

Python2 and Python 3 with CGI

“Python design philosophy emphasizes code readability”

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

- ✦ **Python is Interpreted**
- ✦ **Python is Compiled**
- ✦ **Python is Interactive**
- ✦ **Python is Object-Oriented**
- ✦ **Python is Beginner's Language**

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

Python's feature highlights include:

- ✦ **Easy-to-learn**
- ✦ **Easy-to-read**
- ✦ **Easy-to-maintain**
- ✦ **A broad standard library**
- ✦ **Interactive Mode**
- ✦ **Portable**
- ✦ **Extendable**
- ✦ **Databases**
- ✦ **GUI Programming**
- ✦ **Scalable**

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. It features a fully dynamic type system and automatic memory management, similar to that of Scheme, Ruby, Perl, and Tcl. Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems.

What you need to get started:-

Python 3 (python.org/download)

Most stable release of python is 2.6.5 and 3.1.2

Note :- Python 3.x code is NOT backward-compatible with python 2.x

For eg in python 2, for print, you can `>>> print "lw"`

But in python 3, for print, you have to put parenthesis `>>> print ("lw")`

Imp: you can install python2 and python3 both version on one computer without any trouble

What can you do with python?

- System admin / automation
- Database/ n/w / internet access
- GUI development – games
- Science and math programming
- HW / SW testing
- Hacking task and penetration testing tools

Examples of Python Applications:-

Bittorrent

- Youtube
- Google (App Engine)
- Reddit
- Pixas / illustrator Magic
- NASA / Los Alamos / Fermi LAB
- Spam Bayes / Trac
- Anaconda (fedora / redhat)

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

For example:

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> a=45
```

```
>>> b="redhat"
```

```
>>> c=5.4
```

```
>>> print a
```

```
45
```

```
>>> print b
```

```
redhat
```

```
>>> print c
```

```
5.4
```

Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

- i) **Integer**
- ii) **String**
- iii) **float**
- iv) **tuple**
- v) **list**
- vi) **Dictionary**

MUTABLE VS. IMMUTABLE

MUTABILITY OF COMMON TYPES

The way I like to remember which types are mutable and which are not is that containers and user-defined types are generally mutable while everything else is immutable. Then I just remember the exceptions like tuple which is an immutable container and frozen set which is an immutable version of set (which makes sense, so you just have to remember tuple).

The following are immutable objects:

- Numeric types: int, float, complex
- string
- tuple
- frozen set
- bytes

The following objects are mutable:

- list
- dict
- set
- byte array

Basic Operation upon Integer and Float.

Note: basic operation include listed here

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>>
```

```
>>> a=90
```

```
>>> b=3
```

```
>>> c=12
```

```
>>>
```

```
>>> a + b
```

```
93
```

```
>>> a - c
```

```
78
```

```
>>> a * b
```

```
270
```

```
>>> a / 3
```

```
30
```

```
>>> a / b
```

```
30
```

```
>>> a % b
```

```
0
```

Printing Message and taking Input user from users:

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> print "Hello world !! "
```

```
Hello world !!
```

```
>>>
```

```
>>> print "Hello world !! \n"
```

```
Hello world !!
```

User INPUT:

Python2 using raw_input() for input

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> raw_input("Enter your data : ")
```

```
Enter your data : 56
```

```
'56'
```

--- >> **By default data type is string**

```
>>> raw_input("Enter your data : ")
```

```
Enter your data : hello
```

```
'hello'
```

```
>>> x=raw_input("Enter your data : ")
```

```
Enter your data : 56
```

```
>>> type(x)
```

```
<type 'str'>
```

```
>>> print x
```

```
56
```

```
>>> y=int(x)      ,,>> converting from string to Integer
```

```
>>> print y
```

```
56
```

```
>>> print type(y)
```

```
<type 'int'>
```

```
>>>
```

String Manipulation :

we can perform lots of operation upon string but some most used operation listed here

```
vimal@linux:~$
```

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
>>> s="this is testing is python"
>>> print s
this is testing is python
>>>
```

i) Length count

```
>>> len(s)
25
```

ii) String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example –

```
>>> x="linux"
>>>
>>> print "python can run over %s" % (x)

python can run over linux
>>>
```


Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

l ii) Built-in String Methods

a) counting for substring

```
>>> s="new data generated"
```

```
>>>
```

```
>>> s.count("e")
```

```
4
```

```
>>> s.count("e",0,4)
```

```
1
```

b) String Indexing

```
>>> s
'new data generated'
>>>
>>> print s[0]
n
>>> print s[3]

>>> print s[5]
a
>>> print s[0:7]

new dat
>>> print s[0:]
new data generated
>>>
```

c) string splitting

```
>>> s
'new data generated'
>>> s.split(" ")
['new', 'data', 'generated']
>>>
```

Note: Here i seperated with “ ” space

d) string repetition

```
>>> s  
  
'new data generated'  
  
>>> print s*5  
  
new data generatednew data generatednew data generatednew data generatednew data generated  
  
>>>
```

e) String replace

```
>>> s  
  
'new data generated'  
  
>>> s.replace("data","logs")  
  
'new logs generated'  
  
>>>
```

f) String Join function

```
>>> s  
  
'new data generated'  
  
>>> print " ".join(s)  
  
n e w   d a t a   g e n e r a t e d  
  
>>>
```

Note: here every character of string is join with space

g)

Strip

Python strings have the `strip()`, `lstrip()`, `rstrip()` methods for removing

any character from both ends of a string. If the characters to be removed are not specified then white-space will be removed

```
>>> s=" hey i don't know "
```

```
>>> s.strip(" ")
```

```
"hey i don't know"
```

```
>>> s.strip("\n")
```

```
" hey i don't know "
```

```
>>> s.lstrip()
```

```
"hey i don't know "
```

```
>>>
```

```
>>> s.rstrip()
```

```
" hey i don't know"
```

```
>>>
```

Note:

strip()>> remove from both side

lstrip() >> remove from starting

rstrip()>> remove from endpoints

l) string concatenation

```
>>> s
```

```
" hey i don't know "
```

```
>>>
```

```
>>> x=" who are you..??"
```

```
>>>
```

```
>>> s + x
```

```
" hey i don't know  who are you..??"
```

```
>>>
```

Tuple:

This is collection of multiple values of homogenous or heterogeneous values

```
>>> y=(1,4,66,"redhat")
```

checking type of variable

```
>>> type(y)
```

```
<type 'tuple'>
```

checking length

```
>>> len(y)
```

```
4
```

Some basic operations

i) Indexing of tuple

```
>>> y[0]
```

```
1
```

```
>>> y[0:3]
```

```
(1, 4, 66)
```

```
>>>
```

ii) Adding tuple

```
>>> x=(23,7,"new line")
```

```
>>>
```

```
>>> y=(55,"this is",222)
```

```
>>> x + y
```

```
(23, 7, 'new line', 55, 'this is', 222)
```

```
>>>
```

iii) Searching in tuple

```
>>> x
```

```
(23, 7, 'new line')
```

```
>>> 7 in x
```

```
True
```

```
>>>
```

List :

this is collection of homegeneous or hetrogenious value which support muttability

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

```
>>> a=[2,4,"hiii",6.4]
```

```
>>> type(a)
```

```
<type 'list'>
```

```
>>>
```

Some Basic Opertions

```
>>> a=[2,4,"hiii",6.4]
```

type checking

```
>>> type(a)
```

```
<type 'list'>
```

Indexing of list

```
>>> print a[0]
```

```
2
```

```
>>>
```

```
>>> print a[0:3]
```

```
[2, 4, 'hiii']
```

```
>>> print a[0:2]
```

```
[2, 4]
```

```
>>>
```

length count

```
>>> len(a)
```

```
4
```

Insert and append in list

```
>>> a
```

```
[2, 4, 'hiii', 6.4]
```

```
>>> a.append(7)
```

```
>>> a
```

```
[2, 4, 'hiii', 6.4, 7]
```

```
>>>
```

```
>>> a.insert("me",1)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: an integer is required

```
>>> a.insert(1,"me")
```

```
>>> a
```

```
[2, 'me', 4, 'hiii', 6.4, 7]
```

```
>>>
```

Dictionary:

A dictionary optimizes element lookups. It associates keys to values. Each key must have a value. Dictionaries are used in many programs.

With square brackets, we assign and access a value at a key. With `get()` we can specify a default result. Dictionaries are fast. We create, mutate and test them

Instead: We can use the `get()` method with one or two arguments. This does not cause any annoying errors. It returns `None`.

Argument 1: The first argument to `get()` is the key you are testing. This argument is required.

Argument 2: The second, optional argument to `get()` is the default value. This is returned if the key is not found.

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>> d={}
```

```
>>> d[0]="zero"
```

```
>>> d[1]="hello"
```

```
>>> d[2]=4545
```

```
>>>
```



```
>>> d
```

```
{0: 'zero', 1: 'hello', 2: 4545}
```

```
>>> type(d)
```

```
<type 'dict'>
```

```
>>>
```

```
<type 'dict'>
```

Its a combination of keys and values pair

What are Conditions in python

Conditions tests if a something is True or False, and
it uses Boolean values

(type bool) to check that.

You see that conditions are either True or False (with no quotes!).

Note: for terminating condition in python you need to place (:) at theend.

```
>>> if 23 > 5 :
```

```
...     print "hello world"
```

```
... else :
```

```
...     print "no hello world"
```

```
...
```

```
hello world
```

Important: python follow Indentation for writing code inside loops and conditional statement.

Else if in python

```
>>> a=10

>>>

>>> if a < 5 :

...     print "hiii"

... elif a > 5 :

...     print "yes again !!"

... else :

...     print "no chance"

...

yes again !!

>>>
```

While Loop in python

A conditional program that can not be terminated if its condition goes true by default

he syntax of a while loop in Python programming language is –

while expression:
statement(s)

```
>>> while 3 > 2 :
```

```
...     print "hiii"
```

```
...
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

```
hiii
```

Note: this statement will always perform print operation

Another example

```
>>> while c < 9 :
```

```
...     print "the number is :",c
```

```
...     c = c + 1
```

```
...
```

```
the number is : 0
```

```
the number is : 1
```

```
the number is : 2
```

```
the number is : 3
```

```
the number is : 4
```

```
the number is : 5
```

```
the number is : 6
```

```
the number is : 7
```

```
the number is : 8
```

```
>>>
```

For Loop in python :

It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

```
for iterating_var in sequence:
    statements
```

```
>>> for i in "linuxworld":
...     print i
...
l
i
n
u
x
w
o
r
l
d
```

Note: we can tuple and list variable in for loop

```
>>> primes = [2, 3, 5, 7]
```

```
>>> for prime in primes:
```

```
...     print prime
```

```
...
```

```
2
```

```
3
```

```
5
```

```
7
```

Regular-expression in Python

we are starting with python regular expression here are some examples are given below.

For matchin patern we need to import a module named re (regular expression)

There are some special characters

there are two special functions in regular expression for finding matches

A) `re.match` ----->> find only starting of the string

B) `re.search` ----->> find all the match starting to end

Important : `re.sub`----->> to find and replace

example for `sub`

```
>>> st1
'redhat linux one'
>>>
>>>
>>> m1=re.sub(r'redhat','LINUX',st1)
>>>
>>> m1
```

'LINUX linux one'

some special character or Identifiers

```
\w    -- matches word character
\W    -- matches nonword character
\s    --- matches white space
\S    ----
\d    --- any number
.     ----- anything except new line
```

Some modifiers

```
+ ..... match 1 or more
* ..... match 0 or more
? ..... match 0 or 1
^ ..... match starting of the string
$ ..... match last of the string
```

Example 1 :

```
root@ashulinux:~# python
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> st="welcome to redhat linux for learning"
>>>
>>> import re
>>> m=re.search("or",st)
>>> m.group()
'or'
>>> m=re.search("red",st)
>>> m.group()
'red'
```

Example 2 : match any letter followed by "ed" in given string

```
>>> m=re.search(".ed",st)
>>> m.group()
'red'
```

Note : here (.) indicates for matching any characters

```
>>> st
'welcome to redhat linux for learning'
>>> m=re.search("l....ing",st)

>>> m.group()
'learning'
```

Example 3: finding all matches

```
>>> data="redhat linux is awesome for redhat people"
>>> import re
>>> re.findall('redhat',data)
['redhat', 'redhat']
>>>
```

Function in python

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

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

Defining a Function

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

- Function blocks begin with the keyword **def** followed by the function name and parentheses `(())`.
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon `(:)` and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Defining User Define Function:

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>> def lw():
```

```
...     print "this is wat i want !1"
```

```
...     print "i can execute multiple line"
```

```
...     print "this is reuse of code"
```

```
...
```




```
>>> lw()
```

```
this is wat i want !1
```

```
i can execute multiple line
```

```
this is reuse of code
```

```
>>>
```

Now Function with single parameter:

```
>>> def saysomething(x):
```

```
...     print "hey ",x
```

```
...
```

```
>>> saysomething("vimal")
```

```
hey vimal
```

```
>>>
```

Function with Default parameter:

```
>>> def new(x,y=10):
```

```
...     print x + y
```

```
...
```

```
>>> new(5)
```

```
15
```

```
>>> new(5,90)
```

```
95
```

```
>>>
```

Variable length argument:

```
>>> def varl(x,*y) :  
...     print x  
...     print y  
...  
>>> varl(1,5)  
  
1  
(5,)  
  
>>> varl(1,5,7,8,"redhat")  
  
1  
(5, 7, 8, 'redhat')  
  
>>>
```

Note: here *y is a tuple

Start Write Code in Files

Python is the language where you can use **(Interprator,compiler,IDE)**

here we are going to discuss to code in python in File

Note: You can use any Editor i am using **VI/VIM** in **Redhat linux**.

Step 1: write code like given below

```
vimal@linux:~$ vim code1.py
```

```
#!/usr/bin/python2
```

```
print "Hello World !!"
```

Step 2 : make it executable

```
vimal@linux:~$ chmod +x code1.py
```

Step 3: Run the program

I) Method 1

```
vimal@linux:~$ python code1.py
```

Hello World !!

II) Method 2

```
vimal@linux:~$ ./code1.py
```

Hello World !!

Modules in Python :-

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

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

I) Using Modules in Python

for using module you need to use import command

```
vimal@linux:~$ python
```

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import os
```

```
>>> os.system('date')
```

Wed Mar 2 09:01:46 IST 2016

0

Note: here **os** is the of module and **system** is the name function here to use internal commands os base operating system

More with Modules

i) Describe and exploring built_in Modules

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

```
>>> import commands
```

```
>>> dir(commands)
```

```
['__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'getoutput', 'getstatus',  
'getstatusoutput', 'mk2arg', 'mkarg']
```

```
>>>
```

Here **dir(module_name)** will show you listed function

Using More Modules:-

i) Using commands modules

To run any os related command from python interpreter

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> import commands
```

```
>>> dir(commands)
```

```
['_all_', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'getoutput', 'getstatus',  
'getstatusoutput', 'mk2arg', 'mkarg']
```

```
>>> commands.getoutput('date')
```

```
'Wed Mar 2 11:39:32 IST 2016'
```

```
>>> x=commands.getoutput('date')
```

```
>>> x
```

```
'Wed Mar 2 11:39:35 IST 2016'
```

```
>>>
```

II) Using time module

to check and change time from python this module was created

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

Type “help”, “copyright”, “credits” or “license” for more information.

```
>>>
```

```
>>> import time
```

```
>>> dir(time)
```

```
['__doc__', '__name__', '__package__', 'accept2dyear', 'altzone', 'asctime', 'clock', 'ctime', 'daylight',  
'gmtime', 'localtime', 'mktime', 'sleep', 'strptime', 'struct_time', 'time', 'timezone', 'tzname',  
'tzset']
```

```
>>> time.ctime()
```

```
'Wed Mar 2 11:41:26 2016'
```

```
>>>
```

```
>>> time.clock()
```

```
0.024445
```

```
>>> time.timezone
```

```
-19800
```

```
>>> time.sleep(5)      ...>>>>>>>>. wait for 5 second to interprator
```

III) Using math module

```
>>> import math
```

```
>>> dir(math)
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',  
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',  
'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow', 'radians',  
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

```
>>> math.sin(45)
```

```
0.8509035245341184
```

```
>>> math.sin(90)
```

```
0.8939966636005579
```

```
>>> math.sin(0)

0.0

>>> math.sqrt(36)

6.0

>>>

>>> math.factorial(4)

24

>>> math.log(10)

2.302585092994046

>>>
```

IV) Working with sys module

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import sys
```

```
>>> dir(sys)
```

```
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__stderr__', '__stdin__',
 '__stdout__', '__clear_type_cache__', '_current_frames', '_getframe', '_mercurial', '_multiarch',
 'api_version', 'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright',
 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
 'getdlopenflags', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',
 'gettrace', 'hexversion', 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'py3kwarning', 'pydebug',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'subversion', 'version', 'version_info', 'warnoptions']
```

```
>>>
```



>>> sys.version ...>> checking version of python

'2.7.6 (default, Jun 22 2015, 17:58:13) \n[GCC 4.8.2]'

>>> sys.path >> python path and environment variable

```
['', '/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-ss-client']
```

>>> sys.argv >> for using inline input

['']

Making own modules:

Now here we are going to create own module like python built-in

Step 1 : create a file and write code

vimal@linux:~\$ vim mycal.py

```
#!/usr/bin/python2
```

```
def lw():  
    print " Welcome to lw summer training !! "  
    print "Enjoy the pace and power of python and linux"
```

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

```
def mul(x,y,z):  
    print x * y * z
```

```
def motd():  
    print "This is message of the Day !!"
```


Step 2 : Importing the module

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import mycal
```

```
>>> dir(mycal)
```

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add', 'lw', 'motd', 'mul']
```

```
>>>
```

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import mycal
```

```
>>> dir(mycal)
```

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'add', 'lw', 'motd', 'mul']
```

```
>>> mycal.lw()
```

Welcome to lw summer training !!

Enjoy the pace and power of python and linux

```
>>>
```

```
>>> mycal.add(4,6)
```

```
10
```

File Handling in Python:-

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). Since, random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order.

1. Open file
2. read / write / append (operation)
3. Close file

Opening a File

Python has a built-in function `open()` to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> f=open('mycal.py','r') ....>> here file is open in read mode
```

```
>>> f.read()
```

```
'#!/usr/bin/python2\n\ndef lw():\n\n\tprint " Welcome to lw summer training !! "\n\tprint "Enjoy the  
pace and power of python and linux"\n\ndef add(x,y):\n\treturn x + y\n\ndef  
mul(x,y,z):\n\tprint x * y * z\n\ndef motd():\n\tprint "This is message of the Day !!"\n\n\n'
```

```
>>> f.close()
```

Note:

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode. The default is reading in text mode. In this mode, we get strings when reading from the file. On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files

Mode : Description

'r'	:	open file for reading purpose only
'w'	:	open file only for write purpose
'r+'	:	open file for read and write purpose but in this case file must be exists
'w+'	:	open file for read and write purpose but in this case file is not present the file will be created
'a'	:	It will create a file and open this in append mode but you can not read
'a+'	:	It will create a file and open this in append mode but you can read also

Examples:

i) open a file in write mode

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

```
>>> f=open('hiii.txt','w')
```

```
>>> f.write("count the days its not easy for u \n")
```

```
>>> f.write("knowledge is power \n")
```

```
>>> f.close()
```

```
>>>
```

II) open file for reading purpose

```
>>> f=open('hiii.txt','r')
```

```
>>> f.read()
```

```
'count the days its not easy for u \nknowledge is power \n'
```

```
>>> f.read()
```

```
"
```

```
>>> f.seek(0) .....>> sending pointer to first character
```

```
>>>
```

```
>>> f.readlines()
```

```
['count the days its not easy for u \n', 'knowledge is power \n']
```

```
>>>
```

Exception handling in Python :

When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error

Example :

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> if a < 3
```

```
File "<stdin>", line 1
```

```
    if a < 3
```

```
        ^
```

```
SyntaxError: invalid syntax
```

```
>>> 23 / 0
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: integer division or modulo by zero
```

```
>>>
```

Note: this kind of error will generate which can be logical or syntax error

Important: There is different kind of error in python and some of them are listed below:

AssertionError : Raised when assert statement fails.

AttributeError : Raised when attribute assignment or reference fails.

EOFError : Raised when the input() functions hits end-of-file condition.

FloatingPointError : Raised when a floating point operation fails.

GeneratorExit : Raise when a generator's close() method is called.

ImportError : Raised when the imported module is not found.

IndexError : Raised when index of a sequence is out of range.

KeyError : Raised when a key is not found in a dictionary.

KeyboardInterrupt : Raised when the user hits interrupt key (Ctrl+c or delete).

MemoryError : Raised when an operation runs out of memory.

NameError : Raised when a variable is not found in local or global scope.

NotImplementedError : Raised by abstract methods.

OSError: Raised when system operation causes system related error.

OverflowError : Raised when result of an arithmetic operation is too large to be represented.

ReferenceError : Raised when a weak reference proxy is used to access a garbage collected referent.

RuntimeError :Raised when an error does not fall under any other category.

StopIteration :Raised by next() function to indicate that there is no further item to be returned by iterator.

SyntaxError :Raised by parser when syntax error is encountered.

IndentationError :Raised when there is incorrect indentation.

TabError :Raised when indentation consists of inconsistent tabs and spaces.

SystemError :Raised when interpreter detects internal error.

SystemExit :Raised by sys.exit() function.

TypeError : Raised when a function or operation is applied to an object of incorrect type.

UnboundLocalError :Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.

UnicodeError :Raised when a Unicode-related encoding or decoding error occurs.

ValueError :Raised when a function gets argument of correct type but improper value.

ZeroDivisionError :Raised when second operand of division or modulo operation is zero.

Python Exception Handling - Try, Except and Finally

When an exception occurs in Python, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash. For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A. If never handled, an error message is spit out and our program come to a sudden, unexpected halt

Catching any Exceptions in Python

In Python, exceptions can be handled using a try statement. A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause. It is up to us, what operations we perform once we have caught the exception. Here is a simple example

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> try :
```

```
...     if 45 :
```

```
...         23 / 0
```

```
...     elif True :
```

```
...         "redhat" + 56
```

```
... except :
```



```
... print "error is handled by default !!"
```

```
...
```

error is handled by default !!

Catching Specific Exceptions in Python:

In the above example, we did not mention any exception in the except clause. This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch. A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs. We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code

```
>>> try :
```

```
... 78 / 0
```

```
... import skdfjdsllkf
```

```
... "redhat india" + 45
```

```
... except ZeroDivisionError:
```

```
... print "zero division error !!"
```

```
... except (TypeError, ImportError) :
```

```
... print "new error found !!! "
```

```
...
```

zero division error !!

```
>>>
```

try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources. For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI). In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

vimal@linux:~\$ python

Python 2.7.6 (default, Jun 22 2015, 17:58:13)

[GCC 4.8.2] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

```
>>> try :
```

```
...     print 5445
```

```
... except :
```

```
...     45 / 0
```

```
... finally
```

```
...     print "hello world"
```

```
...
```

```
5445
```

```
hello world
```

Python Socket Programming

I'm only going to talk about INET sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM sockets - unless you really know what you're doing (in which case this HOWTO isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that "socket" can mean a number of subtly different things, depending on context. So first, let's make a distinction between a "client" socket - an endpoint of a conversation, and a "server" socket, which is more like a switchboard operator. The client application (your browser, for example) uses "client" sockets exclusively; the web server it's talking to uses both "server" sockets and "client" sockets.

What is Socket..??

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Creating UDP socket :-

Here we are taking A server and Client example



AT Server Side:

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
```

```
[GCC 4.8.2] on linux2
```

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

```
>>>
```

```
>>> import socket
```

```
>>> s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) ....>>creating socket
```

```
>>> s.bind(("192.168.0.104",9999)) ....>> Binding socket
```

```
>>> s.recvfrom(100) .....>> receiving data
```

AT Client Side

```
vimal@linux:~$ python
```

```
Python 2.7.6 (default, Jun 22 2015, 17:58:13) [GCC 4.8.2] on linux2
```

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

```
>>> import socket
```

```
>>> s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
>>> s.sendto("hiii",("192.168.0.104",9999))
```

4

Now go To server side and check for message :

```
>> s.recvfrom(100)
```

```
('hiii', ('192.168.0.104', 48246))
```

```
>>>
```

Tcp socket :

socket.socket(): Create a new socket using the given address family, socket type and protocol number.

- socket.bind(address):** Bind the socket to **address**.
- socket.listen(backlog):** Listen for connections made to the socket. The **backlog** argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.
- socket.accept():** The return value is a pair (**conn**, **address**) where **conn** is a new socket object usable to send and receive data on the connection, and **address** is the address bound to the socket on the other end of the connection. At **accept()**, a new socket is created that is distinct from the named socket. This new socket is used solely for communication with this particular client. For TCP servers, the socket object used to receive connections is not the same socket used to perform subsequent communication with the client. In particular, the **accept()** system call returns a new socket object that's actually used for the connection. This allows a server to manage connections from a large number of clients simultaneously.
- socket.send(bytes[, flags]):** Send data to the socket. The socket must be connected to a remote socket. Returns the number of **bytes** sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.
- socket.close():** Mark the socket closed. all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed). Sockets are automatically closed when they are garbage-collected, but it is recommended to close() them explicitly.

Note that the **server** socket doesn't receive any data. It just produces **client** sockets. Each **clientsocket** is created in response to some other **client** socket doing a **connect()** to the host and port we're bound to. As soon as we've created that **clientsocket**, we go back to listening for more connections.



server.py

import socket

import time

create a socket object

```
serversocket = socket.socket(  
    socket.AF_INET, socket.SOCK_STREAM)
```

get local machine name

```
host = socket.gethostname()
```

```
port = 9999
```

bind to the port

```
serversocket.bind((host, port))
```

queue up to 5 requests

```
serversocket.listen(5)
```

while True:

establish a connection

```
clientsocket,addr = serversocket.accept()
```

```
print("Got a connection from %s" % str(addr))
```

```
currentTime = time.ctime(time.time()) + "\r\n"
```

```
clientsocket.send(currentTime.encode('ascii'))
```

```
clientsocket.close()
```



Client Side :

```
# client.py
import socket

# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# get local machine name
host = socket.gethostname()

port = 9999

# connection to hostname on the port.
s.connect((host, port))

# Receive no more than 1024 bytes
tm = s.recv(1024)
s.close()

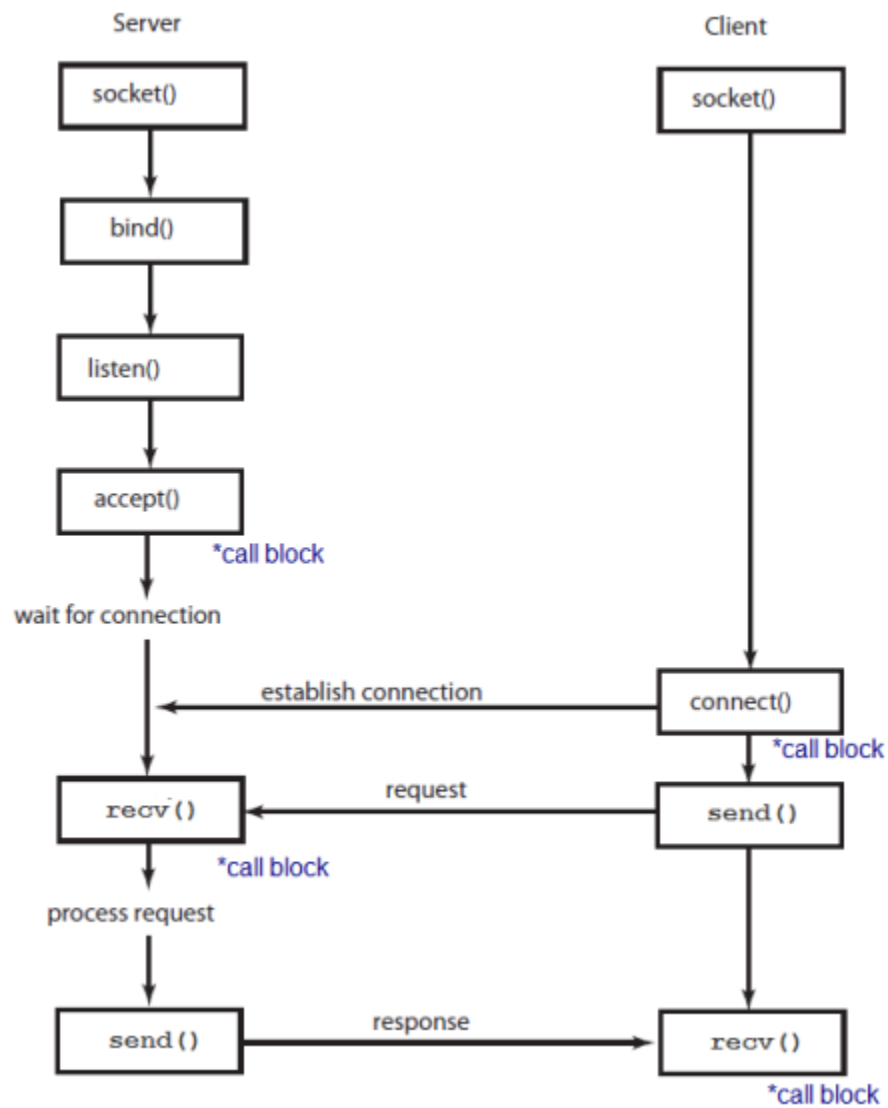
print("The time got from the server is %s" % tm.decode('ascii'))
```

The output from the run should look like this:

```
vimal@linux:~$ python server.py &
Got a connection from ('127.0.0.1', 54597)
```

```
vimal@linux:~$ python client.py
```

The time got from the server is Wed Jan 29 19:14:15 2014



Getting started with Python 3:

Python 3.0 final was released on December 3rd, 2008.

Python 3.0 (a.k.a. "Python 3000" or "Py3k") is a new version of the language that is incompatible with the 2.x line of releases. The language is mostly the same, but many details, especially how built-in objects like dictionaries and strings work, have changed considerably, and a lot of deprecated features have finally been removed. Also, the standard library has been reorganized in a few prominent places.

Note: You can download python setup from “python.org/download”

Important: IN python 3 everything is Object this is one of the Language which is
100 % ObjectOriented

Some Basic Difference in Python 2 and Python3:

i) Print Function:

```
vimal@linux:~$ python3
```

Python 3.4.3 (default, Oct 14 2015, 20:28:29)

[GCC 4.8.4] on linux

Type "help", "copyright", "credits" or "license" for more information.

```
>>>
```

```
>>> print("Hello world !! ")
```

```
Hello world !!
```

```
>>>
```

II) User Input in python

```
vimal@linux:~$ python3
```

```
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
```

```
[GCC 4.8.4] on linux
```

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

```
>>>
```

```
>>> x=input("plz enter any value : ")
```

```
plz enter any value : 456
```

```
>>> type(x)
```

```
<class 'str'>
```

```
>>> print(x)
```

```
456
```

```
>>>
```

Reality of print function in python3

```
>>> print("hii")          ---- >> by default print("hiii",end="\n")
```

```
hii
```

```
>>>
```

```
>>> print("hii",end="")
```

```
hii>>>
```

For any further Studies you can go through:

<https://www.python.org/>

Python CGI Programming:

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA and NCSA.

What is CGI?

The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.

The current version is CGI/1.1 and CGI/1.2 is under progress.

Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

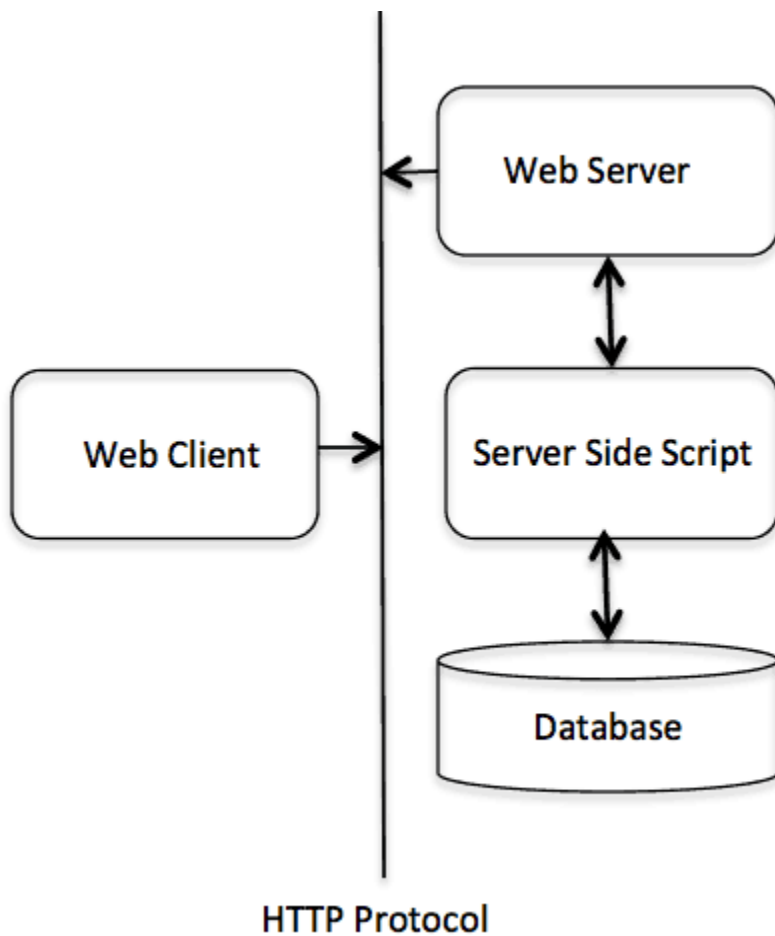
Your browser contacts the HTTP web server and demands for the URL, i.e., filename.

Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.

Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc

CGI Architecture Diagram



Web Server Support and Configuration

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as `/var/www/cgi-bin`. By convention, CGI files have extension `.cgi`, but you can keep your files with python extension `.py` as well.

By default, the Linux server is configured to run only the scripts in the `cgi-bin` directory in `/var/www`. If you want to specify any other directory to run your CGI scripts, comment the following lines in the `httpd.conf` file

```
<Directory "/var/www/cgi-bin">
```

```
    AllowOverride None
```

```
    Options ExecCGI
```

```
    Order allow,deny
```

```
    Allow from all
```

```
</Directory>
```

```
<Directory "/var/www/cgi-bin">
```

```
Options All
```

```
</Directory>
```

First CGI Program

Here is a simple link, which is linked to a CGI script called `hello.py`. This file is kept in `/var/www/cgi-bin` directory and it has following content. Before running your CGI program, make sure you have change mode of file using `chmod 755 lw.py` UNIX command to make file executable

```
root@desktop83 Desktop: cat /var/www/cgi-bin/lw.py
```

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

```
print "Content-type:text/html"
```

```
print ""
```

```
print '<html>'
```

```
print '<head>'
```

```
print '<title>start programming </title>'
```

```
print '</head>'
```

```
print '<body>'
```

```
print '<h2>CGI started </h2>'
```

```
print '</body>'
```

```
print '</html>'
```

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

Passing Information using GET method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the ? character as follows –

<http://192.168.0.83/cgi-bin/hello.py?key1=value1&key2=value2>

Simple FORM Example: GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script

This is my html page using form:

```
root@desktop83 Desktop: cat /var/www/html/index.html
```

```
<form action="/cgi-bin/hello.py" method="get">
```

```
First Name: <input type="text" name="first_name"> <br />
```

```
Last Name: <input type="text" name="last_name" />
```

```
<input type="submit" value="Submit" />
```

```
</form>
```

```
root@desktop83 Desktop: cat /var/www/cgi-bin/hello.py
```

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

```
# Import modules for CGI handling
```

```
import cgi, cgiib
```

```
# Create instance of FieldStorage
```

```
form = cgi.FieldStorage()
```

Get data from fields

```
first_name = form.getvalue('first_name')
last_name = form.getvalue('last_name')

print "Content-type:text/html\r\n\r\n"
print "<html>"
print "<head>"
print "<title>Hello - Second CGI Program</title>"
print "</head>"
print "<body>"
print "<h2>Hello %s %s</h2>" % (first_name, last_name)
print "</body>"
print "</html>"
```

SO this was the basic Idea and Implementation of python CGI programming

PYTHON-CGI

Python Advanced Topics:

Database Handling using python:

Python MySQL database server connectivity and management

Here i am going to discuss how to install and connect MySQL database server using python

Step 1: Install MySQL server related all the software in your redhat 6

```
[root@exam ~]# rpm -qa | grep -i mysql
```

```
mysql-5.1.66-2.el6_3.x86_64
```

```
mysql-devel-5.1.66-2.el6_3.x86_64
```

```
mysql-bench-5.1.66-2.el6_3.x86_64
```

```
mysql-server-5.1.66-2.el6_3.x86_64
```

```
mysql-libs-5.1.66-2.el6_3.x86_64
```

```
perl-DBD-MySQL-4.013-3.el6.x86_64
```

```
mysql-connector-odbc-5.1.5r1144-7.el6.x86_64
```

```
mysql-test-5.1.66-2.el6_3.x86_64
```

If these software are not installed then you can install them using yum

```
[root@exam ~]# yum install mysql*
```

OR

```
[root@exam ~]# yum install mysql-server mysql mysql-connector-odbc mysql-libs
```

Step 2: Installing python and mysql module

```
[root@exam ~]# yum install mysql-python
```

Note: if you don't have rpm package then go to given link

```
#####  
https://sourceforge.net/projects/mysql-python/  
#####
```

Step 3: Start the service of mysql

```
[root@exam ~]# service mysqld restart
```

Step 4: now check with python

```
[root@exam ~]# python
```

Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)

[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import MySQLdb
```

```
>>> dir(MySQLdb)
```

```
['BINARY', 'Binary', 'Connect', 'Connection', 'DATE', 'DATETIME', 'DBAPISet', 'DataError',  
'DatabaseError', 'Date', 'DateFromTicks', 'Error', 'FIELD_TYPE', 'IntegrityError', 'InterfaceError',  
'InternalError', 'MySQLError', 'NULL', 'NUMBER', 'NotSupportedError', 'OperationalError',  
'ProgrammingError', 'ROWID', 'STRING', 'TIME', 'TIMESTAMP', 'Time', 'TimeFromTicks',  
'Timestamp', 'TimestampFromTicks', 'Warning', '__all__', '__author__', '__builtins__', '__doc__',  
'__file__', '__name__', '__package__', '__path__', '__revision__', '__version__', '__mysql__',  
'apilevel', 'connect', 'connection', 'constants', 'debug', 'escape', 'escape_dict',  
'escape_sequence', 'escape_string', 'get_client_info', 'paramstyle', 'release', 'result',  
'server_end', 'server_init', 'string_literal', 'test_DBAPISet_set_equality',  
'test_DBAPISet_set_equality_membership', 'test_DBAPISet_set_inequality',  
'test_DBAPISet_set_inequality_membership', 'thread_safe', 'threadsafety', 'times',  
'version_info']
```

Note:

What is MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API.

Some example for mysql connection:-

```
[root@exam ~]# python
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import MySQLdb
>>> x=MySQLdb.connect('localhost')
>>> import MySQLdb
>>> x=MySQLdb.connect('localhost','root','redhat')
```

Important: Some basic operation of database using python

```
[root@exam ~]# python
```

```
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import MySQLdb
>>> x=MySQLdb.connect('localhost','root','redhat')

>>> y=x.cursor()
>>>
```

Note : Creating Database

```
>>> y.execute('use mysql;')
0L
>>> y.execute('show databases;')
4L
```

Important: execute function will only print number of lines in Query

Use Database:

```
>>> y.execute('use lw;')
```

Creating table in database:

```
>>> sql = '''CREATE TABLE EMPLOYEE (
...     FIRST_NAME CHAR(20) NOT NULL,
...     LAST_NAME CHAR(20),
...     AGE INT,
...     SEX CHAR(1),
...     INCOME FLOAT )'''
>>> y.execute(sql)
0L
```

Commit your changes in database:

```
>> x.commit()
```

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either `fetchone()` method to fetch single record or `fetchall()` method to fetch multiple values from a database table.



fetchone(): It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.

fetchall(): It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

rowcount: This is a read-only attribute and returns the number of rows that were affected by an execute() method.

[root@desktop67 ~]# python

Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)

[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

>>>

>>> import MySQLdb

>>> x=MySQLdb.connect('localhost')

>>> y=x.cursor()

>>> y.execute('show databases;')

4L

>>> y.fetchall()

((('information_schema',), ('lw',), ('mysql',), ('test',)))

>>>



```
>>> y.fetchall()
```

Note : using these steps you can perform any operation upon database.

The most common form of CGI programming handles interaction between user input in an HTML form and a database. HTML forms consist of several input fields, each given a key identifier, and a submit button that sends the data in a query string. Fortunately, the built-in Python CGI module parses this information in a class called `cgi`.

Web Browsing

62

Your browser contacts the HTTP web server and demands for the URL, i.e., filename.

Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.

Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

Web server support and apache configuration:

Before you proceed with CGI Programming, make sure that your Web Server supports CGI and it is configured to handle CGI Programs. All the CGI Programs to be executed by the HTTP server are kept in a pre-configured directory. This directory is called CGI Directory and by convention it is named as `/var/www/cgi-bin`. By convention, CGI files have extension `.cgi`, but you can keep your files with python extension `.py` as well.

```
[root@desktop67 ~]# vim /etc/httpd/conf/httpd.conf
```

```
Directory "/var/www/cgi-bin">
```

```
    AllowOverride None
```

```
    Options +ExecCGI
```

```
    Order allow,deny
```

```
    Allow from all
```

```
</Directory>
```

Executing First code in python CGI:

Writing a simple code in python

```
[root@desktop67 cgi-bin]# pwd
```

```
/var/www/cgi-bin
```

```
[root@desktop67 cgi-bin]# cat first.py
```

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

```
print "content-type:text/html"
```

```
print ""
```

```
print ""\n
```

```
<html>
```

```
<title>web programming </title>
```

```
<h1> hello world </h1>
```

```
</html>
```

```
""""
```

```
[root@desktop67 cgi-bin]# chmod +x first.py
```

Now : Go to your web browser and access file <http://IP/cgi-bin/first.py>

Note:

It is necessary that the script outputs the HTTP header. The HTTP header consists of one or more messages followed by a blank line. If the output of the script is to be interpreted as HTML then the content type will be text/html. The blank line signals the end of the header and is required:

```
print "Content-Type: text/html"
```

```
print
```

or

Many times the blank line will be written as \n:

```
print "Content-Type: text/html\n"
```

More About HTTP headers:-

The line Content-type:text/html\r\n\r\n is part of HTTP header which is sent to the browser to understand the content. All the HTTP header will be in the following form –

There are few other important HTTP headers, which you will use frequently in your CGI Programming.

Content-type: A MIME string defining the format of the file being returned.

Example is Content-type:text/html

Expires: Date The date the information becomes invalid. It is used by the browser to decide when a page needs to be refreshed. A valid date string is in the format 01 Jan 1998 12:00:00 GMT.



Location: URL The URL that is returned instead of the URL requested. You can use this field to redirect a request to any file.

Last-modified: Date The date of last modification of the resource.

Content-length: N The length, in bytes, of the data being returned. The browser uses this value to report the estimated download time for a file.

Set-Cookie: String Set the cookie passed through the string

GET and POST Methods

You must have come across many situations when you need to pass some information from your browser to web server and ultimately to your CGI Program. Most frequently, browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

Simple FORM Example:

GET Method

This example passes two values using HTML FORM and submit button. We use same CGI script first.py to handle this input.

```
[root@desktop67 cgi-bin]# cd /var/www/html/
```

```
[root@desktop67 html]# vim index.html
```

```
[root@desktop67 html]# cat index.html
```

```
<html>
```

```
<form action="/cgi-bin/first.py" method="GET">
```

```
enter data : <input type="text" name="n">
```

```
    <input type="submit" value="Hit">
```

```
</form>
```

```
</html>
```

```
[root@desktop67 cgi-bin]# cd /var/www/cgi-bin/
```

```
[root@desktop67 cgi-bin]# cat first.py
```

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

```
print "content-type:text/html"
```

```
print ""
```

```
import cgi
```

```
x=cgi.FieldStorage()
```

```
y=x.getvalue('n')
```

```
print y
```

```
[root@desktop67 cgi-bin]#
```

Now you can use your data as per your requirement

Now Sending Data Using POST method:

```
[root@desktop67 cgi-bin]# cd /var/www/html/
```

```
[root@desktop67 html]# vim index.html
```

```
[root@desktop67 html]# cat index.html
```

```
<html>
```

```
<form action="/cgi-bin/first.py" method="POST">
```

```
enter data : <input type="text" name="n">
```

```
        <input type="submit" value="Hit">
```

```
</form>
```

```
</html>
```

```
[root@desktop67 cgi-bin]# cd /var/www/cgi-bin/
```

```
[root@desktop67 cgi-bin]# cat first.py
```

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

```
print "content-type:text/html"
```

```
print ""
```

```
import cgi
```

```
x=cgi.FieldStorage()
```

```
y=x.getvalue('n')
```

```
print y
```

Note : Some more HTML web methods to pass data

Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box –

```
[root@desktop67 cgi-bin]# cat /var/www/html/index.html
```

```
<html>
```

```
<form action="/cgi-bin/first.py" method="GET">
```

```
<textarea name="n" cols="40" rows="4">
```

```
type your data here
```

```
</textarea>
```

```
<input type="submit" value="send">
```

```
</form>
```

```
</html>
```

```
[root@desktop67 cgi-bin]# cat /var/www/cgi-bin/first.py
```

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

```
print "content-type:text/html"
```

```
print ""
```

```
import cgi
```

```
x=cgi.FieldStorage()
```

```
y=x.getvalue('n')
```

```
print y
```

Using Cookies in CGI

HTTP protocol is a stateless protocol. For a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. How to maintain user's session information across all the web pages?

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

Expires: The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.

Domain: The domain name of your site.

Path: The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.

Secure: If this field contains the word "secure", then the cookie may only be retrieved with a secure **server**. If this field is blank, no such restriction exists.

Name=Value: Cookies are set and retrieved in the form of key and value pairs.

Setting up Cookies

It is very easy to send cookies to browser. These cookies are sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. Setting the cookies is done as follows –

```
[root@desktop67 cgi-bin]# cat first.py
```

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

```
print "Set-Cookie:UserID=ashutoshh;\r\n"
```

```
print "Set-Cookie:Password=vimal;\r\n"
```

```
print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"
```

```
print "Set-Cookie:Domain=instructor.example.com;\r\n"
```

```
print "Set-Cookie:Path=/perl;\r\n"
```

```
print "Content-type:text/html\r\n\r\n"
```

```
----- Rest of HTML code -----
```

Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP_COOKIE and they will have following form –

key1=value1;key2=value2;key3=value3....

Here is an example of how to retrieve cookies.

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

```
# Import modules for CGI handling
```

```
from os import environ
```

```
import cgi, cgiib
```

```
if environ.has_key('HTTP_COOKIE'):
```

```
    for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):
```

```
        (key, value) = split(cookie, '=');
```

```
        if key == "UserID":
```

```
            user_id = value
```

```
        if key == "Password":
```

```
            password = value
```

```
print "User ID = %s" % user_id
```

```
print "Password = %s" % password
```

File Upload Example

To upload a file, the HTML form must have the enctype attribute set to multipart/form-data. The input tag with the file type creates a "Browse" button.

```
<html>

<body>

    <form enctype="multipart/form-data"

        action="save_file.py" method="post">

        <p>File: <input type="file" name="filename" /></p>

        <p><input type="submit" value="Upload" /></p>

    </form>

</body>

</html>
```

This is python file

```
#!/usr/bin/python

import cgi, os

import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.

fileitem = form['filename']
```

```
# Test if the file was uploaded

if fileitem.filename:

    # strip leading path from file name to avoid

    # directory traversal attacks

    fn = os.path.basename(fileitem.filename)

    open('/tmp/' + fn, 'wb').write(fileitem.file.read())

    message = 'The file "' + fn + '" was uploaded successfully'

else:

    message = 'No file was uploaded'

print """\n
Content-Type: text/html\n
<html>
<body>
    <p>%s</p>
</body>
</html>
"" % (message,)
```