is used to find the occurance of the value in a tuple.

```
In [114]: b
Out[114]: ('jan', 'feb', 'mar', 'apr', 'jan', 'may')

In [112]: b.count('jan')
Out[112]: 2

In [113]: b.count('feb')
Out[113]: 1
```

9.3 Dictionaries

9.3.1 Dictionary

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

we can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

we can use the { } for dictionary .

example:

```
new = {}
```

Lets take a quick example on understanding the dictionary variables.

Examples:

```
In [2]: a = dict()

In [3]: type(a)
Out[3]: dict

In [4]: print a
{}
```

Example on a set of dictionary variables.

```
In [5]: a['one'] = '1'

In [6]: print a
{'one': '1'}

In [7]: a = {'one':'1','two':'2','three':'3','four':'4'}
```

```
In [8]: a
Out[8]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}
```

The order of the key-value pair is not always the same. If we type on different systems we might find different output.

Example

```
In [7]: a = {'one':'1','two':'2','three':'3','four':'4'}

In [8]: a
Out[8]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}
```

Now lets take some examples on how to access each and every value of a dictionary element.

example

```
In [9]: a
Out[9]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}

In [10]: a['four']
Out[10]: '4'
```

If you notice , on giving the key value we are getting the value for the array. But if we lookup for a key , which is not there in the dictionary/hashes we get the below error.

```
In [11]: a['five']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-11-bb0034013dcf> in <module>()
----> 1 a['five']

KeyError: 'five'
```

in operator

we can impletement the in operator on the dictionaries. we can verify if a key is present in the hash or not.

example

```
In [15]: a
Out[15]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}

In [16]: 'four' in a
Out[16]: True

In [17]: 'four' in a
Out[17]: True

In [18]: '1' in a
Out[18]: False

In [19]: '3' in a
Out[19]: False
```

If you notice, we are getting 'True' if not its 'False'.

del operator

we can delete the elements in the dictionary.

```
In [55]: del a['four']

In [56]: a
Out[56]: {'one': '1', 'three': '3', 'two': '2'}
```

9.3.2 Looping in dictionaries

you can use looping to parse via the variable of the dictionary. Lets go via a quick example to understand the same.

Example:

```
In [21]: a
Out[21]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}
```

Output:

```
In [23]: for i in a:
print i,a[i]
 ....:
four 4
three 3
two 2
one 1
```

we can also grab the values in the following way.

```
In [25]: a
Out[25]: {'four': '4', 'one': '1', 'three': '3', 'two': '2'}

In [26]: for k,v in a.items():
 ....:     print(k,v)
 ....:
('four', '4')
('three', '3')
('two', '2')
('one', '1')
```

9.3.3 Lookups

we can always grab the **value** of a dictionary the moment we get the **keys**.

```
value = v[k]
```

similarly, if we want to get key by passing on the values its called as the reverse lookup.

9.3.4 Dictionaries and Tuples

9.3.5 Functions

There are few function in dictionary.

keys

we can use this function to print the keys of the dictionary.

```
In [119]: a
Out[119]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [118]: a.keys()
Out[118]: ['jan', 'apr', 'mar', 'feb']
```

viewkeys

we can use view keys function to print the keys of the dictionary.

```
In [123]: a
Out[123]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [124]: a.viewkeys()
Out[124]: dict_keys(['jan', 'apr', 'mar', 'feb'])
```

iterkeys

we can use the iterkeys function to iterate over the key values.

```
In [138]: a
Out[138]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [140]: for i in a.iterkeys():
  print i
 .....:
jan
apr
mar
feb
```

values

we can use this function to print the values of the dictionary.

```
In [116]: a
Out[116]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [120]: a.values()
Out[120]: ['1', '4', '3', '2']
```

viewvalues

we can use the viewvalues function to print the value of the dictionary.

```
In [123]: a
Out[123]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [122]: a.viewvalues()
Out[122]: dict_values(['1', '4', '3', '2'])
```

itervalues

we can use the itervalues function to print the values of the dictionary.

```
In [142]: a
Out[142]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

 In [141]: for i in a.itervalues():
 print i
 .....:
 1
 4
 3
 2
```

items

This function creates a list of dictionary (key,value) pairs as 2-tuples.

```
In [128]: a
Out[128]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [129]: a.items()
Out[129]: [('jan', '1'), ('apr', '4'), ('mar', '3'), ('feb', '2')]
```

iteritems

we can use the iteritems function to parse through the array items.

```
In [138]: a
Out[138]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [137]: for i,j in a.iteritems():
  print i,j
 .....:
jan 1
apr 4
mar 3
feb 2
```

clear

we can use the clear function to clear the dictionary .

```
In [144]: a
Out[144]: {'apr': '4', 'feb': '2', 'jan': '1', 'mar': '3'}

In [145]: a.clear()

In [146]: a
Out[146]: {}
```

CHAPTER

TEN

UNIT 10: FUNCTIONS

10.1 Introduction

If we want to run a block on code multiple times, we can use the Functions.This feature of calling a block of code multiple times is called as the calling the functions.we can start a function using the defined keyword.

syntax:

```
def my_function()
{
        .. block of code..
}
```

We will discuss more about the function in later topics.

10.1.1 Defining a function

Lets quickly take quick example.

Example:

```
In [1]: def my_function():
...:     print "hello."
...:

In [2]: my_function
Out[2]: <function __main__.my_function>
```

Output:

```
In [3]: my_function()
hello.
```

10.2 Function Parameters

In some cases we need to pass the parameters to the functions. The parameters passed are used by the function for some calulcations or manupulations.

syntax:

```
def my_function(a,b):
{
.. code to use a,b ..
}
```

The values **a** and **b** are function parameters.

Parameters are specified within the pair of parenthesis in the function defination, seperated by commas. Passing the values is also done in the same way.

syntax:

```
my_function(a,b)
```

The values **a** and **b** are called function arguments.

10.2.1 Using Function Parameters

Lets take quick example on understanding the Functional parameters.

Example:

```
In [6]: def great_number(a,b):
if a > b:
    print "%d is greater than %d"  % (a,b)
elif a < b:
    print "%d is less than %d" % (a,b)
else:
    print "%d is equal to %d" % (a,b)
...:
```

Output:

```
In [7]: great_number(2,3)
2 is less than 3

In [8]: great_number(3,2)
3 is greater than 2

In [9]: great_number(3,3)
3 is equal to 3
```

10.3 Local variables

when you declare variable within a function defination, they are within the scope of the function. These variables are called the local variable. These variable names are local to the function. we can call this as the scope of the variable.

10.3.1 Using Local variables

Example:

```
In [11]: a = 10

In [12]: def local_ex(a):
 ....:     print "%d is taken out of scope" % a
```

```
 ....:     a = 5
 ....:     print "%d is the local value in scope" % a
 ....:
```

output:

```
In [13]: a
Out[13]: 10

In [14]: local_ex(a)
10 is taken out of scope
5 is the local value in scope

In [15]: a
Out[15]: 10
```

10.3.2 Using global variables

Sometimes, we can have requirement where we need to use the global variables .There is a function called **global** which we can use within scope of the function.

syntax:

```
global a
```

Example:

```
In [20]: def global_ex():
 ....:     global a
 ....:     print "%d is global variable" %(a)
 ....:     a = 5
 ....:     print "%d is local variable" %(a)
 ....:
```

Output:

```
In [21]: a = 20

In [22]: global_ex()
20 is global variable
5  is local variable
```

10.4 Default argument values

Till now we learned about passing the arguments to the functions.But we might need to keep some arguments to the function as optional . we can do this by specifying the default argument values for parameters.

10.4.1 Using default argument values

Lets go over a quick example to understand the below.

Example:

```
In [27]: def ntimes(message, times=1):
print message * times
....:
```

Output:

```
In [28]: ntimes('hello')
hello

In [29]: ntimes('hero',5)
heroheroheroherohero
```

10.5 Keywords Arguments

Sometimes we can use the keyword inside the function arguments also called as keyword arguments. Here we use the keywords instead of the position to specify the arguments to the functions.

- Here we need not worry about the ordering of the variable in the function.
- we can pass on values to only those parameters which we want.

10.5.1 Using Keyword Arguments

Lets go over a quick example to understand the above concepts.

Example:

```
In [1]: def fun_ex(a,b=10,c=20):
...:     print "The value of a is %d" %(a)
...:     print "The value of b is %d" %(b)
...:     print "The value of c is %d" %(c)
...:
```

Ouput:

```
      In [2]: fun_ex(10)
The value of a is 10
The value of b is 10
The value of c is 20

In [3]: fun_ex(10,c=15)
The value of a is 10
The value of b is 10
The value of c is 15

In [4]: fun_ex(c=10,a=15)
The value of a is 15
The value of b is 10
The value of c is 10
```

10.6 The return statement

The return statement is to return from a function i.e break out of a function. we can optionally return a value from a function as well.

10.6.1 using the return statement

we can use the return statement in our script to quickly understand how its working.

example:

```
In [7]: def a_even(a):
result = a % 2
if result == 0:
    return 'even'
else:
    return 'odd'
...:
```

output:

```
In [8]: a_even(2)
Out[8]: 'even'

In [9]: a_even(3)
Out[9]: 'odd'
```

10.7 Doc strings

Python has a feature called the documention strings which has a shorter name called docstrings. Docstring helps us to document the program information to better understand the function.

10.7.1 using Docstrings

Here is an example of the docstrings.

Example:

```
In [15]: def a_even(a):
''' to return the even number '''
result = a % 2
if result == 0:
    return 'even'
else:
    return 'odd'
....:
```

output:

```
In [16]: print a_even.func_doc
to return the even number
```

10.8 Lambda

The lambda statement allows one to create functions on the fly, i.e., functions that are not bound to a name. It is most commonly used with map(), filter(), etc. Lambda can be used to replace several small functions, and thus save a lot of unnecessary code. Consider the iseven() function defined above. It can be replaced by a lambda statement as follows:

Example:

```
In [33]: b
Out[33]: [1, 2, 3, 4, 5, 6]

In [34]: map( lambda a: a * a , b)
Out[34]: [1, 4, 9, 16, 25, 36]
```

Example:

```
In [38]: b
Out[38]: [1, 2, 3, 4, 5, 6]

In [39]: filter ( lambda a : a % 2 == 0 , b)
Out[39]: [2, 4, 6]
```

10.9 map

Map is a built-in function that takes a function and a list as an argument, applies the function to each element of the list, and returns the output. For instance, map(f, iterA, iterB, ...) returns a list containing f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]),... For example, the following code finds the square of each element in a list:

Example:

```
In [20]: def a_two_mult(a):
....:     return a * 2
....:

In [21]: b = [ 1,2,3,4,5,6]
```

Output:

```
In [22]: map(a_two_mult, b)
Out[22]: [2, 4, 6, 8, 10, 12]
```

10.10 Filter

As the name suggests, filter returns those elements that satisfy a particular condition. For example, the following code filters out all the even numbers:

Example:

```
In [27]: def a_two_mult(a):
return a % 2 == 0
....:

In [28]: b
Out[28]: [1, 2, 3, 4, 5, 6]
```

Output:

```
In [29]: filter (a_two_mult,b)
Out[29]: [2, 4, 6]
```

10.11 Exercises

1 .Define a function max() that takes two numbers as arguments and returns the largest of them. Use the if-then-else construct available in Python. (It is true that Python has the max() function built in, but writing it yourself is nevertheless a good exercise.)

2 .Define a function max_of_three() that takes three numbers as arguments and returns the largest of them.

3 .Write a function max_in_list() that takes a list of numbers and returns the largest one.

4 .Define a function that computes the length of a given list or string. (It is true that Python has the len() function built in, but writing it yourself is nevertheless a good exercise.)

5 .Write a function that takes a character (i.e. a string of length 1) and returns True if it is a vowel, False otherwise.

6 .Define a function sum() and a function multiply() that sums and multiplies (respectively) all the numbers in a list of numbers. For example, sum([1, 2, 3, 4]) should return 10, and multiply([1, 2, 3, 4]) should return 24.

7 .Define a function reverse() that computes the reversal of a string. For example, reverse("I am testing") should return the string "gnitset ma I".

8 .Define a function is_palindrome() that recognizes palindromes (i.e. words that look the same written backwards). For example, is_palindrome("radar") should return True.

9 .Write a function is_member() that takes a value (i.e. a number, string, etc) x and a list of values a, and returns True if x is a member of a, False otherwise. (Note that this is exactly what the in operator does, but for the sake of the exercise you should pretend Python did not have this operator.)

10 .Define a function overlapping() that takes two lists and returns True if they have at least one member in common, False otherwise. You may use your is_member() function, or the in operator, but for the sake of the exercise, you should (also) write it using two nested for-loops.

11 .Define a function generate_n_chars() that takes an integer n and a character c and returns a string, n characters long, consisting only of c:s. For example, generate_n_chars(5,"x") should return the string "xxxxx". (Python is unusual in that you can actually write an expression 5 * "x" that will evaluate to "xxxxx". For the sake of the exercise you should ignore that the problem can be solved in this manner.)

12. Define a procedure histogram() that takes a list of integers and prints a histogram to the screen. For example, histogram([4, 9, 7]) should print the following:

```
****
*********
*******
```

13 .Write a function find_longest_word() that takes a list of words and returns the length of the longest one.

14 .Write a function filter_long_words() that takes a list of words and an integer n and returns the list of words that are longer than n.

15 .A pangram is a sentence that contains all the letters of the English alphabet at least once, for example: The quick brown fox jumps over the lazy dog. Your task here is to write a function to check a sentence to see if it is a pangram or not.

16 .Write a function char_freq() that takes a string and builds a frequency listing of the characters contained in it. Represent the frequency listing as a Python dictionary. Try it with something like char_freq("abbabcbdbabdbdbababababcbcbab").

Write a program which can compute the factorial of a given numbers. The results should be printed in a comma-separated sequence on a single line. Suppose the following input is supplied to the program: 8 Then, the output should be: 40320

Hints: In case of input data being supplied to the question, it should be assumed to be a console input.

CHAPTER

ELEVEN

UNIT 11: MODULES

11.1 Introduction

You have seen how you can reuse code in your program by defining functions once. If we wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules. A module is basically a file containing all your functions and variables that you have defined. To reuse the module in other programs, the filename of the module must have a .py extension. A module can be imported by another program to make use of its functionality.

Lets now try to go through few of the modules function and functionalities to quickly understand about how to use modules,use the function modules . We will also try to use few of the them in the following chapter.

11.1.1 Example using the sys

Lets try to import the sys module.

```
In [4]: import sys
```

The modules provide few of the function and variable which we can use.

```
In [4]: sys.
sys.api_version            sys.exc_info              sys.getcheckinterval       sys.long_info
sys.argv                   sys.exc_type              sys.getdefaultencoding     sys.maxint
sys.builtin_module_names   sys.excepthook            sys.getdlopenflags         sys.maxsize
sys.byteorder              sys.exec_prefix           sys.getfilesystemencoding  sys.maxunicode
sys.call_tracing           sys.executable            sys.getprofile             sys.meta_path
sys.callstats              sys.exit                  sys.getrecursionlimit      sys.modules
sys.copyright              sys.exitfunc              sys.getrefcount            sys.path
sys.displayhook            sys.flags                 sys.getsizeof              sys.path_hooks
sys.dont_write_bytecode    sys.float_info            sys.gettrace               sys.path_importer_ca
sys.exc_clear              sys.float_repr_style      sys.hexversion             sys.platform
```

Lets try to execute few of the function and see how they work

Example:

```
In [6]: sys.version
Out[6]: '2.7.4 (default, Apr 19 2013, 18:28:01) \n[GCC 4.7.3]'

In [7]: sys.path
Out[7]:
['',
'/usr/bin',
'/usr/local/lib/python2.7/dist-packages/Sphinx-1.2b3-py2.7.egg',
```

```
'/usr/lib/python2.7',
'/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk',
'/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PILcompat',
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/pymodules/python2.7',
'/usr/lib/python2.7/dist-packages/ubuntu-sso-client',
'/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode',
'/usr/lib/python2.7/dist-packages/IPython/extensions']
```

The moment we say **import sys** we are telling the python that we want to use the *sys* module. When we call the sys module , it tries to call the sys.py modules in the directories listed in the sys.path variable.

Note, sys is short for 'system'.

11.2 Compiling the .pyc files

Python createsa byte-compiled files with the extension .pyc which is related to the intermediate form that Python transforms the program into (remember the intro section on how Python works ?). This .pyc file is useful when you import the module the next time from a different program - it will be much faster since part of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent. So, now you know what those .pyc files really are.

11.3 The from .. import statement

If you notice, the moment we say **import sys** its imports all the all the modules and variables. But while calling them we use "sys.<variable>* names. To avoid this we can use the **from .. import** statement. Lets take a quick example.

```
In [9]: sys.version
Out[9]: '2.7.4 (default, Apr 19 2013, 18:28:01) \n[GCC 4.7.3]'
```

But we don't want to use **sys.version** we want to use just **version** . Lets try to see how we can work out with it.

```
In [11]: from sys import version

In [12]: version
Out[12]: '2.7.4 (default, Apr 19 2013, 18:28:01) \n[GCC 4.7.3]'
```

Similary if we want to use all variable and modules related to sys without using the **sys** at the beginning we can run the import in the way the example is giving below.

```
In [13]: from sys import *

In [14]: version
Out[14]: '2.7.4 (default, Apr 19 2013, 18:28:01) \n[GCC 4.7.3]'

In [15]: path
Out[15]:
['',
'/usr/bin',
```

```
'/usr/local/lib/python2.7/dist-packages/Sphinx-1.2b3-py2.7.egg',
'/usr/lib/python2.7',
'/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk',
'/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PILcompat',
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/pymodules/python2.7',
'/usr/lib/python2.7/dist-packages/ubuntu-sso-client',
'/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode',
'/usr/lib/python2.7/dist-packages/IPython/extensions']
```

Note: Avoid using the **from .. import** as it may conflict with other module variables.

11.4 A module's __name__

Every module has a name and statements in the module can find out the name of it modules.When a modules is imported for the first time, the main block in the modules is run. If we want the block to be run only the program was used by the block itself and not when imported from other module. We can achieve this using the __name__ modules.

11.4.1 Using a modules __name__

Lets take a quick example to understand the **__name__** concept.

First lets create a quick script

```
1 #!/usr/bin/python
2
3 if __name__ == '__main__':
4   print "The program is being run by itself"
5 else:
6   print "The program is imported from another module"
```

Now if try to execute it

```
santosh-PI945GCM python-examples # python new.py
The program is being run by itself
```

Now lets try to import the module.

```
In [1]: import new
The program is imported from another module
```

Did you notice the **.pyc** file which got created.

```
-rwxr-xr-x 1 root root  151 Dec 22 20:25 new.py
-rw-r--r-- 1 root root  235 Dec 22 20:29 new.pyc
santosh-PI945GCM python-examples # file new.pyc
new.pyc: python 2.7 byte-compiled
santosh-PI945GCM python-examples #
```

11.5 Making your own modules

We can create our own modules. Creating modules is as easy as writing a python program.

11.5.1 Creating your own modules

In this section we will work with the creation of our own modules and try toimplement them with a example.

Example:

```
1 #!/usr/bin/python
2
3 def hello():
4   print "hello this is my first module"
5
6 Version = "1.0"
```

Lets try to import it and start using it. Lets try to start using it.

```
In [1]: import new1

In [2]: new1.hello()
hello this is my first module

In [3]: new1.Version
Out[3]: '1.0'
```

11.5.2 from ..import

Lets implement the **from..import** module on the function we created.

```
In [4]: from new1 import *

In [5]: hello()
hello this is my first module

In [6]: Version
Out[6]: '1.0'
```

11.6 The dir() function

we can use the built-in dir function to list the identifiers that a module defines. The identifiers are the functions, classes and variables defined in that module. When you supply a module name to the dir() function, it returns the list of the names defined in that module. When no argument is applied to it, it returns the list of names defined in the current module.

11.6.1 Using the dir function

Lets try to run the dir() function on module we have created.

```
In [7]: import new1

In [8]: dir(new1)
Out[8]:
['Version',
'__builtins__',
'__doc__',
'__file__',
'__name__',
'__package__',
'hello']

In [9]:
```

11.7 Exercises

1 .Write a program that calculates and prints the value according to the given formula: Q = Square root of [(2 * C * D)/H] Following are the fixed values of C and H: C is 50. H is 30. D is the variable whose values should be input to your program in a comma-separated sequence. Example Let us assume the following comma separated input sequence is given to the program: 100,150,180 The output of the program should be: 18,22,24

Hints: If the output received is in decimal form, it should be rounded off to its nearest value (for example, if the output received is 26.0, it should be printed as 26) In case of input data being supplied to the question, it should be assumed to be a console input.

CHAPTER

TWELVE

UNIT 12: FILES

12.1 Fancier Output Formatting

The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, str() will return the same value as repr(). Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

```
In [2]: for x in range(10):
  print repr(x).rjust(2),repr(x*x).rjust(3),repr(x**3).rjust(4)
 ...:
0   0    0
1   1    1
2   4    8
3   9   27
4  16   64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
```

Same kind of output we can formate using the the `format` option also.

```
In [5]: for x in range(10):
print '{0:2d} {1:3d} {2:3d}'.format(x,x*x,x*x*x)
 ...:
0   0   0
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729

In [6]: print '{} is a {} technology'.format('python','opensource')
python is a opensource technology
```

We can use indexing also in the formate statement.

```
In [7]: print '{0} is a {1} technology'.format('python','opensource')
python is a opensource technology

In [8]: print '{1} is a {0} technology'.format('python','opensource')
opensource is a python technology
```

we can also give them names in the formate statements.

```
In [9]: print '{lang} is a {tech} technology'.format(lang='python',tech='opensource')
python is a opensource technology

In [12]: print '{!s} is a great day'.format('monday')
monday is a great day

In [13]: print '{!r} is a great day'.format('monday')
'monday' is a great day

In [14]: print '{} is a great day'.format('monday')
monday is a great day

In [17]: import math

In [18]: math.pi
Out[18]: 3.141592653589793

In [19]: print 'The value of pi is approximately {0:.3f}.'.format(math.pi)
The value of pi is approximately 3.142.
```

Few more options you can look into `str.ljust()` , `str.rjust` , `str.centre()` . These methods do not write anything , they just return a new string. If the string is too long , they don't truncate it, but return it unchanged.

12.2 Reading and writing files

open returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
In [3]: f = open('/tmp/newfile.txt','w')
In [4]: print f
<open file '/tmp/newfile.txt', mode 'w' at 0x22bf270>
In [5]: type(f)
Out[5]: file
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used.

There are various modes.

- `'r'` when the file will only be read .
- `'w'` for only writing (an existing file with the same name will be erased).
- `'a'` opens the file for appending; any data written to the file is automatically added to the end.
- `'r+'` opens the file for both reading and writing.

The *mode* argument is optional; `'r'` will be assumed if it's omitted.

```
In [6]: f = open('/tmp/newfile.txt')
In [7]: type(f)
Out[7]: file
```

```
In [8]: print f
<open file '/tmp/newfile.txt', mode 'r' at 0x22bf300>
```

On Windows, `'b'` appended to the mode opens the file in binary mode. There are also modes like `'rb'`, `'wb'`, and `'r+b'`. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in `JPEG` or `EXE` files. Be very careful to use binary mode when reading and writing such files. On Unix,it doesn't hurt to append a `'b'` to the mode, so you can use it platform-independently for all binary files.

12.3 Methods of File Objects

12.3.1 Reading

`f.read`

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

Content of the file

```
In [16]: ! cat /tmp/newfile.txt
Hello today is a great day
```

Few point to quickly oberve in the below example.

- `open` open the file and assigns it to a new filehandle `f`.
- `f.close()` closes the file handle.
- `f.read(4)` reads four bits of the file.
- `f.read()` completely reads the file.

```
In [23]: f = open('/tmp/newfile.txt')

In [24]: type(f)
Out[24]: file

In [25]: print f
<open file '/tmp/newfile.txt', mode 'r' at 0x22bf300>

In [26]: f.read(4)
Out[26]: 'Hell'

In [27]: f.read(7)
Out[27]: 'o today'

In [28]: f.close()
```

`f.readine()`

`f.readline()` reads a single line from the file; 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. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

Lets take a file called as `/tmp/newfile.txt`, lets see the contents of the file.

```
In [33]: ! cat /tmp/newfile.txt
Hello today is a great day
Hello this is line 2.
Hello this is line 3.
Hello this is line 4.
Hello this is line 5.
Hello this is line 6.
```

Let's try now to use the `readline` function.

```
In [34]: f = open('/tmp/newfile.txt','r')

In [35]: print f
<open file '/tmp/newfile.txt', mode 'r' at 0x22bf4b0>

In [36]: print f.readline()
Hello today is a great day


In [37]: print f.readline()
Hello this is line 2.
```

One more quick way of reading lines from a file is loop over the file object. This is memory efficient, fast, and leads to simple code

```
In [38]: for line in f:
 ....:     print line
 ....:
Hello this is line 3.

Hello this is line 4.

Hello this is line 5.

Hello this is line 6.
```

`readlines()`

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

Example on `readlines()`

```
In [40]: f = open('/tmp/newfile.txt','r')

In [41]: f.readlines()
Out[41]:
['Hello today is a great day\n',
'Hello this is line 2.\n',
'Hello this is line 3.\n',
'Hello this is line 4.\n',
'Hello this is line 5.\n',
'Hello this is line 6.\n']
```

Example on `list(f)`

```
In [44]: list(f)
Out[44]:
['Hello today is a great day\n',
'Hello this is line 2.\n',
```

```
'Hello this is line 3.\n',
'Hello this is line 4.\n',
'Hello this is line 5.\n',
'Hello this is line 6.\n']
```

12.3.2 Writing

To open a file and write content into the file, we need to open it in the write mode.

```
f = open("/tmp/newfile.txt","w")
```

Lets go over a quick example to quickly write into the file.

`f.write(string)`

writes the contents of *string* to the file, returning `None`.

```
In [46]: ! cat /tmp/newfile.txt
Hello today is a great day
Hello this is line 2.
Hello this is line 3.
Hello this is line 4.
Hello this is line 5.
Hello this is line 6.

In [47]: f = open('/tmp/newfile.txt','w')

In [48]: f.write("Hello this is line number 1.\n Hello this is line number 2.\n Hello this is line nu

In [49]: f.close()

In [50]: !cat /tmp/newfile.txt
Hello this is line number 1.
Hello this is line number 2.
Hello this is line number 3.
```

12.3.3 Modify

Lets try to modify the contents of the above file `/tmp/newfile.txt`.

```
In [63]: !cat /tmp/newfile.txt
Hello this is line number 1.
Hello this is line number 2.
Hello this is line number 3.
```

Lets try to read the contents of the above file into an array.

```
In [67]: f = open('/tmp/newfile.txt','r')

In [68]: list_value = f.readlines()

In [69]: list_value
Out[69]:
['Hello this is line number 1.\n',
' Hello this is line number 2.\n',
' Hello this is line number 3.\n']
```

```
In [70]: f.close()
```

Now lets try to change one of the statements.

```
In [71]: list_value
Out[71]:
['Hello this is line number 1.\n',
' Hello this is line number 2.\n',
' Hello this is line number 3.\n']

In [72]: list_value[2]
Out[72]: ' Hello this is line number 3.\n'

In [73]: list_value[2] = ' Hello i am chaning the number to 4.\n'

In [74]: list_value
Out[74]:
['Hello this is line number 1.\n',
' Hello this is line number 2.\n',
' Hello i am chaning the number to 4.\n']
```

Now lets try to put these changes into a file.

```
In [75]: f = open('/tmp/newfile.txt','w')

In [76]: f.writelines(list_value)

In [77]: f.close()

In [78]: ! cat /tmp/newfile.txt
Hello this is line number 1.
Hello this is line number 2.
Hello i am chaning the number to 4.
```

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

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 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* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

12.4 Pickles

Pickle help in saving a object to a file. To work with pickle you need to first import the pickle object. Input using Pickle¶

```
In [4]: import pickle
In [5]:
```

First lets create a object to use it along with pickle. To continue with the pickle lets try to go with an example.

```
In [5]: training_list = ["Linux","Python","Perl","shell"]

In [6]: training_list
Out[6]: ['Linux', 'Python', 'Perl', 'shell']

In [13]: outfile = open('/tmp/new.txt','wb')
In [14]: pickle.dump(training_list,outfile)

In [14]: pickle.dump(training_list,outfile)

In [15]: outfile.close()

In [16]: ! cat /tmp/new.txt
(lp0
S'Linux'
p1
aS'Python'
p2
aS'Perl'
p3
aS'shell'
p4
a.
```

If you notice the file /tmp/new.txt is in form of a binary file, so it won't be in term of normal text file.

12.5 Output using Pickle

Let now try to read the content from the pickled object /tmp/new.txt.

```
In [18]: import pickle

In [19]: infile  = open('/tmp/new.txt','rb')

In [20]: newlist = pickle.load(infile)

In [21]: newlist

Out[21]: ['Linux', 'Python', 'Perl', 'shell']

In [24]: infile.close()
```

Note: One issue with is pickle only put one object in a file . If you try to load multipe object it gets messed out. Better option is to put muliple object in a list and pickle it.

CHAPTER

THIRTEEN

UNIT 13: EXCEPTIONS

13.1 Exceptions

More often than not, a program is written to address a certain problem. Whenever Python throws an error on a program that is programmed with proper Python syntax, it has hit upon an exception. While your data may vary, you can program Python to expect variability in the data. Using the information presented in this tutorial, your Python programs will cope quite nicely and deal with failures in theway that you decide.

13.2 Simulating Errors

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

To simulate an error lets try to print a string which is not defined.

Example 1: NameError

```
In [1]: print string
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-9d3f5b3573df> in <module>()
----> 1 print string

NameError: name 'string' is not defined
```

Example 2: ZeroDivisionError

```
In [3]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: integer division or modulo by zero
```

13.3 Various Exceptions

There are lots of exceptions that python handles. Lets try to understand how to get to see the various exceptions.

First we need to import the exceptions.

```
In [4]: import exceptions
```

If we run dir on the exception we get to see various types of exceptions supported by python.

```
In [5]: dir(exceptions)
Out[5]:
['ArithmeticError',
'AssertionError',
'AttributeError',
'BaseException',
'BufferError',
'BytesWarning',
'DeprecationWarning',
'EOFError',
'EnvironmentError',
'Exception',
'FloatingPointError',
'FutureWarning',
'GeneratorExit',
'IOError',
'ImportError',
'ImportWarning',
'IndentationError',
'IndexError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'NameError',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'ReferenceError',
'RuntimeError',
'RuntimeWarning',
'StandardError',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__doc__',
'__name__',
'__package__']
```

13.4 Cleanup the errors

Lets try to check our errors and try to close them up.

Lets go back to our examples

Example 1: NameError

```
In [1]: print string
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-9d3f5b3573df> in <module>()
----> 1 print string

NameError: name 'string' is not defined
```

Now lets try to remove these verbose messages. Now lets introduce ourselfs to try .. except

Solution 1: using try and except

In the following case what ever be the exception type we are going to get message.

```
In [7]: try:
 ...:     print string
 ...: except:
 ...:     print "Please provide a defined string value"
 ...:
Please provide a defined string value
```

Solution 2: using try and except with an exception

```
In [11]: try:
 ....:     print string
 ....: except NameError:
 ....:     print "Please provide a defined string value"
 ....:
Please provide a defined string value
```

Now lets try to sort out the example 2.

Example 2: ZeroDivisionError

```
In [3]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: integer division or modulo by zero
```

Now lets try to remove these verbose messages. Now lets introduce ourselfs to try .. except

Solution 1: using try and except

```
In [13]: try:
 ....:     1/0
 ....: except:
 ....:     print "Cannot divide using zero"
 ....:
Cannot divide using zero
```

Solution 2: Using try and except with exceptions.

```
In [14]: try:
 ....:     1/0
 ....: except ZeroDivisionError:
 ....:     print "Cannot divide using zero"
 ....:
Cannot divide using zero
```

13.5 More than one Exceptions

Now that we can catch errors and turn them into simple exceptions. The question now is how do you catch more than one kind of error.We can use multiple except statements.

Lets go back to our **example 2** of dividing by 0 and turn the division equation into variables instead of values, we should check for both attribution of value and zero division.

Example 1: Creation of errors

First lets create a small program as below.

```
#!/usr/bin/python

value1 = input("enter a number1:")
value2 = input("enter a number2:")

print "division of the two string:" , value1/value2
print "division by zero(value2): " , value1/value2
```

In the first case, lets enter a alphabet instead of a number.

Error 1: Entering a alphabets instead of a number.

```
In [22]: run /python-examples/ex1.py
enter a number1:a
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/python2.7/dist-packages/IPython/utils/py3compat.pyc in execfile(fname, *where)
  176             else:
  177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/python-examples/ex1.py in <module>()
    1 #!/usr/bin/python
    2
----> 3 value1 = input("enter a number1:")
    4 value2 = input("enter a number2:")
    5

<string> in <module>()

NameError: name 'a' is not defined
```

Error 2: Entering one of numbers a zero.

```
In [24]: run /python-examples/ex1.py
enter a number1:2
enter a number2:0
division of the two string:---------------------------------------------------------------------------
```

```
ZeroDivisionError                         Traceback (most recent call last)
/usr/lib/python2.7/dist-packages/IPython/utils/py3compat.pyc in execfile(fname, *where)
  176             else:
  177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/python-examples/ex1.py in <module>()
    4 value2 = input("enter a number2:")
    5
----> 6 print "division of the two string:" , value1/value2
    7 print "division by zero(value2): " , value1/value2

ZeroDivisionError: integer division or modulo by zero
```

Solution : Now lets try to solve these two errors in our program.

```
#!/usr/bin/python
1 try:
2   a = input("Please enter the dividend (the number to be divided):")
3   b = input("Please enter the divisor (the number by which to divide):")
4 except NameError:
5   print "Please enter numbers only"
6
7 try:
8  print a/b
9 except NameError:
10  print "Both values must be integers!"
11 except ZeroDivisionError:
12  print "The divisor must not be zero. Division by null is not allowed."
```

Test runs:

```
In [3]: run ex2.py
Please enter the dividend (the number to be divided):a
Please enter numbers only
Both values must be integers!

In [4]: run ex2.py
Please enter the dividend (the number to be divided):1
Please enter the divisor (the number by which to divide):b
Please enter numbers only
Both values must be integers!

In [5]: run ex2.py
Please enter the dividend (the number to be divided):1
Please enter the divisor (the number by which to divide):0
The divisor must not be zero. Division by null is not allowed.
```

we can perform the same action on multiple errors, we can combine the errors into a single argument using expect.

```
try:
print a/b
except (NameError,ZeroDivisionError):
print "Both values must be integers!, can't divide by zero."
```

13.6 Taking Exceptions

While it is all well and good to handle an exception and do something predictable when it happens, what if you want to process the error as data, perhaps printing it in a bold HTML typeface? Simply pass to except the variable by which you want to reference the error data.

```
1 #!/usr/bin/python
2
3 try:
4   a = input("Please enter the dividend (the number to be divided):")
5   b = input("Please enter the divisor (the number by which to divide):")
6 except NameError,error:
7   print "Please enter numbers only"
8
9 print "The error is %s" %(error)
```

13.7 How to handle without any errors

We may want Python to behave one way if there is an exception and another way if no error occurred. In this case, you will want to use a structure of try...except...else.

Example: An example on using try..except..else

```
 1 #!/usr/bin/python
 2
 3 try:
 4   a = input("Please enter the dividend (the number to be divided):")
 5   b = input("Please enter the divisor (the number by which to divide):")
 6 except NameError,error:
 7   print "Please enter numbers only \n"
 8   print "The error is %s" %(error)
 9 else:
 10  print "Good!!! you entered both as numbers \n"

Output:

santosh-PI945GCM python-examples # python ex4.py
Please enter the dividend (the number to be divided):6
Please enter the divisor (the number by which to divide):2
Good!!! you entered both as numbers
```

13.8 Raising Exceptions

The standard way by which Python "handles" exceptions is by throwing an error of some sort. If the program is not coded to "catch" the error and do something with it, the program usually crashes, as we saw earlier. However, even if the program works just as you intended, you can program in your own exceptions using the raise statement.

```
In [5]: raise NameError
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-a586bde47246> in <module>()
----> 1 raise NameError
```

```
NameError:

In [6]: raise SyntaxError
File "<string>", line unknown
SyntaxError
```

we can use any kind of error that is included in the module exceptions.we can also raise our own exceptions. However, the exception raised must be a class or an instance of a class. Anything else will result in a TypeError. If we do not raise in a class we get a TypeError.

```
In [7]: raise
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-7-26814ed17a01> in <module>()
----> 1 raise

TypeError: exceptions must be old-style classes or derived from BaseException, not NoneType
```

Example: An example using the classes.

```
1 #!/usr/bin/env python
2
3 age = raw_input("please enter the age 1-150:")
4
5 class InvalidAgeRangeException(Exception):
6   def __init__(self,age):
7     self.age = age
8
9 def validate_age(age):
10   if age < 0 or age > 150:
11     raise InvalidAgeRangeException(age)
12
13 try:
14   age=int(age)
15   validate_age(age)
16 except ValueError as e:
17   print e
18 except InvalidAgeRangeException as e:
19   print 'Invalid age range, you entered : %s' % e.age
```

Output:

```
santosh-PI945GCM python-examples # python ex5.py
please enter the age 1-150:250
Invalid age range, you entered : 250
```

Whenever an exception is raised in the code, Python will raise that error even if it sees nothing wrong. In this way, the programs can be stricter than the Python interpreter. For example, you have a set of options and would like a NameError error to be thrown if the user chooses an option not listed. Python will do that as part of the exception handling process, instead of throwing a KeyError or something similar.

13.9 Customised Exceptions

Rather than raising exceptions like NameError or IOError. We can also provide a way to present customised error messages by using the build-in class exceptions. We can use "Exceptions" as the arguments of raise.

```
 In [9]: raise Exception
 ---------------------------------------------------------------------------
 Exception                                 Traceback (most recent call last)
 <ipython-input-9-fca2ab0ca76b> in <module>()
 ----> 1 raise Exception

 Exception:
```

we could raise a simple error by not giving any further arguments than "Exception". If you want to pr

```
In [11]: raise Exception , "Hello you made a warning !!!"
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-11-126717561ff3> in <module>()
----> 1 raise Exception , "Hello you made a warning !!!"

Exception: Hello you made a warning !!!
```

CHAPTER

FOURTEEN

UNIT 14: REGULAR EXPRESSIONS

14.1 Introduction

14.1.1 Import re

In python regular expression are not loaded by default. We need to import regular expression .

syntax:

```
In [1]: import re
```

14.1.2 Object os re

Once we import the **re** module , we get some object. We can check on the objects in couple of ways.

TYPE I: Run *dir(re)* to see all the modules .

```
In [2]: dir(re)
Out[2]:
['DEBUG',
'DOTALL',
'I',
'IGNORECASE',
'L',
'LOCALE',
'M',
'MULTILINE',
'S',
'Scanner',
'T',
'TEMPLATE',
'U',
'UNICODE',
'VERBOSE',
'X',
'_MAXCACHE',
'__all__',
'__builtins__',
'__doc__',
'__file__',
'__name__',
'__package__',
'__version__',
```

```
'_alphanum',
'_cache',
'_cache_repl',
'_compile',
'_compile_repl',
'_expand',
'_pattern_type',
'_pickle',
'_subx',
'compile',
'copy_reg',
'error',
'escape',
'findall',
'finditer',
'match',
'purge',
'search',
'split',
'sre_compile',
'sre_parse',
'sub',
'subn',
'sys',
'template']
```

TYPE II: Since we are in ipython, we can easily hit tab after *re.<tab>* .

```
In [3]: re.
re.DEBUG        re.S            re.compile      re.search
re.DOTALL       re.Scanner      re.copy_reg     re.split
re.I            re.T            re.error        re.sre_compile
re.IGNORECASE   re.TEMPLATE     re.escape       re.sre_parse
re.L            re.U            re.findall      re.sub
re.LOCALE       re.UNICODE      re.finditer     re.subn
re.M            re.VERBOSE      re.match        re.sys
re.MULTILINE    re.X            re.purge        re.template
```

14.1.3 Getting Started.

Now lets go through a quick example on understanding the match operator. We will be working more on match in later chapters. For now lets go through the example to quickly understand the python regular expression complilation.

STEP I: Compiling the regular expression:

```
In [3]: reg = re.compile("python")

In [4]: type(reg1)
Out[4]: _sre.SRE_Pattern
```

STEP II: Help on the match operator:

```
In [5]: reg.match?
Type:       builtin_function_or_method
String Form:<built-in method match of _sre.SRE_Pattern object at 0x2e29490>
Docstring:
match(string[, pos[, endpos]]) --> match object or None.
Matches zero or more characters at the beginning of the string
```

STEP III : Now lets try to match the expression.

```
In [8]: reg.match("python")
Out[8]: <_sre.SRE_Match at 0x2d6ced0>

In [13]: print reg.match("pytho")
None
```

If you check it now, **<_sre.SRE_Match at 0x2d6ced0>** says that the regular expression is a match. If the regular expression is not a match we get **None**.

STEP IV : To verify our match is printed we can use **group()** to get it printed.

```
In [10]: print reg.match("python").group()
python
```

Note we have not worked on how to handle the case sensitive issue yet. We will cover it in coming topics.

```
In [14]: print reg.match("PYTHON")
None
```

Lets put all of this into a file called **python_re1.py** .

Quick code

```
1 #!/usr/bin/python
2 import re
3
4 string = 'python'
5 reg = re.compile(string)
6 matchs = reg.match('python')
7 print matchs
8 print "\n";
9 print "Now lets print the matches word \n"
10 print matchs.group()
```

14.2 Special Characters

The special characters are:

`'^'` (Caret.) Matches the start of the string.

Example:

```
In [31]: string = "This is python class"
In [34]: reg = re.compile("^This",re.IGNORECASE)
In [32]: match = reg.match(string)
In [36]: print match
<_sre.SRE_Match object at 0x2e89440>
In [37]: print match.group()
This
```

One more way of doing it :

```
In [1]: string = "This is python class"
In [2]: import re
In [3]: re.match("^This",string).group()
Out[3]: 'This'
```

'$' Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline.

Example:

```
In [18]: import re
In [19]: string = "foo foobar"
In [20]: reg = re.compile("foobar$")
In [21]: reg.match(string)
```

'.' (Dot.) In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.

```
In [12]: string="This"
In [13]: re.search("....",string)
Out[13]: <_sre.SRE_Match at 0x1fce1d0>
In [14]: re.search("....",string).group()
Out[14]: 'This'
```

'*' Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. ab* will match 'a', 'ab', or 'a' followed by any number of 'b's.

```
In [20]: string
Out[20]: 'This'

In [21]: re.search(".*",string).group()
Out[21]: 'This'
```

'+' Causes the resulting RE to match 1 or more repetitions of the preceding RE. ab+ will match 'a' followed by any non-zero number of 'b's; it will not match just 'a'.

```
In [22]: string = "abbba"
In [23]: re.match("ab+a",string).group()
Out[23]: 'abbba'
```

'?' Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. ab? will match either 'a' or 'ab'.

```
In [24]: string = "hello"
In [25]: re.match(r'hello?',string).group()
Out[25]: 'hello'
In [26]: string = "hell"
In [27]: re.match(r'hello?',string).group()
Out[27]: 'hell'
```

***?, +?, ??** The '*', '+', and '?' qualifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE <.*> is matched against '<H1>title</H1>', it will match the entire string, and not just '<H1>'. Adding '?' after the qualifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using .*? in the previous expression will match only '<H1>'.

If we below the code comes as follows:

```
In [28]: string="<H1>title</H1>"
In [29]: re.match(r'<.*>',string)
Out[29]: <_sre.SRE_Match at 0x1fce370>
In [30]: re.match(r'<.*>',string).group()
Out[30]: '<H1>title</H1>'
```

To get the output as required, we can do the following.

```
In [31]: re.match(r'<.*?>',string).group()
Out[31]: '<H1>'
```

{m} Specifies that exactly *m* copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, `a{6}` will match exactly six `'a'` characters, but not five.

```
In [13]: string="aaashique"
In [14]: re.match('a{3}shique',string).group()
Out[14]: 'aaashique'
```

If you notice there is error in the below code as string does not match the number of a's .

```
In [15]: re.match('a{2}shique',string).group()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-70518e6d94b5> in <module>()
----> 1 re.match('a{2}shique',string).group()

AttributeError: 'NoneType' object has no attribute 'group'
```

{m,n} Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, `a{3,5}` will match from 3 to 5 `'a'` characters. Omitting *m* specifies a lower bound of zero, and omitting *n* specifies an infinite upper bound. As an example, `a{4,}b` will match `aaaab` or a thousand `'a'` characters followed by a `b`, but not `aaab`. The comma may not be omitted or the modifier would be confused with the previously described form.

Notice the numbers of a's in the string is **aaashique** .

```
In [17]: string="aaashique"
In [18]: re.match('a{2,3}shique',string).group()
Out[18]: 'aaashique'
```

Notice the numbers of a's in the string is **aashique** .

```
In [19]: string="aashique"
In [20]: re.match('a{2,3}shique',string).group()
Out[20]: 'aashique'
```

{m,n}? Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string `'aaaaaa'`, `a{3,5}` will match 5 `'a'` characters, while `a{3,5}?` will only match 3 characters.

```
In [31]: string="aaaaaa"
In [32]: re.match('a{2,3}',string).group()
Out[32]: 'aaa'
In [33]: re.match('a{2,3}?',string).group()
Out[33]: 'aa'
```

'\' Either escapes special characters (permitting you to match characters like `'*'`, `'?'`, and so forth), or signals a special sequence; special sequences are discussed below.

If you're not using a raw string to express the pattern, remember that Python also uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by Python's parser, the backslash and subsequent character are included in the resulting string. However, if Python would recognize the resulting sequence, the backslash should be repeated twice. This is complicated and hard to understand, so it's highly recommended that you use raw strings for all but the simplest expressions.

```
In [41]: string = "<H*>test<H*>"
In [42]: re.match('<H\*>',string).group()
Out[42]: '<H*>'
```

[] Used to indicate a set of characters. In a set:

- Characters can be listed individually, e.g. `[amk]` will match `'a'`, `'m'`, or `'k'`.

```
In [45]: string = "hello"
In [46]: re.match("h[eE]llo",string).group()
Out[46]: 'hello'

In [47]: string = "hEllo"
In [48]: re.match("h[eE]llo",string).group()
Out[48]: 'hEllo'
```

- Ranges of characters can be indicated by giving two characters and separating them by a `'-'`, for example `[a-z]` will match any lowercase ASCII letter, `[0-5][0-9]` will match all the two-digits numbers from `00` to `59`, and `[0-9A-Fa-f]` will match any hexadecimal digit. If `-` is escaped (e.g. `[a\-z]`) or if it's placed as the first or last character (e.g. `[a-]`), it will match a literal `'-'`.

```
In [56]: string = "hEllo"

In [57]: re.match("[a-z].*",string).group()
Out[57]: 'hEllo'
```

- Special characters lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.

```
In [63]: string="<h*>test<h*>"

In [64]: re.match("<h[*]>",string).group()
Out[64]: '<h*>'
```

- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will both match a parenthesis.

\number Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+) \1` matches `'the the'` or `'55 55'`, but not `'thethe'` (note the space after the group). This special sequence can only be used to match one of the first 99 groups. If the first digit of *number* is 0, or *number* is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value *number*. Inside the `'['` and `']'` of a character class, all numeric escapes are treated as characters.

\A Matches only at the start of the string.

\b Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore character. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning/end of the string, so the precise set of characters deemed to be alphanumeric depends on the values of the `UNICODE` and `LOCALE` flags. For example, `r'\bfoo\b'` matches `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` but not `'foobar'` or `'foo3'`. Inside a character range, `\b` represents the backspace character, for compatibility with Python's string literals.

\B Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'py\B'` matches `'python'`, `'py3'`, `'py2'`, but not `'py'`, `'py.'`, or `'py!'`. `\B` is just the opposite of `\b`, so is also subject to the settings of `LOCALE` and `UNICODE`.

\d When the `UNICODE` flag is not specified, matches any decimal digit; this is equivalent to the set `[0-9]`. With `UNICODE`, it will match whatever is classified as a decimal digit in the Unicode character properties database.

\D When the `UNICODE` flag is not specified, matches any non-digit character; this is equivalent to the set `[^0-9]`. With `UNICODE`, it will match anything other than character marked as digits in the Unicode character properties database.

\s When the UNICODE flag is not specified, it matches any whitespace character, this is equivalent to the set [\t\n\r\f\v]. The LOCALE flag has no extra effect on matching of the space. If UNICODE is set, this will match the characters [\t\n\r\f\v] plus whatever is classified as space in the Unicode character properties database.

\S When the UNICODE flags is not specified, matches any non-whitespace character; this is equivalent to the set [^ \t\n\r\f\v] The LOCALE flag has no extra effect on non-whitespace match. If UNICODE is set, then any character not marked as space in the Unicode character properties database is matched.

\w When the LOCALE and UNICODE flags are not specified, matches any alphanumeric character and the underscore; this is equivalent to the set [a-zA-Z0-9_]. With LOCALE, it will match the set [0-9_] plus whatever characters are defined as alphanumeric for the current locale. If UNICODE is set, this will match the characters [0-9_] plus whatever is classified as alphanumeric in the Unicode character properties database.

\W When the LOCALE and UNICODE flags are not specified, matches any non-alphanumeric character; this is equivalent to the set [^a-zA-Z0-9_]. With LOCALE, it will match any character not in the set [0-9_], and not defined as alphanumeric for the current locale. If UNICODE is set, this will match anything other than [0-9_] plus characters classied as not alphanumeric in the Unicode character properties database.

\Z Matches only at the end of the string.

14.3 Module Contents

14.3.1 Ignore Case (re.IGNORECASE)

Now lets work on how to ignore the case. In python regular expression we have the "re.IGNORECASE" to ignore the case.

STEP I: To ignore the case we need to use **re.IGNORECASE** while compiling.

```
In [17]: reg = re.compile("python",re.IGNORECASE)
```

STEP II: Lets try to match the the **python** using the **PYTHON**

```
In [18]: match = reg.match("PYTHON")
```

STEP III : If you notice we got a re match object, so its a MATCH.

```
In [19]: print match
<_sre.SRE_Match object at 0x2e7ced0>
```

STEP IV: Now lets try to print it using the **group()** operator.

```
In [20]: print match.group()
PYTHON
```

14.3.2 re.DOTALL

Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.

If you notice in the below example, we have defined multiline string.

```
In [8]: string = "Today is a sunday.\nTomorrow is monday"

In [9]: print string
```

```
Today is a sunday.
Tomorrow is monday
```

If you check in below, you can see the string match only gives one line.

```
In [12]: reg = re.compile('.*')

In [13]: reg.match(string)
Out[13]: <_sre.SRE_Match at 0x1e58e00>

In [14]: reg.match(string).group()
Out[14]: 'Today is a sunday.'
```

Let try using the re.DOTALL now.

```
In [15]: string = "Today is a sunday.\nTomorrow is monday"

In [16]: print string
Today is a sunday.
Tomorrow is monday

In [17]: reg = re.compile('.*',re.DOTALL)

In [18]: reg.match(string)
Out[18]: <_sre.SRE_Match at 0x1f27e00>

In [19]: reg.match(string).group()
Out[19]: 'Today is a sunday.\nTomorrow is monday'
```

If you had noticed, both the lines getting printed.

14.4 Search vs Match

Python offers two different primitive operations based on regular expressions: `re.match()` checks for a match only at the beginning of the string, while `re.search()` checks for a match anywhere in the string (this is what Perl does by default).

For example:

```
>>> re.match("c", "abcdef")   # No match
>>> re.search("c", "abcdef") # Match
<_sre.SRE_Match object at ...>
```

Regular expressions beginning with `'^'` can be used with `search()` to restrict the match at the beginning of the string:

```
>>> re.match("c", "abcdef")   # No match
>>> re.search("^c", "abcdef") # No match
>>> re.search("^a", "abcdef")  # Match
<_sre.SRE_Match object at ...>
```

Note however that in `MULTILINE` mode `match()` only matches at the beginning of the string, whereas using `search()` with a regular expression beginning with `'^'` will match at the beginning of each line.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<_sre.SRE_Match object at ...>
```

14.5 Raw String Notation

Raw string notation (`r"text"`) keeps regular expressions sane. Without it, every backslash (`'\'`) in a regular expression would have to be prefixed with another one to escape it. For example, the two following lines of code are functionally identical:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object at ...>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object at ...>
```

When one wants to match a literal backslash, it must be escaped in the regular expression. With raw string notation, this means `r"\\"`. Without raw string notation, one must use `"\\\\"`, making the following lines of code functionally identical:

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object at ...>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object at ...>
```

14.6 Grouping

MatchObject.group([group1, ...])

> Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *group1* defaults to zero (the whole match is returned). If a *groupN* argument is zero, the corresponding return value is the entire matching string; if it is in the inclusive range [1..99], it is the string matching the corresponding parenthesized group. If a group number is negative or larger than the number of groups defined in the pattern, an `IndexError` exception is raised. If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned.
>
> ```
> >>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
> >>> m.group(0)       # The entire match
> 'Isaac Newton'
> >>> m.group(1)       # The first parenthesized subgroup.
> 'Isaac'
> >>> m.group(2)       # The second parenthesized subgroup.
> 'Newton'
> >>> m.group(1, 2)    # Multiple arguments give us a tuple.
> ('Isaac', 'Newton')
> ```
>
> If the regular expression uses the `(?P<name>...)` syntax, the *groupN* arguments may also be strings identifying groups by their group name. If a string argument is not used as a group name in the pattern, an `IndexError` exception is raised.
>
> A moderately complicated example:
>
> ```
> >>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
> >>> m.group('first_name')
> 'Malcolm'
> >>> m.group('last_name')
> 'Reynolds'
> ```
>
> Named groups can also be referred to by their index:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

If a group matches multiple times, only the last match is accessible:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

groups([default])

Return a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. (Incompatibility note: in the original Python 1.5 release, if the tuple was one element long, a string would be returned instead. In later versions (from 1.5.1 on), a singleton tuple is returned in such cases.)

For example:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

If we make the decimal place and everything after it optional, not all groups might participate in the match. These groups will default to `None` unless the *default* argument is given:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

groupdict([default])

Return a dictionary containing all the *named* subgroups of the match, keyed by the subgroup name. The *default* argument is used for groups that did not participate in the match; it defaults to `None`. For example:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

start([group])

MatchObject.end([group])

Return the indices of the start and end of the substring matched by *group*; *group* defaults to zero (meaning the whole matched substring). Return `-1` if *group* exists but did not contribute to the match. For a match object *m*, and a group *g* that did contribute to the match, the substring matched by group *g* (equivalent to `m.group(g)`) is

```
m.string[m.start(g):m.end(g)]
```

Note that `m.start(group)` will equal `m.end(group)` if *group* matched a null string. For example, after `m = re.search('b(c?)', 'cba')`, `m.start(0)` is 1, `m.end(0)` is 2, `m.start(1)` and `m.end(1)` are both 2, and `m.start(2)` raises an `IndexError` exception.

An example that will remove *remove_this* from email addresses:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
```

```
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

CHAPTER

FIFTEEN

UNIT 15: CLASSES

15.1 Introduction to Classes

Till now we have worked mostly on the procedure oriented way of programming. We will be convering the object oriented programming in the following sessions. Classes and objects are two main part of object oriented programming.

Object is a thing which contains data and had methods. A Method is nothing but a function. Object has build in attributes to describe it.

For working with object we need to build a class. In short a class is nothing but a blue print for Objects.

Lets quickly summarise the whole concept quickly

- Variable that belong to a object or class are called fields.
- Objects can have some functions associated and they are called methods.
- Both the fields and the methods are refered to as attributes of the class.

Lets go ahead creating a class.

Syntax:

```
class Classname:
```

Example:

```
In [1]: class newclass:
 ...:     course1='python'
 ...:     type_course='Basic'
 ...:     def course_location(self):
 ...:         print "Hey we are working in this location'
 ...:
```

If you have noticed we have just created a class:

```
In [4]: newclass()
Out[4]: <__main__.newclass instance at 0x19be050>
```

The above statement says that we have instance of newclass in the **__main__** modules. Also we can see the address of of the computer memory where the object is stored.

Now lets try to extrace the fields and methods with in the class.For this lets create a object called example_object:

```
In [6]: type(example_object)
Out[6]: instance

In [7]: type(newclass())
Out[7]: instance
```

```
In [8]: example_object.course1
Out[8]: 'python'

In [9]: example_object.type_course
Out[9]: 'Basic'

In [10]: example_object.course_location()
Hey we are working in this location
```

15.2 Understanding Classes

Lets go bit more into understand classes.

15.2.1 Self

There is one difference between the functions and methods in classes. The methods have a varible **self** assigned to them. self variable refers to the object itself. In short we can say self is a placeholder for the object.

Note, we can give any name to the for this parameter, but its recommended to strongly stick to variable name **self**.

Lets now create a class with some methods and try to understand the concept of self as a whole:

```
In [11]: class newclass:
 ....:     def coursename(self,name):
 ....:         self.name = name
 ....:     def displayname(self):
 ....:         return self.name
 ....:     def printcourse(self):
 ....:         print "course name is %s" %self.name
 ....:
```

Lets now try to create couple of objects and try to call them using the methods we created:

```
In [12]: first_course = newclass()

In [13]: second_course = newclass()

In [14]:

In [14]: first_course.coursename('python')

In [15]: second_course.coursename('perl')

In [16]: first_course.displayname()
Out[16]: 'python'

In [17]: first_course.printcourse()
course name is python

In [18]:

In [18]: second_course.displayname()
Out[18]: 'perl'
```

```
In [19]: second_course.printcourse()
course name is perl
```

15.3 Method __init_ / constructs

There are few method name in python which have some significance. Lets discuss about the **__init__** now. Its a special method in a class.Which we dont have to call the moment we create our object. Lets take some quick example:

```
In [12]: class new:
 ....:     def __init__(self):
 ....:         print 'Hello this constructs'
 ....:         self.variable = 'hello'
 ....:

In [13]: Pobject = new()
Hello this constructs

In [14]: Pobject.variable
Out[14]: 'hello'
```

If you have noticed in the above example, the moment i created the object the attributes got called automatically.

Lets consider one more example:

```
1 #!/usr/bin/python
 2
 3 class shape:
 4   def __init__(self,x,y):
 5     self.x = x
 6     self.y = y
 7   description = "This shape has not been described yet"
 8   author = "Nobody has claimed to make this shape yet"
 9
10   def area(self):
11     return self.x * self.y
12   def perimeter(self):
13     return 2 * self.x + 2 * self.y
14   def describe(self,text):
15     self.description = text
16   def authorname(self,text):
17     self.author = text
18   def scalesize(self,scale):
19     self.x = self.x * scale
20     self.x = self.y * scale
21
22 rectangle = shape(100,45)
23
24 print rectangle.area()
25 print rectangle.perimeter()
26 print rectangle.describe(" wide rectangle more than twice as wide as tall")
27 print rectangle.scalesize(0.5)
28 print rectangle.area()
```

Output:

```
tcloud-Vostro-2520 python-examples # python class1.py
4500
```

```
290
None
None
1012.5
```

15.4 Inheritance

One of the benefits of the object oriented programming is the reuse of the code.Inheritence is one of the ways by which we can achieve this. Inheritance can be imargined as implementing a parentclass and childclass relationship.

So lets take a quick example:

```
In [36]: class Parentclass:
 ....:     def coursename(self,name):
 ....:         self.name = name
 ....:         return self.name
 ....:

In [24]: class Childclass(Parentclass):
 ....:     pass


In [37]: P_Obj1 = Parentclass()

In [38]: P_Obj1.coursename('python')
Out[38]: 'python'

In [42]: C_Obj1 = Childclass()

In [43]: C_Obj1.coursename('python')
Out[43]: 'python'
```

Lets take one quick example.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific charac- teristics such as salary, courses and leaves for teachers and, marks and fees for students.

You can create two independent classes for each type and process them but adding a new common char- acteristic would mean adding to both of these independent classes. This quickly becomes unwieldy. A better way would be to create a common class called SchoolMember and then have the teacher and student classes inherit from this class i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in SchoolMember, this is automatically reflected in the subtypes as well. For example, you can add a new ID card field for both teachers and students by simply adding it to the SchoolMember class. However, changes in the sub- types do not affect other subtypes. Another advantage is that if you can refer to a teacher or student object as a SchoolMember object which could be useful in some situations such as counting of the number of school members. This is called polymorphism where a sub-type can be substituted in any situation where a parent type is expected i.e. the object can be treated as an instance of the parent class.

Also observe that we reuse the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.

The SchoolMember class in this situation is known as the base class or the superclass. The Teacher and Student classes are called the derived classes or subclasses.

```
1 #!/usr/bin/python
2
3 class SchoolMember:
4   ''' Represents any school member'''
5   def __init__(self,name,age):
6     self.name = name
7     self.age = age
8     print '(Initilazition School Members: %s)' %self.name
9
10   def tell(self):
11     '''Tell my detals'''
12     print 'Name:"%s" Age:"%s" ' %(self.name,self.age)
13
14 class Teacher(SchoolMember):
15   '''Represents a teacher '''
16   def __init__(self,name,age,salary):
17     SchoolMember.__init__(self,name,age)
18     self.salary = salary
19     print '(Initilazition Teacher: %s)' % self.name
20
21   def tell(self):
22     SchoolMember.tell(self)
23     print 'Salary: "%d"' %self.salary
24
25 class Student(SchoolMember):
26    ''' Represents a student.'''
27    def __init__(self,name,age,marks):
28     SchoolMember.__init__(self,name,age)
29     self.marks = marks
30     print '(Initilization Student: %s)' %self.name
31
32    def tell(self):
33     SchoolMember.tell(self)
34     print 'Marks: "%d"' % self.marks
35
36
37 t = Teacher('Mrs. bhagyalakshmi',40,30000)
38 s = Student('dinesh',22,75)
39
40 print
41
42 members = [t,s]
43 for  member in members:
44   member.tell()
45
```

Output:

```
(Initilazition School Members: Mrs. bhagyalakshmi)
(Initilazition Teacher: Mrs. bhagyalakshmi)
(Initilazition School Members: dinesh)
(Initilization Student: dinesh)

Name:"Mrs. bhagyalakshmi" Age:"40"
Salary: "30000"
Name:"dinesh" Age:"22"
Marks: "75"
```

Explanation:

To use inheritance, we specify the base class names in a tuple following the class name in the class definition. Next, we observe that the __init__ method of the base class is explicitly called using the self variable so that we can initialize the base class part of the object. This is very important to re- member - Python does not automatically call the constructor of the base class, you have to explicitly call it yourself.

We also observe that we can call methods of the base class by prefixing the class name to the method call and then pass in the self variable along with any arguments. Notice that we can treat instances of Teacher or Student as just instances of the SchoolMember when we use the tell method of the SchoolMember class.

Also, observe that the tell method of the subtype is called and not the tell method of the School- Member class. One way to understand this is that Python always starts looking for methods in the type, which in this case it does. If it could not find the method, it starts looking at the methods belonging to its base classes one by one in the order they are specified in the tuple in the class definition.

15.5 OverWrite Variables on a subclass

In previous topics we learned about how to inherit the attributes of the parent in the child class. But sometimes we might want to just inherit only few of the features from the parent class and not all the attributes. Lets take a quick example to understand the same:

```
In [1]: class Parentclasses:
 ...:     course1 = 'python'
 ...:     course2 = 'perl'
 ...:

In [2]: class Childclasses(Parentclasses):
 ...:     course2 = 'Linux'
 ...:

In [3]: POB = Parentclasses()

In [4]: POB.course1
Out[4]: 'python'

In [5]: POB.course2
Out[5]: 'perl'

In [6]: COB = Childclasses()

In [7]: COB.course1
Out[7]: 'python'

In [8]: COB.course2
Out[8]: 'Linux'
```

If you notice in the above example, we have overwritten the course2 field of the Parentclasses with the course2 field of the child class.

15.6 Subclass with Multiple Parent classes

Sometimes we need to create subclass with multiple parent classes. In the below example we will try to understand how to achieve the same

```
In [16]: class coursepython:
 ....:     chp1 ="Introduction"
 ....:     chp2 ="Help"
 ....:

In [23]: class coursePadv:
 ....:     chp4="numpy"
 ....:     chp5="skipy"
 ....:

In [18]: class childPython(coursepython,coursePadv):
 ....:     chp3="dictionaries"
 ....:

In [25]: objc = childPython()

In [26]: objc.chp1
Out[26]: 'Introduction'

In [27]: objc.chp2
Out[27]: 'Help'

In [28]: objc.chp3
Out[28]: 'dictionaries'

In [29]: objc.chp4
Out[29]: 'numpy'

In [30]: objc.chp5
Out[30]: 'skipy'
```

In the above example of you notice childPython has inherited fields from multiple parent classes.

Summary:

- The ability to group similar functions and variable together is called encapsulation.
- A variable inside a class is known as Attribute.
- A function inside a class is know as method.
- A class is a same category of things as variables,lists,dictionaries etc. That is,they are objects.
- A class is known as data structure - It holds data,and methods to process the data.

15.7 Examples:

15.7.1 Example 1

```
#!/usr/bin/env python
class person:
  def print_name(self):     # name of argument is called self
    print self.name

# construing a object based on class.
p = person()

# name is property of the object called name.
```

```
p.name = 'Gautam'
p.print_name()
person.print_name(p)

print type(person)  # class object.
print type(p)       # Class instance.
```

15.7.2 Example 2:

```
santosh-PI945GCM oop # cat second.py
#!/usr/bin/env python
def __init__(self,name):
 self.name = name

def print_name(self):
 print self.name

p= person('Santosh')
p.print_name()
```

15.8 Exercises

1 .Define a class which has at least two methods: getString: to get a string from console input printString: to print the string in upper case. Also please include simple test function to test the class methods.

Hints: Use __init__ method to construct some parameters

CHAPTER

SIXTEEN

UNIT 16: DEBUGGING IN PYTHON

16.1 Introduction

Pdb is an interactive shell to debug the python code.

For this section of debugging we will use a small program called buggy.py.

```
def divide_one_by(division):
  return 1/division
if __name__ == '__main__':
  divide_one_by(0)
```

16.2 Various modes to get to pdb

- script mode
- Postmorted mode
- run mode
- Trace mode

16.2.1 Script Mode

We can use the script mode to step through the entire script in pdb.

```
python -m pdb buggy.py
> /python-examples/debug/buggy.py(3)<module>()
-> def divide_one_by(divisor):
(Pdb)
```

16.2.2 Enhanced script Mode

This mode works in python version 3.2 and other.

```
python -m pdb -c continue buggy.py
```

16.2.3 Post Mortem Mode

Used after a uncaught exception has been raised.

```
python -i buggy.py
Traceback (most recent call last):
File "buggy.py", line 7, in <module>
  divide_one_by(0)
File "buggy.py", line 4, in divide_one_by
  return 1/divisor
ZeroDivisionError: integer division or modulo by zero
>>> import pdb
>>> pdb.pm()
> /python-examples/debug/buggy.py(4)divide_one_by()
-> return 1/divisor
(Pdb)
```

16.2.4 Run mode

Used to execute some expression under pdb control. uses current local and global to interprete the expression.

```
import buggy,pbd:pbd.run('buggy.divide_one_by(0))
```

16.2.5 Trace Mode

```
pbd.set_trace()
```

- Convenient to stick to development and test code.
- Just another python expression so you can conditionalize in expression.
- Better use this mode always.

```
def divide_one_by(divisor):
  import pdb:pdb.set_trace()
  return 1/divisor
```

16.3 Getting started

16.3.1 How to get help

Once inside the `pdb` command prompt, type of `h` or `help` to get the help.

```
(Pdb) h

Documented commands (type help <topic>):
========================================
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
```

16.3.2 How to quit

we can quit out of the pdb commnad prompt by `quit()` or `q` key mode.

16.4 Examine

16.4.1 List down the code

l or *list* will list 11 lines of the code.

```
python-examples # python -m pdb inherit.py
> /python-examples/inherit.py(3)<module>()
-> class SchoolMember:
(Pdb) list
1   #!/usr/bin/python
2
3  -> class SchoolMember:
4     ''' Represents any school member'''
5     def __init__(self,name,age):
6       self.name = name
7       self.age = age
8       print '(Initilazition School Members: %s)' %self.name
9
10     def tell(self):
11       '''Tell my detals'''
(Pdb)
```

List from lines 1 to 13

```
(Pdb) l 1,13
1   #!/usr/bin/python
2
3  -> class SchoolMember:
4     ''' Represents any school member'''
5     def __init__(self,name,age):
6       self.name = name
7       self.age = age
8       print '(Initilazition School Members: %s)' %self.name
9
10     def tell(self):
11       '''Tell my detals'''
12       print 'Name:"%s" Age:"%s" ' %(self.name,self.age)
13
```

16.4.2 where are we

w or *where* prints the current position of the code.

```
(Pdb) where
/usr/lib/python2.7/bdb.py(400)run()
-> exec cmd in globals, locals
<string>(1)<module>()
> /python-examples/inherit.py(3)<module>()
-> class SchoolMember:
(Pdb)
```

16.4.3 How to repeat the last command

Please hit on enter to repeat the last command. Note is should not be a list command.

16.4.4 How to execute the current statement

`n` or `nexg` is the current statement and step over.

```
(Pdb) n
> /python-examples/inherit.py(14)<module>()
-> class Teacher(SchoolMember):
(Pdb) n
> /python-examples/inherit.py(25)<module>()
-> class Student(SchoolMember):
(Pdb) n
> /python-examples/inherit.py(37)<module>()
-> t = Teacher('Mrs. bhagyalakshmi',40,30000)
```

16.4.5 How to step into the function

`s` or `step` is used to execute and step into the function.

```
-> def a_even(a):
(Pdb) s
> /python-examples/debug/latest.py(4)a_even()
-> if a % 2 == 0:
(Pdb) where
/usr/lib/python2.7/bdb.py(400)run()
-> exec cmd in globals, locals
<string>(1)<module>()
/python-examples/debug/latest.py(10)<module>()
-> a_even(value)
> /python-examples/debug/latest.py(4)a_even()
-> if a % 2 == 0:
(Pdb) l
1   #!/usr/bin/python
2
3   def a_even(a):
4  ->   if a % 2 == 0:
5       return "even"
6     else:
7       return "odd"
8
```

```
9   value = int(raw_input("please enter the number:"))
10   a_even(value)
[EOF]
```

16.4.6 How to restart the debugging again

`restart` or `r` is used to restart the program.

```
(Pdb) restart
Restarting latest.py with arguments:

> /python-examples/debug/latest.py(3)<module>()
-> def a_even(a):
(Pdb) h
```

16.4.7 How to print variables

use `print` or `p` or `pp` to print the variable within the program.

```
debug # python latest.py
please enter the number:22
> /python-examples/debug/latest.py(11)<module>()
-> a_even(value)
(Pdb) l
6     else:
7       return "odd"
8
9   value = raw_input("please enter the number:")
10   import pdb;pdb.set_trace()
11  -> a_even(value)
[EOF]
(Pdb) p value
'22'
(Pdb) pp value
'22'
(Pdb) print value
22
```

16.4.8 How to step into the function

use `s` or `step` to step into the function.

```
debug # python latest.py
 please enter the number:21
 > /python-examples/debug/latest.py(11)<module>()
 -> a_even(value)
 (Pdb) l
 6     else:
 7       return "even"
 8
 9   value = int(raw_input("please enter the number:"))
 10   import pdb;pdb.set_trace()
 11  -> a_even(value)
 [EOF]
```

```
(Pdb) s
--Call--
> /python-examples/debug/latest.py(3)a_even()
-> def a_even(a):
(Pdb) l
1   #!/usr/bin/python
2
3  -> def a_even(a):
4     if a % 2 == 0:
5       return "odd"
6     else:
7       return "even"
8
9   value = int(raw_input("please enter the number:"))
10  import pdb;pdb.set_trace()
11  a_even(value)
(Pdb) s
> /python-examples/debug/latest.py(4)a_even()
-> if a % 2 == 0:
(Pdb) l
1   #!/usr/bin/python
2
3   def a_even(a):
4  ->   if a % 2 == 0:
5       return "odd"
6     else:
7       return "even"
8
9   value = int(raw_input("please enter the number:"))
10  import pdb;pdb.set_trace()
11  a_even(value)
```

16.4.9 Continue executing until the current function returns

Use `r` or `return` to execute the current function till it returns a value.

```
Pdb) r
--Return--
> /python-examples/debug/latest.py(7)a_even()->'even'
-> return "even"
```

16.4.10 Turning off the `pdb` prompt

You probably noticed that the "q" command got you out of pdb in a very crude way — basically, by crashing the program.

If you wish simply to stop debugging, but to let the program continue running, then you want to use the `c` (for "continue") command at the (Pdb) prompt. This will cause your program to continue running normally, without pausing for debugging. It may run to completion. Or, if the pdb.set_trace() statement was inside a loop, you may encounter it again, and the (Pdb) debugging prompt will appear once more.

16.5 Command Language

- Readlines is used in the debugging so arrow keys and other readlines key bindings work.

- Aliases can be used.
- Full command name or it shortened version can be used at (pdb) prompt.
- Exceptions generated when a pdb expression generates an exceptions dont impact the current program state.

CHAPTER

SEVENTEEN

UNIT 17: LOGGING

17.1 Introduction

Logging in one of the important modules. This module helps us in tracking the log message in our own log files. Also our application logs can include our own messages with other third party modules. In this section we learn about how to integrate the logger modules with our syslog messages.Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

17.1.1 When to use logging

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

The default level is WARNING, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.

```
import logging

In [13]: logging.info('This is just information')

In [14]: logging.warn('This is just warning')
WARNING:root:This is just warning
```

printed out on the console. The INFO message doesn't appear because the default level is WARNING. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. 'WARNING:root:This is just warning'. Don't worry about the 'root' part for now: it will be explained later. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

17.2 Logging to a File

We will go around with a basic example of logging to the file.

```
In [5]: import logging

In [6]: import logging as l

In [7]: l.basicConfig(filename='/tmp/log.txt',level=l.DEBUG)

In [8]: l.debug('This message should go to logfile')

In [9]: l.info('This is just info')

In [10]: l.warning('This is just warning')

In [11]: ! cat /tmp/log.txt
DEBUG:root:This message should go to logfile
INFO:root:This is just info
WARNING:root:This is just warning
```

17.3 Logging from multiple modules

If your program consists of multiple modules, here's an example of how you could organize logging in it:

```
# myapp.py
import logging
import mylib
```

```
def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()

# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

which is hopefully what you were expecting to see. You can generalize this to multiple modules, using the pattern in *mylib.py*. Note that for this simple usage pattern, you won't know, by looking in the log file, *where* in your application your messages came from, apart from looking at the event description.

17.4 Logging variable data

To log variable data, use a format string for the event description message and append the variable data as arguments. For example:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

```
WARNING:root:Look before you leap!
```

As you can see, merging of variable data into the event description message uses the old, %-style of string formatting.

17.5 Changing the format of displayed messages

To change the format which is used to display messages, you need to specify the format you want to use:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

which would print:

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

Notice that the 'root' which appeared in earlier examples has disappeared. For a full set of things that can appear in format strings.

17.6 Syslog Integration

- First and formost we need to import the logger modules.

```
import logging
```

- Defination and instantion of logger messages.

```
logger = logging.getLogger()
```

- Defination of handler

```
han = logging.FileHandler('log1.log')
```

- Defination of the formatted string.

```
logging.Formatter('%(asctime)s %(levelname)s %(message)s')
```

- Add Handler to Logger

```
han.setFormatter(format)
logger.addHandler(han)
```

- Invocation of logger

```
logmessage = "testing logger in Python"
logger.error(logmessage)
```

Lets try to all of these together:

```
i  1 #!/usr/bin/python
2 import logging
3
4
5 logger = logging.getLogger('My Application')
6 han = logging.FileHandler('log1.log')
7 format = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
8 han.setFormatter(format)
9 logger.addHandler(han)
10
11 logmessage = "testing the log messages"
12
13 logger.error(logmessage)
```

If you notice the file `log1.log` is getting created in the same location as the script. Run the script:

```
2014-03-23 11:55:28,709 ERROR testing the log messages
2014-03-23 11:55:42,438 ERROR testing the log messages
```

17.7 How it works

For setting up the Logger message we need to follow up the following model.

1. Model

2. Logger
3. handler
4. Destination (File,syslog,SMTP,UDP/Socket,etc)

17.8 Integration with Syslog or rsyslog

Rsyslog is an open source software utility used on UNIX and Unix-like computer systems for forwarding log messages in an IP network. It implements the basic syslog protocol, extends it with content-based filtering, rich filtering capabilities, flexible configuration options and adds important features such as using TCP for transport.It will be very helpful for Linux administrators to view and troubleshoot errors if something went wrong.

Before RHEL5 we had syslog and from RHEL6 we have rsyslog for the for logging messages.

Lets now try to work through the syslog interaction.

- First import the necessary modules

```
import logging
import logging.handlers
```

- Defination and instantation of the logger objects

```
logger = logging.getLogger()
```

- Defination of Handlers

```
han = logging.FileHandler('log1.log')
han1 = logging.handlers.syslogHandlers()
```

Note `han1` is the handler for the syslog.

- Defination of the formate string

```
format = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
han.setFormatter(format)
han1.setFormatter(format)
```

- Adding handlers to logger

```
logger.addHandler(han)
logger.addHandler(han1)
```

- Invocation of logger

```
logmessage="Testing logging in python"
logger.error(logmessage)
```

17.8.1 Custom Levels

Defining your own levels is possible, but should not be necessary, as the existing levels have been chosen on the basis of practical experience. However, if you are convinced that you need custom levels, great care should be exercised when doing this, and it is possibly *a very bad idea to define custom levels if you are developing a library*. That's because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean different things for different libraries.

17.9 Useful Handlers

In addition to the base `Handler` class, many useful subclasses are provided:

1. `StreamHandler` instances send messages to streams (file-like objects).
2. `FileHandler` instances send messages to disk files.
3. `BaseRotatingHandler` is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use `RotatingFileHandler` or `TimedRotatingFileHandler`.
4. `RotatingFileHandler` instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. `TimedRotatingFileHandler` instances send messages to disk files, rotating the log file at certain timed intervals.
6. `SocketHandler` instances send messages to TCP/IP sockets.
7. `DatagramHandler` instances send messages to UDP sockets.
8. `SMTPHandler` instances send messages to a designated email address.
9. `SysLogHandler` instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. `NTEventLogHandler` instances send messages to a Windows NT/2000/XP event log.
11. `MemoryHandler` instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. `HTTPHandler` instances send messages to an HTTP server using either `GET` or `POST` semantics.
13. `WatchedFileHandler` instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
14. `NullHandler` instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the 'No handlers could be found for logger XXX' message which can be displayed if the library user has not configured logging. See *Configuring Logging for a Library* for more information.

New in version 2.7: The `NullHandler` class.

The `NullHandler`, `StreamHandler` and `FileHandler` classes are defined in the core logging package. The other handlers are defined in a sub- module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Logged messages are formatted for presentation through instances of the `Formatter` class. They are initialized with a format string suitable for use with the % operator and a dictionary.

For formatting multiple messages in a batch, instances of `BufferingFormatter` can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough, instances of `Filter` can be added to both `Logger` and `Handler` instances (through their `addFilter()` method). Before deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic `Filter` functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.

CHAPTER

EIGHTEEN

UNIT 18: SOCKET PROGRAMMING

18.1 Introduction

To summarise the basics, sockets are the fundamental "things" behind any kind of network communications done by your computer. For example when you type www.google.com in your web browser, it opens a socket and connects to google.com to fetch the page and show it to you. Same with any chat client like gtalk or skype. Any network communication goes through a socket.

18.2 UDP and TCP

Layered on top of IP

18.2.1 UDP - Packets

1. datagrams, SOCK_DGRAM
2. unordered
3. unreliable
4. unconnected

18.2.2 TCP - stream

1. steams of bytes,SOCK_STREAM
2. ordered
3. reliable
4. connected endpoints

*.recvfrom(),.recv(),.sendto(),.send() '' and co behave differently on socket types.

The concept of "connections" apply to SOCK_STREAM/TCP type of sockets. Connection means a reliable "stream" of data such that there can be multiple such streams each having communication of its own. Think of this as a pipe which is not interfered by data from other pipes. Another important property of stream connections is that packets have an "order" or "sequence".

Other sockets like UDP , ICMP , ARP dont have a concept of "connection". These are non-connection based communication. Which means you keep sending or receiving packets from anybody and everybody.

18.3 Working with Sockets

Lets get started with creating a socket.

```
In [9]: import socket

In [10]: s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

In [11]: print 'socket created'
socket created
```

Few things to note here:

- Function socket.socket creates a socket and returns a socket descriptor which can be used in other socket related functions.
- Address Family : AF_INET (This is for IP version of 4 or IPV4)
- Type : SOCK_STREAM (This means connection oriented TCP protocol)

Now lets try to handle if a socket cannot be created lets try to create a exception for the same.

```
In [15]: import socket,sys

In [17]: try:
   ....:     s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
   ....: except socket.error,error:
   ....:     print 'Failed to create a socket. Error Code: ' + str(error[0]) + ' , Error message :' +
   ....:     sys.exit()
   ....:
```

So we have not created a socket lets try some server using this socket.

18.3.1 How to connect to a server

In this section we will learn about connecting to the server socket.

```
1 #!/usr/bin/python
2 import socket,sys
3
4 # creating a socket
5
6 try:
7   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8 except socket.error,error:
9   print 'Failed to create a socket. Error Code: ' + str(error[0]) + ' , Error message :' + str(erro
10   sys.exit()
11
12 print 'socket created'
13
14 host = 'blog.tuxfux.com'
15
16 try:
17   remote_ip = socket.gethostbyname( host )
18 except:
19   print 'Hostname could not be resolved.exiting'
20   sys.exit()
21
```

```
22 print 'Ip address of ' + host + ' is ' + remote_ip
23
```

In the example above we have connected to a host `blog.tuxfux.com` using the **socket.gethostbyname**() method.

18.3.2 Connecting on a port

Now we have got the ip of the server, we have to try connecting to the server with a port number. Lets try to connect to the host using the port.

```
1 #!/usr/bin/python
2 import socket,sys
3
4 # creating a socket
5
6 try:
7   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8 except socket.error,error:
9   print 'Failed to create a socket. Error Code: ' + str(error[0]) + ' , Error message :' + str(erro
10   sys.exit()
11
12 print 'socket created'
13
14 host = 'blog.tuxfux.com'
15 port = 80
16
17 try:
18   remote_ip = socket.gethostbyname( host )
19 except:
20   print 'Hostname could not be resolved.exiting'
21   sys.exit()
22
23 print 'Ip address of ' + host + ' is ' + remote_ip
24
25 print "connecting to the server \n"
26
27 s.connect((remote_ip,port))
28
29 print 'Socket Connected to ' + host + ' on ip ' + remote_ip
```

It creates a socket and then connects. Try connecting to a port different from port 80 and you should not be able to connect which indicates that the port is not open for connection. This logic can be used to build a port scanner.

```
1 #!/usr/bin/python
2 import socket,sys
3
4 # taking port from first argument
5
6 if len(sys.argv) != 2:
7   print '''
8           ./sys.argv[0] <port no>
9                   '''
10   sys.exit()
11 else:
12   port = int(sys.argv[1])
13
14
```

```
15
16 # creating a socket
17
18 try:
19   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
20 except socket.error,error:
21   print 'Failed to create a socket. Error Code: ' + str(error[0]) + ' , Error message :' + str(er
22   sys.exit()
23
24 print 'socket created'
25
26 host = 'localhost'
27
28 try:
29   remote_ip = socket.gethostbyname( host )
30 except:
31   print 'Hostname could not be resolved.exiting'
32   sys.exit()
33
34 print 'Ip address of ' + host + ' is ' + remote_ip
35
36 print "connecting to the server \n"
37
38 s.connect((remote_ip,port))
39
40 print 'Socket Connected to ' + host + ' on ip ' + remote_ip + ' on port ' + str(port)
```

OK, so we are now connected. Lets do the next thing , sending some data to the remote server.

18.3.3 Sending data

In this section we will learn about sending the data using the `sendall` function.

```
31  # sending a data outside
32  message = raw_input("enter the message you want to pass on: \n")
33
34  try:
35    s.sendall(message)
36  except socket.error:
37    print 'send failed'
38    sys.exit()
39
40  print 'Message sending succesfully'
```

If you notice above, we are using `s.sendall(message)` for sending the message.

Let see our modification we have done in our current program.

```
1 #!/usr/bin/python
2 import socket,sys
3
4 # creating a socket
5
6 try:
7   s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8 except socket.error,error:
9   print 'Failed to create a socket. Error Code: ' + str(error[0]) + ' , Error message :' + str(erro
10   sys.exit()
```

```
11
12 print 'socket created'
13
14 host = 'blog.tuxfux.com'
15 port = 80
16
17 try:
18   remote_ip = socket.gethostbyname( host )
19 except:
20   print 'Hostname could not be resolved.exiting'
21   sys.exit()
22
23 print 'Ip address of ' + host + ' is ' + remote_ip
24
25 print "connecting to the server \n"
26
27 s.connect((remote_ip,port))
28
29 print 'Socket Connected to ' + host + ' on ip ' + remote_ip
30
31 # sending a data outside
32 message = raw_input("enter the message you want to pass on: \n")
33
34 try:
35   s.sendall(message)
36 except socket.error:
37   print 'send failed'
38   sys.exit()
39
40 print 'Message sending succesfully'
```

18.3.4 Receiving data

We can now use the `recv` function to receive data on a socket. Let try to append this function to our other example.

```
43 # receiving the data from system.
44 reply = s.recv(4096)
45
46 print reply
```

18.3.5 How to close the socket

we can close the socket using the `close()` function.

```
s.close()
```

18.3.6 Lets Revise

So what we convered till now.

1. Creating a socket.

```
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

2. Connecting to a server.

```
s.connect((remote_ip,port))
```

3. Sending data.

```
s.sendall(message)
```

4. Receiving data.

```
reply = s.recv(4096)
```

18.4 Programming a socket servers

Servers basically do the following:

1. Opening a socket.
2. Bind to a address(and port).
3. Listen for incoming connections.
4. Accept connections.
5. Read/send.

18.4.1 Bind a socket

we can use the `bind` to bind a socket to a particular address and port.

```
In [46]: import socket

In [47]: s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

In [48]: host = 'localhost'

In [49]: s.bind((host,9999))
```

If you notice in the above example we have opened a socket and binded the address to the port.

w that bind is done, its time to make the socket listen to connections. We bind a socket to a particular IP address and a certain port number. By doing this we ensure that all incoming data which is directed towards this port number is received by this application.

18.4.2 Listen to incoming connections

After binding a socket to a port the next thing we need to do is listen for connections. For this we need to put the socket in listening mode. Function socket_listen is used to put the socket in listening mode. Just add the following line after bind.

```
In [59]: s.listen(1)

In [60]: s.listen?
Type:       instancemethod
String Form:<bound method _socketobject.listen of <socket._socketobject object at 0x3357b40>>
Docstring:
listen(backlog)
```

```
Enable a server to accept connections.  The backlog argument must be at
least 0 (if it is lower, it is set to 0); it specifies the number of
unaccepted connections that the system will allow before refusing new
connections.
```

If you do a network scan now you will find we will get a port for that.

```
nmap localhost

Starting Nmap 6.00 ( http://nmap.org ) at 2014-04-05 13:23 IST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00015s latency).
Not shown: 993 closed ports
PORT     STATE SERVICE
21/tcp   open  ftp
80/tcp   open  http
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
631/tcp  open  ipp
3306/tcp open  mysql
9999/tcp open  abyss

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

The parameter of the function listen is called backlog. It controls the number of incoming connections that are kept "waiting" if the program is already busy. So by specifying 1, it means that if 1 connections are already waiting to be processed, then the 2th connection request shall be rejected. This will be more clear after checking socket_accept.

Now comes the main part of accepting new connections.

18.4.3 Accept Connections

Function `socket_accept` is used for accepting the connections.

```
1 #!/usr/bin/python
2
3 import sys,socket
4
5 HOST = 'localhost'
6 PORT = 9999
7 s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8
9 try:
10   s.bind((HOST, PORT))
11 except socket.error, msg:
12   print 'Bind Failed. Error code : ' + str(msg[0]) + ' message ' + msg[1]
13   sys.exit()
14
15 print 'Socket binding completed'
16
17 s.listen(10)
18 print 'socket now listening'
19
20 conn,addr = s.accept()
21
22 print "connected with" +  addr[0] + ':' + str(addr[1])
23
```

Once we run this program we should get the following output:

```
python server.py
Socket binding completed
socket now listening
```

Now if we try to connect from the client like `telent` we will see that the connection is off.

```
telnet localhost 9999
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
```

Output from the program `server.py` is as follows.

```
python server.py
Socket binding completed
socket now listening
connected with127.0.0.1:40095
```

We accepted an incoming connection but closed it immediately. This was not very productive. There are lots of things that can be done after an incoming connection is established. Afterall the connection was established for the purpose of communication. So lets reply to the client.

Function `sendall` can be used to send something to the socket of the incoming connection and the client should see it.

```
1 #!/usr/bin/python
2
3 import sys,socket
4
5 HOST = 'localhost'
6 PORT = 9999
7 s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8
9 try:
10   s.bind((HOST, PORT))
11 except socket.error, msg:
12   print 'Bind Failed. Error code : ' + str(msg[0]) + ' message ' + msg[1]
13   sys.exit()
14
15 print 'Socket binding completed'
16
17 s.listen(10)
18 print 'socket now listening'
19
20 conn,addr = s.accept()
21
22 print "connected with" +  addr[0] + ':' + str(addr[1])
23
24 data = conn.recv(1024)
25 conn.sendall(data)
26
27 conn.close()
28 s.close()
```

So the client(telnet) received a reply from server.

We can see that the connection is closed immediately after that simply because the server program ends after accepting and sending reply. A server like www.google.com is always up to accept incoming connections.

It means that a server is supposed to be running all the time. Afterall its a server meant to serve. So we need to keep our server RUNNING non-stop. The simplest way to do this is to put the accept in a loop so that it can receive incoming connections all the time.

18.4.4 Live server

```
1 #!/usr/bin/python
2
3 import sys,socket
4
5 HOST = 'localhost'
6 PORT = 9999
7 s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
8
9 try:
10   s.bind((HOST, PORT))
11 except socket.error, msg:
12   print 'Bind Failed. Error code : ' + str(msg[0]) + ' message ' + msg[1]
13   sys.exit()
14
15 print 'Socket binding completed'
16
17 s.listen(10)
18 print 'socket now listening'
19
20 while 1:
21   conn,addr = s.accept()
22   print "connected with" +  addr[0] + ':' + str(addr[1])
23
24   data = conn.recv(1024)
25   reply = 'OK ... ' + data
26   if not data:
27     break
28
29   conn.sendall(reply)
30
31 conn.close()
32 s.close()
```

In the above code we have used a while loop to keep on accepting the connections. Now lets try to make the connection using the some of the connection from multiple terminals.

```
telnet localhost 9999
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
OK ... hello
^]

telnet> quit
Connection closed.
```

So if you notice, we have done created multiple connection to the port from multiple locations.

```
./live_server.py
Socket binding completed
socket now listening
```

```
connected with127.0.0.1:41157
connected with127.0.0.1:41158
connected with127.0.0.1:41159
```

18.4.5 Handling Connections

To handle every connection we need a separate handling code to run along with the main server accepting connections. One way to achieve this is using threads. The main server program accepts a connection and creates a new thread to handle communication for the connection, and then the server goes back to accept more connections.

We shall now use threads to create handlers for each connection the server accepts.

```
1 #!/usr/bin/python
2
3 import sys,socket
4 from thread import *
5
6 HOST = 'localhost'
7 PORT = 9999
8 s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
9
10 try:
11   s.bind((HOST, PORT))
12 except socket.error, msg:
13   print 'Bind Failed. Error code : ' + str(msg[0]) + ' message ' + msg[1]
14   sys.exit()
15
16 print 'Socket binding completed'
17
18 s.listen(10)
19 print 'socket now listening'
20
21 def clientthread(conn):
22   conn.send('welcome to the server. Type something and hit enter \n')
23
24   while True:
25     data = conn.recv(1024)
26     reply = 'OK .. ' + data
27     if not data:
28       break
29     conn.sendall(reply)
30
31   conn.close()
32
33
34 while 1:
35   conn,addr = s.accept()
36   print "connected with" +  addr[0] + ':' + str(addr[1])
37   start_new_thread(clientthread,(conn,))
38
39 s.close()
```

CHAPTER

NINETEEN

UNIT 19: CGI PROGRAMMING

19.1 Introduction

CGI stands for `common gateway interface`. CGI allows the web server to interact with external programs. These programs can range in purpose and scope, but primarily they help to add dynamic content to a website. Most often these applications are Python programs with the extension .py, but CGI execution is not limited to Python.

19.2 Getting started with CGI

One of the stumbling blocks most people come up against with CGI is getting their Apache server to recognize the CGI directory and to allow for the execution of commands from within that directory.

Let's first talk about the directories. If you look in the /var/www (the document root of Apache), you will find a sub-directory called cgi-bin. This is not where your Python programs and other various files will be placed. Within the /usr/lib/ directory, you will find another cgi-bin directory; it is the repository for your executables. If that directory does not exist, create it with the command: sudo mkdir /usr/lib/cgi-bin.

```
cgi-bin # pwd
/usr/lib/cgi-bin
```

19.3 Configuring apache

A directive must be created so Apache knows about CGI – where its directories are located and what it can do. In some Apache configurations, this is done within the httpd.conf file. Because this is Ubuntu, we are going to add the directive to the /etc/apache/sites-available/default.

Make sure you have these entries inside the file

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
  AllowOverride None
  Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
  Order allow,deny
  Allow from all
</Directory>
```

Lastly , don't forget to restart the apache server.

```
sudo service apache2 restart
```

19.4 Time to Test:

19.4.1 Example 1:

Lets try to put a small code in the /usr/lib/cgi-bin directory.

```
#!/usr/bin/python

print "Content-Type: text/html"
print

print "<h1><font face='arial' size='+2'>Hello world.. From CGI python </font></strong></h1>"
```

19.4.2 Example 2:

Lets create a basic `form` for accepting the values. The post method gets into action the moment we hit the `submit` button.

```
#!/usr/bin/python

print "Content-type: text/html"
print

print "<html>"
print "<title>Our first CGI Python Program</title>"
print "<body>"
print "<form action='action.py' method='post'>"
print "Name: <input type='text' name='name' size='25'><br>"
print "Title: <input type='text' name='title' size='30'><br>"
print "Email: <input type='text' name='email' size='30'><br>"
print "<input type='submit' value='submit'>"
print "</form>"
print "</body>"
print "</html>"
```

Action method `action.py`

```
#!/usr/bin/python

import cgi

print "Content-type: text/html"
print

form = cgi.FieldStorage()
name = form["name"].value
title = form["name"].value
email = form["email"].value

print "Thanks %s for submitting the email %s" %(name,email)
```

To learn more about the CGI : https://docs.python.org/2/library/cgi.html

CHAPTER

TWENTY

UNIT 20: DATABASE CONNECTIVITY

20.1 Introduction

Python supports multiple databases. One of the main reasons for including this topic is very simple. We might need to connect to databases for updating,querying and inserting of records. Database is one quick way of keeping the data persistent.

We have lots of databases where we have support for python:

- Mysql
- Oracle
- NOsql
- sqllite
- Mariadb

In this topic we will work around with the `mysql` database and try to understand how to play around with the database elements.

20.1.1 How to install mysql-server

I am working on a linux mint operating system and i tried installing the mysql-server database using the command given below.

```
sudo apt-get install mysql-server
```

20.1.2 How to know what modules to install

Since we are planning to work on a mysql database , lets try to verify the same.

```
sudo apt-cache search MySQLdb
```

Output

```
python-mysqldb - Python interface to MySQL
python-mysqldb-dbg - Python interface to MySQL (debug extension)
bibus - bibliographic database
eikazo - graphical frontend for SANE designed for mass-scanning
mysql-utilities - collection of scripts for managing MySQL servers
```

Lets install the modules for mysql.

```
sudo apt-get install python-mysqldb
```

Lets get started with some basics:

20.1.3 Creating a database

First lets create a datbase called `mydata` . For those of you who are new to the mysql database you can please go with the command below.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 42
Server version: 5.5.34-0ubuntu0.13.04.1 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Lets try running the database

```
mysql> create database mydata;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mydata             |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

If you notice in the above command , we have created a database `mydata`.

20.1.4 Creation of the user and giving grants

Now lets try to create a user and try to give it all the necessary grants.

```
mysql> create user testuser;
Query OK, 0 rows affected (0.00 sec)
```

We can see the mysql database if the user is created on not using the following commands

```
mysql> select user from mysql.user where user='testuser';
+----------+
| user     |
+----------+
```

```
| testuser |
+----------+
1 row in set (0.00 sec)
```

20.1.5 Granting access to the users

Now lets try to grant access to the user `testuser` and provide password `testuser` with ALL grant privileges on database `mydata` .

```
mysql> grant all on mydata.* to 'testuser'@'localhost' IDENTIFIED BY 'testuser';
Query OK, 0 rows affected (0.00 sec)
```

So, now we are set. Lets try to loging using the `testuser` credentials.

```
mysql -u testuser -p mydata
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 5.5.34-0ubuntu0.13.04.1 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select user();
+--------------------+
| user()             |
+--------------------+
| testuser@localhost |
+--------------------+
1 row in set (0.00 sec)

mysql> select schema();
+----------+
| schema() |
+----------+
| mydata   |
+----------+
1 row in set (0.00 sec)

mysql>
```

20.1.6 Connecting mysql and python

Lets first try to verify if we have the `MySQLdb` installed in our systems.

```
In [1]: import MySQLdb as mdb

In [2]: mdb.
mdb.BINARY                                mdb.NotSupportedError                     mdb.escape_
mdb.Binary                                mdb.OperationalError                      mdb.escape_
mdb.Connect                               mdb.ProgrammingError                      mdb.get_cli
```

```
mdb.Connection                    mdb.ROWID                         mdb.paramst
mdb.DATE                          mdb.STRING                        mdb.release
mdb.DATETIME                      mdb.TIME                          mdb.result
mdb.DBAPISet                      mdb.TIMESTAMP                     mdb.server_
mdb.DataError                     mdb.Time                          mdb.server_
mdb.DatabaseError                 mdb.TimeFromTicks                 mdb.string_
mdb.Date                          mdb.Timestamp                     mdb.test_DB
mdb.DateFromTicks                 mdb.TimestampFromTicks            mdb.test_DB
mdb.Error                         mdb.Warning                       mdb.test_DB
mdb.FIELD_TYPE                    mdb.apilevel                      mdb.test_DB
mdb.IntegrityError                mdb.connect                       mdb.thread_
mdb.InterfaceError                mdb.connection                    mdb.threads
mdb.InternalError                 mdb.constants                     mdb.times
mdb.MySQLError                    mdb.debug                         mdb.version
mdb.NULL                          mdb.escape
mdb.NUMBER                        mdb.escape_dict
```

Now lets try to make a connection to our `mydata` database.

```
In [2]: con = mdb.connect('localhost','testuser','testuser','mydata')
```

Next we have to create a `con.cursor()` .

```
In [4]: cur = con.cursor()
```

```
In [5]: con.cursor?
Type:       instancemethod
String Form:<bound method Connection.cursor of <_mysql.connection open to 'localhost' at 31b3ca0>>
File:       /usr/lib/python2.7/dist-packages/MySQLdb/connections.py
Definition: con.cursor(self, cursorclass=None)
Docstring:
Create a cursor on which queries may be performed. The
optional cursorclass parameter is used to create the
Cursor. By default, self.cursorclass=cursors.Cursor is
used.
```

Now lets try to understand who is the user logged into the database.

```
In [8]: cur.execute("select user()")
Out[8]: 1L
```

From the connection we get the cursor object . The cursor is used to traverse the records from the result set. We call the execute() method of the cursor and execute the SQL statement.

```
In [9]: username = cur.fetchone()
```

Here in this connection we are fetching only one record. So we are calling the fetchone() function.

Once everything is over , we can close the connection.

```
In [9]: username = cur.fetchone()
```

20.2 Some basic mysql operations:

In this section we will try to complete some basic mysql operation .

20.2.1 Creating and populating a table

First import the MySQLdb module.

```
In [7]: import MySQLdb as mdb
```

Lets try to connect to the database using the credentials.

`localhost` is the host to connect. `testuser` is the username. `testuser` is the password. `mydata` is the database.

```
In [8]: con = mdb.connect('localhost','testuser','testuser','mydata')
```

From the connection we get the cursor object.The cursor is used to traverse the records from the result set.

```
In [9]: cur = con.cursor()
```

We call the `execute()` method of the cursor and execute the SQL statements.

```
In [11]: cur.execute('create table students(rollno int,name varchar(25))')
Out[11]: 0L
```

If you now check in the database we will see the table `student` is created.

```
mysql> show tables;
+------------------+
| Tables_in_mydata |
+------------------+
| students         |
+------------------+
1 row in set (0.00 sec)

mysql> desc students;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| rollno | int(11)     | YES  |     | NULL    |       |
| name   | varchar(25) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

We can also use the `executemany` options to run multiple queries at a time.

```
In [10]: cur.executemany?
Type:       instancemethod
String Form:<bound method Cursor.executemany of <MySQLdb.cursors.Cursor object at 0x64c2a10>>
File:       /usr/lib/python2.7/dist-packages/MySQLdb/cursors.py
Definition: cur.executemany(self, query, args)
Docstring:
Execute a multi-row query.

query -- string, query to execute on server

args

  Sequence of sequences or mappings, parameters to use with
  query.

Returns long integer rows affected, if any.

This method improves performance on multiple-row INSERT and
```

```
REPLACE. Otherwise it is equivalent to looping over args with
execute().
```

Lets go with one of the examples of inserting multiple records into the table `students`.

```
In [14]: cur.executemany("insert into students(rollno,name) values (%s,%s)",[(1,'hari'),(2,'priya'),
Out[14]: 3L

In [15]: con.commit()
```

20.2.2 Retrieving data

Now since we have inserted data into the database , lets try to retrieve the records.

`cur.fetchall()`

```
In [2]: import MySQLdb as mdb
In [3]: con = mdb.connect('localhost','testuser','testuser','mydata')
In [4]: cur = con.cursor()
In [5]: cur.execute("select * from students")
Out[5]: 3L
In [6]: rows = cur.fetchall()
In [7]: for i in rows:
 ...:     print i
 ...:
(1L, 'hari')
(2L, 'priya')
(3L, 'vicky')
```

If you notice in the above example, `cur.fetchall()` function fetches all the rows.

`cur.fetchone()`

Using the `cur.fetchone()` method we can fetch the rows one by one.

```
In [2]: import MySQLdb as mdb
In [3]: con = mdb.connect('localhost','testuser','testuser','mydata')
In [4]: cur = con.cursor()
In [5]: cur.execute("select * from students")
Out[5]: 3L

In [12]: print cur.rowcount
3

In [13]: for i in range(cur.rowcount):
 ....:     row = cur.fetchone()
 ....:     print row[0],row[1]
 ....:
1 hari
2 priya
3 vicky
```

The `cur.rowcount` property gives the number of rows returned by the SQL statements.

20.2.3 Dictionary cursors

There are multiple cursor types in the MySQLdb module. The default cursor returns the data in tuple of tuples. When we use a dictionary cursor, the data is sent in a form of python dictionaries. This way we can refer to the data by the

column names.

```
In [14]: import MySQLdb as mdb

In [15]: con = mdb.connect('localhost','testuser','testuser','mydata')

In [16]: cur = con.cursor(mdb.cursors.DictCursor)

In [17]: cur.execute("select * from students")
Out[17]: 3L

In [18]: rows = cur.fetchall()

In [20]: print rows
({'name': 'hari', 'rollno': 1L}, {'name': 'priya', 'rollno': 2L}, {'name': 'vicky', 'rollno': 3L})

In [22]: for row in rows:
  print row['name'],row['rollno']
 ....:
hari 1
priya 2
vicky 3
```

20.2.4 Column headers

We will show , how to print column headers with the data from the database table.

```
In [4]: import MySQLdb as mdb

In [5]: con = mdb.connect('localhost','testuser','testuser','mydata')

In [7]: cur = con.cursor()
In [9]: cur.execute("select * from students")
Out[9]: 3L

In [10]: rows = cur.fetchall()

In [11]: desc = cur.description

In [12]: print "%s %3s" %(desc[0][0],desc[1][0])
rollno name

In [13]: for row in rows:
 ....:     print "%2s %3s" %row
 ....:
1 hari
2 priya
3 vicky
```

The `cur.description` attribute of the cursor returns information about each of the result columns of a query.

CHAPTER

TWENTYONE

UNIT : SO WHAT NOW ?

21.1 so what now ?

Thanks everyone for going on with these sessions. Python is all about learning daily and keeping on updating the skill set on a day to day basis. I am here by providing few of the sites and links which will help you gain sufficient knowledge.

My Recommended groups:

http://in.groups.yahoo.com/group/TUXFUX/

https://groups.google.com/forum/#!forum/lamp_tutorial

or

mail : tuxfux.hlp@gmail.com for access to group.

Lastly, I would like your suggestions and comments on the tutorial. Please feel free to reach me in case of any grammatical or content issues.

Mail me : tuxfux.hlp@gmail.com