How to Import MYSQLdb For Use in Program?

Once installed the module, you can import it like any other module:

```
import MySQLdb
```

A Class Called Table

To access a database, it is best to define a class. Here I define a class called Table which allows me to retrieve data (__getitem__) and to know the number of records (__len__).

```
class Table:
     def __init__(self, db, name):
          self.db = db
          self.name = name
          self.dbc = self.db.cursor()

     def __getitem__(self, item):
          self.dbc.execute("select * from %s limit %s, 1" %(self.name, item))
          return self.dbc.fetchone()

     def __len__(self):
          self.dbc.execute("select count(*) from %s" % (self.name))
          l = int(self.dbc.fetchone()[0])
          return l
```

Defining an Instance of Table

We need to define the database and create an instance of Table.

```
db = MySQLdb.connect( db="testable")
records = Table(db, "test")
```

Here the database name is 'testable'. The table from 'testable' which we will query is 'test'. It may be stating the obvious, but before executing the program, you must ensure that you actually have access to a database and table by these names (or whatever names you insert into the program). As usual, one can tweak these lines so that the values are derived from the command line by using sys.argv[1] and sys.argv[2], respectively.

We shall instead take the value of sys.argv[1] as the value for the query. So, import the sys module by adding ", sys" to the line where we imported MySQLdb. If you are doing this in the Python shell, simply type:

```
>>> import sys
```

Querying the Database

We need to define a main function in which we query the database. The main() function will include a simple loop for iterating through the rows of the table as follows:

```
def main():
     for c in xrange(len(records)):

          fvalues = records[c]

          print fvalues

          if fvalues[0] == sys.argv[1]:

               print fvalues[1]
```

Here we set up a for loop which runs from 1 (understood) to whatever the length of the table is. For each row, the entire row is read into 'fvalues'. If the first value in the row is equal to the string the program received from the user, the record is printed.

Getting MySQLdb

Perhaps the easiest Python module to use with MySQL is MySQLdb. In case you have not gone through the tutorial on reading from a MySQL database, MySQLdb is available from Sourceforge. The main documentation on this module may also be found at Sourceforge

Taking Data From the Command Line

We will be taking the data from the command line. To do this, we will use the sys module, so add to the same line ", sys". The import line should now read as follows:

```
import MySQLdb, sys
```

If you are doing this in the Python shell, simply type:

```
>>> import sys
```

Python will not be bothered by the late addition.
Then define two variables, id and input, according to the sys.argv[] arguments:

```
id = sys.argv[1]

input = sys.argv[2]
```

This tells Python that the first argument after the program's name should be assigned to the variable 'id'. Likewise, the second argument gets assigned to 'input'.

A Touch of Class

It is best to define a class in order to access a database. Here I again define a class called Table which allows me to insert data (additem). Unlike the example code for reading from the database, we here do not need to know the length of the database to read through all of its records.

The following code presupposes a table with two columns in it: the first field is the record identifier and the second is the record content.

```
class Table:

     def __init__(self, db, name):

          self.db = db

          self.name = name

          self.dbc = self.db.cursor()


     def additem(self, item):

          sql = "INSERT INTO " + self.name + " VALUES(" + id + ", " + item + ")"
          self.dbc.execute(sql)

          return
```

Rather than define the statement within the argument of self.dbc.execute, I here define it and then call it as an argument. If you have trouble with your script and it is giving unexpected output, try to define the variables separately from the arguments and then print them out. This will give you insight into the flow of the program and enable you to see where things are going awry.

The main() Thing

In the definition of the main() function, we need to define the database and create an instance of Table.

```
def main():

      db = MySQLdb.connect( db="testable")

      table = Table(db, "test")

      table.additem()
```

Here the database name is 'testable'. The table from 'testable' which we will query is 'test'. It may be stating the obvious, but before executing the program, you must ensure that you actually have access to a database and table by these names (or whatever names you insert into the program). As usual, one can tweak these lines so that the values are derived from the command line by using sys.argv[1] and sys.argv[2], respectively.

Calling the main() Function

We need to call the main function. In keeping the main activity of the program within a function, the program remains compartmentalized and maintainable. So type the following to finish the program:

```
if __name__ == '__main__':

      main()
```

Executing the Program

With this class and function, one can easily add to MySQL tables on the fly. As it stands, the program expects input like this when it is executed:

>python insertmysql.py <id> <data> If the program allowed for the database, user, and table to be defined at run-time (i.e., when the program is called), you then have a very flexible data insertion program that you can call from other programs.

While reading and writing is about all most people do with databases, if you would like to read more about MySQLdb, see the MySQLdb User's Guide at SourceForge .

Psycopg and Python: Perfect Together

The module we will use for this tutorial is again psycopg. It is available from http://www.initd.org/projects/psycopg1. So download it and install it using the directions that come with the package.

Once it is installed, you can import it like any other module:

```
# libs for database interface

import psycopg
```

If you would like your program to take input from the keyboard, you will also want to import the sys module. In this tutorial, I will illustrate how every major part of the SQL statement can be input from the keyboard. So, let your import line read as follows:

```
import sys, psycopg
```

Receiving Command Line Arguments for the PostgreSQL Statement - I

Before opening a connection to the database, we should attend to the variables we would like to define. As we intend to read from the database, we will be using a SELECT statement; in our case, we will ask for all data that matches a given string to be returned unformatted. In the interest of increased flexibility on our returned data, we will also use a WHERE clause. So the essential skeleton of the statement to be run looks like this:

SELECT * FROM <table> WHERE <column> <operator> <string> As you can probably guess, we will be inputting four pieces of data:

- **table:** the name of the table from the given database
- **column:** the name of the column within which the operation is to be performed
- **operation:** the actual operation to be performed; this may be any of the unary or boolean operations or a simple "IS" or "IS NOT"
- **string:** the string to be evaluated against the data

SQL gurus will rightly object at the absence of the semi-colon to end the statement. Psycopg, however, takes care of line termination signals for us.

Receiving Command Line Arguments for the PostgreSQL Statement - II

With that understanding, we then need to assign values to these variables using the sys module. The sys module has an attribute argv which is an array holding arguments from when the program is executed. By way of example, when one uses the shell command 'mkdir', the name of the directory to be created is the first (and only) argument of the command. If using Python's sys.argv, this argument would be sys.argv[1] -- the name by which the program is executed is always sys.argv[0]. Each additional argument follows in sequence.

Our program will take four arguments, one for each part of the SELECT statement to be made.

```
table = sys.argv[1]

column = sys.argv[2]

string = sys.argv[3]

operation = sys.argv[4]
```

Connecting to PostgreSQL Through Psycopg

To open a connection to a database, psycopg needs two arguments: the name of the database ('dbname') and the name of the user ('user'). If the program is to be executed in the name of a user other than the one used for the PostgreSQL account, you will also need to use the 'password=' option. The syntax for opening a connection follows this format:

```
<variable name for connection> = psycopg.connect('dbname=<dbname>', 'user=<user>')
```

For our database, we shall use the database name 'Melange' and the username 'tempsql'. For the connection object within the program, let's use the variable 'connection'. As mentioned, we are writing this program without classes and without any other function than main(). So, the beginning of main(), including our connection command will read as follows:

```
def main():

     connection = psycopg.connect('dbname=Melange', 'user=tempsql')
```

Naturally, this command will only work if both variables are accurate: there must be a real database named 'Melange' to which a user named 'tempsql' has access. If either of these conditions are not filled, Python will throw an error.

Next, Python likes to be able to keep track of where it last left off in reading and writing to the database. In psycopg, this is called the cursor, but we will use the variable 'mark' for our program. So, we can then construct the following assignment:

```
mark = connection.cursor()
```

Forming the PostgreSQL SELECT Statement

Now we can define the statement we would have executed. Since the variables are already defined at runtime (i.e., when the program is executed), we can create the statement and plug the variables in like concatenating a string.

```
statement = 'SELECT * FROM ' + table + ' WHERE ' + column + ' ' + operator + ' ' + string
```

Do note that this statement will work for any value. When using this statement one must supply the quotes for any character strings. However, if one wants to match character strings alone, not allowing for numerical calculations, one may supply the quotes within the statement itself.

```
statement = 'SELECT * FROM ' + table + ' WHERE ' + column + ' ' + operator + ' \'' + string +\ '\''
```

If you use this statement instead of the previous one, it is a good idea to evaluate the variable operator and to reject any numerical operators. Otherwise, PostgreSQL, and therefore both psycopg and Python, will throw an error.

Executing the SELECT Statement Through Psycopg

We need to tell psycopg to pass the statement to PostgreSQL. We do this by use the method 'execute', a method of connection.cursor(). The next statement thus looks like this:

```
mark.execute(statement)
```

The data returned will be an array of lists, one list for every line returned.
Given how flexible Python tends to be, one might ask why we define statement separately instead of passing its contents directly to execute. By defining statement separately, one is able to debug the program with fewer complications by simply inserting a print command at the appropriate point.

Consider, for example: PostgreSQL keeps throwing an error (e.g., "ERROR: syntax error at or near "' "]). You look at the SELECT statement a thousand times and still cannot figure out what is off. If you have embedded the statement, you have no way of printing the argument and thereby seeing the SELECT statement from the computer's perspective. If you define it separately, you can print it and better grasp where things are awry. In this way, form is kept separate from function.

A Container for the Results

Now, after executing the statement, we need a container into which psycopg can pour the results. We shall call this 'records'. One assigns the results to records using the 'fetchall' method as follows:

```
record = mark.fetchall()
```

'record' is an array holding the lists returned by execute. [Note that, if one merely wants the first hit, one can use the method 'fetchone' similarly.]
Having dumped the results into an array, we now need to extract them in an orderly fashion. The simplest and neatest way of doing this is with a 'for' loop:

```
for i in record:

     print i
```

Note that I say 'neatest' from the perspective of programming, not of viewing the data. One naturally programs for functionality, but one must always keep in mind the person who will need to read, modify, or debug your program in six months or a year from its creation and avoid spaghetti code.

This loop will return the records in list form. The user will probably not like this format, but I leave it to you to configure the output according to their needs.

Finishing and Executing the Program

we should return Python's attention from the main() function and finish off the program.

```
if __name__ == '__main__':

     main()
```

As usual, the last loop evaluates the runtime command from the user and passes Python's attention to main().
One can now call the program with the necessary four arguments.

python ./readpostgresql.py cornucopia id > 0 The results are tasty:

(1, 'banana')
(2, 'kiwi')
(3, 'nectarine')

Using this knowledge, you can easily automate access to several PostgreSQL databases and collate the data for any purpose.

Psycopg: Install and Import

The module we will use here is psycopg. It is available from http://www.initd.org/projects/psycopg1. So download it and install it using the directions that come with the package.

Once it is installed, you can import it like any other module:

```
# libs for database interface

import psycopg
```

If any of your fields require a date or time, you will also want to import the datetime module, which comes standard with Python.

```
import datetime
```

Python to PostgreSQL: Open Sesame

To open a connection to a database, psycopg needs two arguments: the name of the database ('dbname') and the name of the user ('user'). The syntax for opening a connection follows this format:

```
<variable name for connection> = psycopg.connect('dbname=<dbname>', 'user=<user>')
```

For our database, we shall use the database name 'Birds' and the username 'robert'. For the connection object within the program, let's use the variable 'connection'. So, our connection command will read as follows:

```
connection = psycopg.connect('dbname=Birds', 'user=robert')
```

Naturally, this command will only work if both variables are accurate: there must be a real database named 'Birds' to which a user named 'robert' has access. If either of these conditions are not filled, Python will throw an error.

Mark Your Place in PostgreSQL With Python

Python likes to be able to keep track of where it last left off in reading and writing to the database. In psycopg, this is called the cursor, but we will use the variable 'mark' for our program. So, we can then construct the following assignment:

```
mark = connection.cursor()
```

Separating PostgreSQL Form and Python Function

While some SQL insertion formats allow for understood or unstated column structure, we will be using the following template for our insert statements:

```
INSERT INTO <table> (columns) VALUES (values) ;
```

While we could pass a statement in this format to the psycopg method 'execute' and so insert data into the database, this quickly becomes convoluted and confusing. A better way is to compartmentalize the statement separately from the 'execute' command as follows:

```
statement = 'INSERT INTO ' + table + ' (' + columns + ') VALUES (' + values + ')'

mark.execute(statement)
```

In this way, form is kept separate from function. Such separation often helps in debugging.

Python, PostgreSQL, and the 'C' Word

After passing the data to PostgreSQL, we must commit the data to the database:

```
connection.commit()
```

Now we have constructed the basic parts of our function 'insert'. Put together, the parts look like this:

```
connection = psycopg.connect('dbname=Birds', 'user=robert')

mark = connection.cursor()

statement = 'INSERT INTO ' + table + ' (' + columns + ') VALUES (' + values + ')'

mark.execute(statement)

connection.commit()
```

Define the Parameters

You will notice that we have three variables in our statement: table, columns, and values. These thus become the parameters with which the function is called:

```
def insert(table, columns, values):
```

We should, of course, follow that with a doc string:

```
'''Function to insert the form data 'values' into table 'table'

according to the columns in 'column' '''
```

Put it All Together And Call It

We have a function for inserting data into a table of our choice, using columns and values defined as needed.

```
def insert(table, columns, values):

      '''Function to insert the form data 'values' into table 'table' according to the columns in 'column'
'''


      connection = psycopg.connect('dbname=Birds', 'user=robert')

      mark = connection.cursor()

      statement = 'INSERT INTO ' + table + ' (' + columns + ') VALUES (' + values + ')'

      mark.execute(statement)

      connection.commit()

      return
```

To call this function, we simply need to define the table, columns, and values and pass them as follows:

```
type = "Owls"

fields = "id, kind, date"

values = "17965, Barn owl, 2006-07-16"
```

insert(type, fields, values)

Python CGI First Steps: Importing cgi and cgitb

To use a Python program as a CGI script, you must import the cgi module. For purposes of troubleshooting your script, you should also import the module cgitb. This provides alternative exception handlers which offer more informative error messages in the event of an exception. While you can write CGI scripts without cgitb, it is recommended for its helpful exception handling.

If you have trouble running your CGI scripts, remember to check your permissions and the setup of your Apache server. If you do not have administrator access to your web server, check with someone who does.

Accessing CGI Form Data

CGI passes information to Python in the form of a class called FieldStorage. In your Pythonic CGI script, you must create an instance of FieldStorage in order to access the CGI data. The main attribute of class FieldStorage which you will need to know is 'getvalue'. For example, the data from an address book form might be accessed as follows:

```
# Import modules for CGI handling


import cgi, cgitb


# Create instance of FieldStorage

form = cgi.FieldStorage()


# Get data from field 'name'

name = form.getvalue('name')


# Get data from field 'address'
```

```
address = form.getvalue('address')


# Get data from field 'phone'

phone = form.getvalue('phone')


# Get data from field 'email'

email = form.getvalue('email')
```

From here, the data has been assigned to the variables. You can thus handle the values within the program like other literals.

Attributes of FieldStorage

Every instance of class FieldStorage (e.g., 'form') has the following attributes:

form.name The name of the field, if it is specified

form.filename If an FTP transaction, the client-side filename

form.value The value of the field as a string

form.file file object from which data can be read

form.type The content type, if applicable

form.type_options The options of the 'content-type' line of the HTTP request, returned as a dictionary

form.disposition The field 'content-disposition'; None if unspecified

form.disposition_options The options for 'content-disposition'

form.headers All of the HTTP headers returned as a dictionary

So, instead of writing above

```
name = form.getvalue('name')
```

I could have written

```
name = form['name'].value
```

Other Methods of Accessing Data

In addition to 'getvalue', the following methods may be used to access form data by field name:

- form.getfirst Returns the first value found for a field by the given name
- form.getlist Returns a list of all values found for a field by the given name. If no values are found, it returns an empty list.

In a case where each field name is unique, all three methods will return the same results. So the following will return the same data:

```
name = form.getvalue('name')

name = form.getfirst('name')

name = form.getlist('name')
```

The method 'getlist', however, will return a list of values. The others will simply return a string.

Calendar In Python

Python's calendar module is part of the standard library. It allows the output of a calendar by month or by year and also provides other, calendar-related functionality.

The calendar module itself depends on the datetime module. But we will also need datetime for our own purposes, as we will see. So we should import both of these. Also, in order to do some string splitting, we will need the re module. Let's import them all in one go.

```
import re, datetime, calendar
```

By default, the calendars begin the week with Monday (day 0), per the European convention, and ends with Sunday (day 6). If you prefer Sunday as the first day of the week, use the **setfirstweekday()** method to change the default to day 6 as follows:

```
calendar.setfirstweekday(6)
```

To toggle between the two, you could pass the first day of the week as an argument using the sys module. You would then check the value with an if statement and set the **setfirstweekday()** method accordingly.

```
import sys

firstday = sys.argv[1]

if firstday == "6":

calendar.setfirstweekday(6)
```

Preparing the Months of the Year

In our calendar, it would be nice to have a header for the calendar that reads something like "A Python-Generated Calendar For..." and have the current month and year. In order to do this, we need to get the month and year from the system. This functionality is something that calendar provides, Python can retrieve the month and year. But we still have a problem. As all system dates are numeric and do not contain unabbreviated or non-numeric forms of the months, we need a list of those months. Enter the list year.

```
year = ['January',

'February',

'March',

'April',

'May',

'June',

'July',

'August',

'September',

'October',

'November',

'December']
```

Now when we get the number of a month, we can access that number (minus one) in the list and get the full month name.

A Day Called "Today"

Starting the main() function, let's ask datetime for the time.

```
def main():

     today = datetime.datetime.date(datetime.datetime.now())
```

Curiously, the datetime module has a datetime class. It is from this class that we call two objects: now() and date(). The method datetime.datetime.now() returns an object containing the following information: year, month, date, hour, minute, second, and microseconds. Of course, we have no need for the time information. To cull out the date information alone, we pass the results of now() to datetime.datetime.date() as an argument. The result is that today now contains the year, month, and date separated by em-dashes.

Splitting the Current Date

To break this bit of data into more managable pieces, we must split it. We can then assign the parts to the variables current_yr, current_month, and current_day respectively.

```
current = re.split('-', str(today))

current_no = int(current[1])

current_month = year[current_no-1]

current_day = int(re.sub('\A0', '', current[2]))

current_yr = int(current[0])
```

To understand the first line of this code, work from the right to the left and from the inside outward. First, we stringify the object today in order to operate on it as a string. Then, we split it using the em-dash as a delimiter, or token. Finally, we assign those three values as a list to 'current'.

In order to deal with these values more distinctly and to call the long name of the current month out of year, we assign the number of the month to current_no. We can then do a bit of subtraction in the subscript of year and assign the month name to current_month.

In the next line, a bit of substitution is needed. The date which is returned from datetime is a two-digit value even for the first nine days of the month. A zero functions as a place holder, but we would rather our calendar have just the single digit. So we substitute no value for every zero that begins a string (hence '\A'). Finally, we assign the year to current_yr, converting it to an integer along the way.

Methods that we will call later will require input in integer format. Therefore, it is important to ensure that all of the date data is saved in integer, not string, form.

The HTML and CSS Preamble

Before we print the calendar, we need to print the HTML preamble and CSS layout for our calendar. Go to this page for the code to print the CSS and HTML preamble for the calendar. and copy the code into your program file. The CSS in the HTML of this file follows the template offered by Jennifer Kyrnin, About's Guide to Web Design. If you do not understand this part of the code, you may want to consult her helps for learning CSS and HTML. Finally, to customise the month name, we need the following line:

```
print '<h1> %s %s </h1 >' %(current_month, current_yr)
```

Printing the Days of the Week

Now that the basic layout is output, we can set up the calendar itself. A calendar, at its most basic point, is a table. So let's make a table in our HTML:

```
print '''

<table id="month" >

<thead >

<tr >

<th class="weekend" >Sunday</th >

<th >Monday</th >

<th >Tuesday</th >

<th >Wednesday</th >
```

```
<th >Thursday</th >

<th >Friday</th >

<th class="weekend" >Saturday</th >

</tr >

</thead >

<tbody >

'''
```

Now our program will print our desired header with the current month and year. If you have used the command-line option mentioned earlier, here you should insert an if-else statement as follows:

```
if firstday == '0':


     print '''

     <table id="month" >

     <thead >

     <tr >

     <th >Monday</th >

     <th >Tuesday</th >

     <th >Wednesday</th >

     <th >Thursday</th >

     <th >Friday</th >

     <th class="weekend" >Saturday</th >

     <th class="weekend" >Sunday</th >

     </tr >

     </thead >

     <tbody >

      '''

else: ## Here we assume a binary switch, a decision between '0' or not '0'; therefore, any non-zero argument
will cause the calendar to start on Sunday.

     print '''

     <table id="month" >

     <thead >

     <tr >

     <th class="weekend" >Sunday</th >

     <th >Monday</th >

     <th >Tuesday</th >

     <th >Wednesday</th >

     <th >Thursday</th >

     <th >Friday</th >

     <th class="weekend" >Saturday</th >

     </tr >

     </thead >

     <tbody >

     '''
```

Getting the Calendar Data

Now we need to create the actual calendar. To get the actual calendar data, we need the calendar module's monthcalendar() method. This method takes two arguments: the year and the month of the desired calendar (both in integer form). It returns a list which contains lists of the dates of the month by week. So if we count the number of items in the returned value, we have the number of weeks in the given month.

```
month = calendar.monthcalendar(current_yr, current_no)

nweeks = len(month)
```

The Number of Weeks In A Month

Knowing the number of weeks in the month, we can create a for loop which counts through a range() from 0 to the number of weeks. As it does, it will print out the rest of the calendar.

```
for w in range(0,nweeks):

     week = month[w]

     print "<tr>"

     for x in xrange(0,7):

          day = week[x]

          if x == 5 or x == 6:

               classtype = 'weekend'

          else:

               classtype = 'day'


           if day == 0:

                classtype = 'previous'

                print '<td class="%s"></td>' %(classtype)

           elif day == current_day:

                print '<td class="%s"><strong>%s</strong></span><div class="%s"></div></td>' %(classtype,
day, classtype)

           else:

                print '<td class="%s">%s</span><div class="%s"></div></td>' %(classtype, day, classtype)

                print "</tr>"


           print ''' </tbody>

</table>

</div>

</body>

</html>'''
```

We will discuss this code line-by-line on the next page.

The 'for' Loop Examined

After this range has been started, the dates of the week are culled from month according to the value of the counter and assigned to week. Then, a tabular row is created to hold the calendar dates.

A for loop then walks through the days of the week so they can be analyzed. The calendar module prints a '0' for every date in the table that does not have a valid value. A blank value would work better for our purposes so we print the bookends of tabular data without a value for those dates.

Next, if the day is the current one, we should highlight it somehow. Based on the td class today, the CSS of this page will cause the current date to be rendered against a dark background instead of the light background of the other dates.

Finally, if the date is a valid value and is not the current date, it is printed as tabular data. The exact color combinations for these are held in the CSS style preamble.

The last line of the first for loop closes the row. With the calendar printed our task is finished and we can close the HTML document.

```
print "</tbody></table></body></html>"
```

Calling the main() Function

As all of this code is in the main() function, do not forget to call it.

```
if __name__ == "__main__":

main()
```

Just this simple calendar can be used in any way that needs a calendar representation. By hyperlinking the dates in the HTML, one can easily create a diary functionality. Alternatively, one can check against a diary file and then reflect which dates are taken by their color. Or, if one converts this program into a CGI script, one can have it generated on the fly.

Network Programming is not Voodoo

For many people, network programming is a black art, with network programmers being the witch-doctors of the Web. As with most things that relate to computers, it really is easy once you know a little bit about it. This tutorial will help you grasp the basics of client operations by building a simple web client.

Programming Networks in Python: the Basics

All network transactions happen between clients and servers. In most protocols , the clients ask a certain address and receive data. The servers watch a port and give information.

To affect a network connection you need to know the host, the port, and the actions allowed on that port. Each port is associated with a service. Each server watches a different port.

Most web servers run on port 80, though it may sometimes be 8080. FTP lives on port 21, and secure shell (SSH) is on 22. For email, POP, SMTP, and IMAP all live on different ports.

You should note that these addresses are the common port numbers for the different services. A network administrator can change them for his or her network. As long as the client asks for the correct service on the right port at the right address, communication will still happen. Google's mail service, for example, does not run on the common port numbers but, because they know how to access their accounts, users can still get their mail.

Importing Modules for Network Programming in Python

As usual, our program will be more flexible if we assign values dynamically instead of hard-coding values. Therefore, let's import the sys module so we can grab input from the command line.

Next, import the socket module. This is the bedrock of most network programming in Python. While different modules exist for the various protocols, the socket module allows you to access any port on any machine and read or write to it. Other modules are certainly more appropriate for their given tasks (e.g., httplib, ftplib, gopher, poplib, etc.), but socket is foundational to each of the others (the httplib module, for example, imports socket).

Next, we need to declare a few variables.

Giving Python the Internet Protocol Information

As I mentioned earlier, every network client needs to know the address of the machine, the port of the service, and the name of the file on which it is to operate. Theoretically, we could take the port number from the command line. However, because the port usually determines the service, it is safer to hardwire the port (and therefore the service) into the program. For a web service, we will look on port 80.

```
import port = 80
```

Next, let's take the server's address and the name of the file from the command line. Because we are working on the Web, we can make us of the DNS and allow for URLs.

```
host = sys.argv[1]

filename = sys.argv[2]
```

Creating a Socket With Python

In order to access the Internet, we need to create a socket. The syntax for this call is as follows:

```
<variable> = socket.socket(<family>, <type>) </blockquote>
```

The recognised socket families are:

- AF_INET: IPv4 protocols (both TCP and UDP)
- AF_INET6: IPv6 protocols (both TCP and UDP)
- AF_UNIX: UNIX domain protocols

The first two are obviously internet protocols. Anything that travels over the internet can be accessed in these families. Many networks still do not run on IPv6. So, unless you know otherwise, it is safest to default to IPv4 and use AF_INET.

The socket type refers to the type of communication used through the socket. The five socket types are as follows:

- SOCK_STREAM: a connection-oriented, TCP byte stream
- SOCK_DGRAM: UDP transferral of datagrams (self-contained IP packets that do not rely on client-server confirmation)
- SOCK_RAW: a raw socket
- SOCK_RDM: for reliable datagrams
- SOCK_SEQPACKET: sequential transfer of records over a connection

By far, the most common types are SOCK_STEAM and SOCK_DGRAM because they function on the two protocols of the IP suite. The latter three are much rarer and so may not always be supported.

Let's therefore create a socket and assign it to variable; here I use c (for connection).

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

Connecting Sockets With Python

After creating the socket, we need to connect to it using the connect method of the socket object. The socket is essentially an opening in the networking capacity of the computer. In connecting, we give it a host and port for network communication.

```
c.connect((host, port))
```

It is worth noting that the address for the connection is given as a tuple (hence the double parentheses).
Before we can read from the socket, however, we need to make a file-like object from it. Remember, Python read and writes to file-like objects, not sockets. So, we essentially tell Python to view the socket as a file. All sockets have a function makefile that takes two arguments: the mode and the buffer size. In our case, we want a simple read mode that has no buffer.

```
fileobj = c.makefile('r', 0)
```

Python Asks the Web Server for the File

Now we get to communicate with the server. In retrieving information from a web server, web browsers send their requests in the following format:

```
<type of request> <file name> <protocol to use>
```

Our web client will ask the server to "GET" a given file using the HTTP 1.0 protocol. To communcate this, we write to the file-like object of the socket instance:

```
fileobj.write("GET "+filename+" HTTP/1.0\n\n")
```

The server will then respond with the data, and that data will be buffered as a file-like object. But our program will not get it unless we read it. So read the file object into another variable, here called buff.

```
buff = fileobj.readlines()
```

Printing the Web Page With Python

Now, all of the data is contained within buff. All we need to do is step through it, printing as we go.

```
for line in buff: print line
```

Now, you can save the program and call it with the name of the server and the name of the file you want to see. For example, try:

```
python simple_web_client.py f2finterview.com /
```

If you run your own server, you can access your web directory as follows:

python simple_web_client.py localhost / What you will receive is the raw HTML of the index file. You can then process it as plain text or parse it with **urllib** or **urllib2**.

The Python Code for a Simple Web Client Program

To ensure that you have all the lines needed for this program, here is the code for this web client.

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


# import sys for handling command line argument

# import socket for network communications

import sys, socket
```

```
# hard-wire the port number for safety's sake
# then take the names of the host and file from the command line
port = 80
host = sys.argv[1]
filename = sys.argv[2]

# create a socket object called 'c'
c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect to the socket
c.connect((host, port))

# create a file-like object to read
fileobj = c.makefile('r', 0)

# Ask the server for the file
fileobj.write("GET "+filename+" HTTP/1.0\n\n")

# read the lines of the file object into a buffer, buff
buff = fileobj.readlines()

# step through the buffer, printing each line
for line in buff:
print line
```

Simple Web Server in Python

As a complement to the network client tutorial, this tutorial shows how to implement a simple web server in Python. To be sure, this is no substitute for Apache or Zope. There are also more robust ways to implement web services in Python, using modules like BaseHTTPServer. This server uses the socket module exclusively.

You will recall that the socket module is the backbone of most Python web service modules. As with the simple network client, building a server with it illustrates the basics of web services in Python transparently. BaseHTTPServer itself imports the socket module to affect a server.

Running Servers

By way of review, All network transactions happen between clients and servers. In most protocols , the clients ask a certain address and receive data.

Within each address, a multitude of servers can run. The limit is in the hardware. With sufficient hardware (RAM, processor speed, etc.), the same computer can serve as a web server, an ftp server, and mail server (pop, smtp, imap, or all of the above) all at the same time. Each service is associeted with a port. The port is bound to a socket. The server listens to its associated port and gives information when requests are received on that port.

Communicating Via Sockets

So to affect a network connection you need to know the host, the port, and the actions allowed on that port. Most web servers run on port 80. However, in order to avoid conflict with an installed Apache server, our web server will run on port 8080. In order to avoid conflict with other services, it is best to keep HTTP services on port 80 or 8080. These are the two most common. Obviously, if these are used, you must find an open port and alert users to the change.

As with the network client, you should note that these addresses are the common port numbers for the different services. As long as the client asks for the correct service on the right port at the right address, communication will still happen. Google's mail service, for example, did not initially run on the common port numbers but, because they know how to access their accounts, users can still get their mail.

Unlike the network client, all variables in the server are hardwired. Any service that is expected to run constantly should not have the variables of its internal logic set at the command line. The only variation on this would be if, for some reason, you wanted the service to run occasionally and on various port numbers. If this were the case, however, you would still be able to watch the system time and change bindings accordingly.

So our sole import is the socket module.

```
import socket
```

Next, we need to declare a few variables.

Hosts and Ports

As already mentioned, the server needs to know the host to which it is to be associated and the port on which to listen. For our purposes, we shall have the service apply to any host name at all.

```
host = ''

port = 8080
```

The port, as mentioned earlier, will be 8080. So note that, if you use this server in conjunction with the network client, you will need to change the port number used in that program.

Creating a Socket

Whether to request information or to serve it, in order to access the Internet, we need to create a socket. The syntax for this call is as follows:

```
<variable> = socket.socket(<family>, <type>)
```

The recognised socket families are:

- AF_INET: IPv4 protocols (both TCP and UDP)
- AF_INET6: IPv6 protocols (both TCP and UDP)
- AF_UNIX: UNIX domain protocols

The first two are obviously internet protocols. Anything that travels over the internet can be accessed in these families. Many networks still do not run on IPv6. So, unless you know otherwise, it is safest to default to IPv4 and use AF_INET.

The socket type refers to the type of communication used through the socket. The five socket types are as follows:

- SOCK_STREAM: a connection-oriented, TCP byte stream
- SOCK_DGRAM: UDP transferral of datagrams (self-contained IP packets that do not rely on client-server confirmation)
- SOCK_RAW: a raw socket
- SOCK_RDM: for reliable datagrams
- SOCK_SEQPACKET: sequential transfer of records over a connection

By far, the most common types are SOCK_STEAM and SOCK_DGRAM because they function on the two protocols of the IP suite (TCP and UDP). The latter three are much rarer and so may not always be supported.

So let's create a socket and assign it to a variable.

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

Setting Socket Options

After creating the socket, we then need to set the socket options. For any socket object, you can set the socket options by using the setsockopt() method. The syntax is as follows:

socket_object.setsockopt(level, option_name, value) For our purposes, we use the following line:

```
c.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

The term 'level' refers to the categories of options. For socket-level options, use SOL_SOCKET. For protocol numbers, one would use IPPROTO_IP. SOL_SOCKET is a constant attribute of the socket. Exactly which options are available as part of each level are determined by your operating system and whether you are using IPv4 or IPv6.

The documentation for Linux and related Unix systems can be found in the system documentation. The documentation for Microsoft users can be found on the MSDN website. As of this writing, I have not found Mac documentation on socket programming. As Mac is roughly based upon BSD Unix, it is likely to implement a full complement of options.

In order to ensure reusability of this socket, we use the SO_REUSEADDR option. One could restrict the server to only run on open ports, but that seems unnecessary. Do note, however, that if two or more services are deployed on the same port, the effects are unpredictable. One cannot be certain which service will receive which packet of information.

Finally, the '1' for a value is the value by which the request on the socket is known in the program. In this way, a program can listen on a socket in very nuanced ways.'

Binding the Port to the Socket

After creating the socket and setting its options, we need to bind the port to the socket.

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

The binding done, we now tell the computer to wait and to listen on that port.

```
c.listen(1)
```

If we want to give feedback to the person who calls the server, we could now enter a print command to confirm that the server is up and running.

Handling a Server Request

Having setup the server, we now need to tell Python what to do when a request is made on the given port. For this we reference the request by its value and use it as the argument of a persistent while loop.

When a request is made, the server should accept the request and create a file object to interact with it.

```
while 1:

     csock, caddr = c.accept()

     cfile = csock.makefile('rw', 0)
```

In this case, the server uses the same port for reading and writing. Therefore, the makefile method is given an argument 'rw'. The null length of the buffer size simply leaves that part of the file to be determined dynamically.

Sending Data to the Client

Unless we want to create a single-action server, the next step is to read input from the file object. When we do that, we should be careful to strip that input of excess whitespace.

```
line = cfile.readline().strip()
```

The request will come in the form of an action, followed by a page, the protocol, and the version of the protocol being used. If one wants to serve a web page, one splits this input to retrieve the page requested and then reads that page into a variable which is then written to the socket file object. A function for reading a file into a dictionary can be found in the blog.

In order to make this tutorial a bit more illustrative of what one can do with the socket module, we will forego that part of the server and instead show how one can nuance the presentation of data. Enter the next several lines into the program.

```
cfile.write('HTTP/1.0 200 OK\n\n')

cfile.write('<html><head><title>Welcome %s!</title></head>' %(str(caddr)))

cfile.write('<body><h1>Follow the link...</h1>')

cfile.write('All the server needs to do is ')

cfile.write('to deliver the text to the socket. ')

cfile.write('It delivers the HTML code for a link, ')

cfile.write('and the web browser converts it. <br><br><br><br>')

cfile.write('<font size="7"><center> <a href="http://python.about.com/index.html">Click me!</a>
</center></font>')

cfile.write('<br><br>The wording of your request was: "%s"' %(line))

cfile.write('</body></html>')
```

Final Analysis and Shutting Down

If one is sending a web page, the first line is a nice way of introducing the data to a web browser. If it is left out, most web browsers will default to rendering HTML. However, if one includes it, the 'OK' must be followed by two new line characters. These are used to distinguish the protocol information from the page content.

The syntax of the first line, as you can probably surmise, is protocol, protocol version, message number, and status. If you have ever gone to a web page that has moved, you have probably received a 404 error. The 200 message here is simply the affirmative message.

The rest of the output is simply a web page broken up over several lines. You will note that the server can be programmed to use user data in the output. The final line reflects the web request as it was received by the server.

Finally, as the closing acts of the request, we need to close the file object and the server socket.

```
cfile.close()

csock.close()
```

Now save this program under a recognisable name. After you call it with 'python program_name.py', if you programmed a message to confirm the service as running, this should print to the screen. The terminal will then seem to pause. All is as it should be. Open your web browser and go to localhost:8080. You should then see the output of the write commands we gave. Please note that, for the sake of space, I did not implement error handling in this program. However, any program released into the 'wild' should. See "Error Handling in Python" for more.

Name five modules that are included in python by default (many people come searching for this, so some more examples of modules which are often used are included)

datetime	(used to manipulate date and time)
re	(regular expressions)
urllib, urllib2	(handles many HTTP things)
string	(a collection of different groups of strings for example all lower_case letters etc)
itertools	(permutations, combinations and other useful iterables)
ctypes	(from python docs: create and manipulate C data types in Python)
email	(from python docs: A package for parsing, handling, and generating email messages)
__future__	(Record of incompatible language changes. like division operator is different and much better when imported from __future__)
sqlite3	(handles database of SQLite type)
unittest	(from python docs: Python unit testing framework, based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework)
xml	(xml support)
logging	(defines logger classes. enables python to log details on severity level basis)
os	(operating system support)
pickle	(similar to json. can put any data structure to external files)
subprocess	(from docs: This module allows you to spawn processes, connect to their input/output/error pipes, and obtain their return codes)
webbrowser	(from docs: Interfaces for launching and remotely controlling Web browsers.)
traceback	(Extract, format and print Python stack traces)

Name a module that is not included in python by default

mechanize

django

gtk

A lot of other can be found at pypi.

What is __init__.py used for?

It declares that the given directory is a module package. #Python Docs (From Endophage's comment)

When is pass used for?

pass does nothing. It is used for completing the code where we need something. For eg:

```
class abc():

    pass
```

What is a docstring?

docstring is the documentation string for a function. It can be accessed by

function_name.__doc__

it is declared as:

```
def function_name():

"""your docstring"""
```

Writing documentation for your progams is a good habit and makes the code more understandable and reusable.

What is list comprehension?

Creating a list by doing some operation over data that can be accessed using an iterator. For eg:

```
>>>[ord(i) for i in string.ascii_uppercase]

     [65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90]

 >>>
```

What is map?

map executes the function given as the first argument on all the elements of the iterable given as the second argument. If the function given takes in more than 1 arguments, then many iterables are given. #Follow the link to know more similar functions
For eg:

```
>>>a='ayush'

>>>map(ord,a)

.... [97, 121, 117, 115, 104]

>>> print map(lambda x, y: x*y**2, [1, 2, 3], [2, 4, 1])

.... [4, 32, 3]


Help on built-in function map in module __builtin__:


map(...)

map(function, sequence[, sequence, ...]) -> list


Return a list of the results of applying the function to the items of

the argument sequence(s).&nbsp; If more than one sequence is given, the

function is called with an argument list consisting of the corresponding

item of each sequence, substituting None for missing values when not all

sequences have the same length.&nbsp; If the function is None, return a list of

the items of the sequence (or a list of tuples if more than one sequence).
```

What is the difference between a tuple and a list?

A tuple is immutable i.e. can not be changed. It can be operated on only. But a list is mutable. Changes can be done internally to it.

```
tuple initialization: a = (2,4,5)

list initialization: a = [2,4,5]
```

The methods/functions provided with each types are also different. Check them out yourself.

Using various python modules convert the list a to generate the output 'one, two, three'

```
a = ['one', 'two', 'three']

Ans:&nbsp;&nbsp; ", ".join(a)
```

```
>>>help(str.join)
Help on method_descriptor:
 join(...)
 S.join(iterable) -> string
 Return a string which is the concatenation of the strings in the
 iterable.&nbsp; The separator between elements is S.
```

What would the following code yield?

word = 'abcdefghij'
print word[:3] + word[3:]

'abcdefghij' will be printed.
This is called string slicing. Since here the indices of the two slices are colliding, the string slices are 'abc' and 'defghij'. The '+' operator on strings concatenates them. Thus, the two slices formed are concatenated to give the answer 'abcdefghij'.

Optimize these statements as a python programmer.

word = 'word'
print word.__len__()

```
word = 'word'
print len(word)
```

Write a program to print all the contents of a file

```
try:
    with open('filename','r') as f:
        print f.read()
except IOError:
    print "no such file exists"
```

What will be the output of the following code

a = 1
a, b = a+1, a+1
print a
print b

```
2
2
```

The second line is a simultaneous declaration i.e. value of new a is not used when doing b=a+1.

This is why, exchanging numbers is as easy as:

a,b = b,a

Given the list below remove the repetition of an element.

All the elements should be unique
words = ['one', 'one', 'two', 'three', 'three', 'two']

A bad solution would be to iterate over the list and checking for copies somehow and then remove them!

One of the best solutions I can think of right now:

```
a = [1,2,2,3]

list(set(a))
```

set is another type available in python, where copies are not allowed. It also has some good functions available used in set operations (like union, difference).

Iterate over a list of words and use a dictionary to keep track of the frequency(count) of each word. for example

{‘one’:2, ‘two’:2, ‘three’:2}

```
>>> def dic(words):

  a = {}

  for i in words:

    try:

      a[i] += 1

    except KeyError: ## the famous pythonic way:

      a[i] = 1&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; ## Halt and catch fire

  return a


>>> a='1,3,2,4,5,3,2,1,4,3,2'.split(',')

>>> a

['1', '3', '2', '4', '5', '3', '2', '1', '4', '3', '2']

>>> dic(a)

{'1': 2, '3': 3, '2': 3, '5': 1, '4': 2}
```

Without using try-catch block:

```
>>> def dic(words):

  data = {}

  for i in words:

    data[i] = data.get(i, 0) + 1

  return data


>>> a

['1', '3', '2', '4', '5', '3', '2', '1', '4', '3', '2']
```

```
>>> dic(a)

{'1': 2, '3': 3, '2': 3, '5': 1, '4': 2}
```

PS: Since the collections module (which gives you the defaultdict) is written in python, I would not recommend using it. The normal dict implementation is in C, it should be much faster. You can use timeit module to check for comparing the two.
So, David and I have saved you the work to check it. Check the files on github. Change the data file to test different data.

Write the following logic in Python:

If a list of words is empty, then let the user know it's empty, otherwise let the user know it's not empty.

Can be checked by a single statement (pythonic beauty):

```
print "The list is empty" if len(a)==0 else "The list is not empty"


>>> a=''

>>> print "'The list is empty'" if len(a)==0 else "'The list is not empty'"

'The list is empty'

>>> a='asd'

>>> print "'The list is empty'" if len(a)==0 else "'The list is not empty'"

'The list is not empty'
```

Demonstrate the use of exception handling in python.

```
try:

  import mechanize as me

except ImportError:

  import urllib as me
```

## here you have atleast 1 module imported as me.
This is used to check if the users computer has third party libraries that we need. If not, we work with a default library of python. Quite useful in updating softwares.
PS: This is just one of the uses of try-except blocks. You can note a good use of these in API's.
Also note that if we do not define the error to be matched, the except block would catch any error raised in try block.

Print the length of each line in the file 'file.txt' not including any whitespaces at the end of the lines.

```
with open("filename.txt", "r") as f1:

  print len(f1.readline().rstrip())
```

rstrip() is an inbuilt function which strips the string from the right end of spaces or tabs (whitespace characters).

Print the sum of digits of numbers starting from 1 to 100 (inclusive of both)

```
print sum(range(1,101))
```

range() returns a list to the sum function containing all the numbers from 1 to 100. Please see that the range function does not include the end given (101 here).

```
print sum(xrange(1, 101))
```

xrange() returns an iterator rather than a list which is less heavy on the memory.

Create a new list that converts the following list of number strings to a list of numbers.

num_strings = ['1','21','53','84','50','66','7','38','9']

use a list comprehension

```
>>> [int(i) for i in num_strings]

[1, 21, 53, 84, 50, 66, 7, 38, 9]
```

#num_strings should not contain any non-integer character else ValueError would be raised. A try-catch block can be used to notify the user of this.

Another one suggested by David using maps:

```
>>> map(int, num_strings)

    [1, 21, 53, 84, 50, 66, 7, 38, 9]
```

Create two new lists one with odd numbers and other with even numbers

num_strings = [1,21,53,84,50,66,7,38,9]

```
>>> odd=[]

>>> even=[]

>>> for i in n:

    even.append(i) if i%2==0 else odd.append(i)


## all odd numbers in list odd

## all even numbers in list even
```

Though if only one of the lists were requires, using list comprehension we could make:

```
even = [i for i in num_strings if i%2==0]

odd = [i for i in num_strings if i%2==1]
```

But using this approach if both lists are required would not be efficient since this would iterate the list two times.!

Write a program to sort the following intergers in list

nums = [1,5,2,10,3,45,23,1,4,7,9]

nums.sort() # The lists have an inbuilt function, sort()
sorted(nums) # sorted() is one of the inbuilt functions)

Python uses TimSort for applying this function. Check the link to know more.

Write a for loop that prints all elements of a list and their position in the list.

Printing using String formatting

```
>>> for index, data in enumerate(asd):

.... print "{0} -> {1}".format(index, data)


0 -> 4

1 -> 7

2 -> 3

3 -> 2

4 -> 5

5 -> 9
```

OR

```
>>> asd = [4,7,3,2,5,9]


>>> for i in range(len(asd)):

.... print i+1,'-->',asd[i]


1 --> 4

2 --> 7

3 --> 3

4 --> 2

5 --> 5

6 --> 9
```

The following code is supposed to remove numbers less than 5 from list n, but there is a bug. Fix the bug.

```
n = [1,2,5,10,3,100,9,24]


for e in n:

  if e<5:

    n.remove(e)

  print n
```

after e is removed, the index position gets disturbed. Instead it should be:

```
a=[]

for e in n:

  if e >= 5:

    a.append(e)

n = a
```

OR again a list comprehension:

```
return [i for i in n if i >= 5]
```

What will be the output of the following

```
def func(x,*y,**z):
.... print z

func(1,2,3)
```

Here the output is :

{} #Empty Dictionay

x is a normal value, so it takes 1..

y is a list of numbers, so it takes 2,3..

z wants named parameters, so it can not take any value here.

Thus the given answer.

Write a program to swap two numbers.

```
a = 5
b = 9
```

as i told earlier too, just use:

```
a,b = b,a
```

What will be the output of the following code

```
class C(object):
.... def__init__(self):
.... self.x =1

c=C()
print c.x
print c.x
print c.x
print c.x
```

All the outputs will be 1, since the value of the the object's attribute(x) is never changed.

```
1
1
1
1
```

x is now a part of the public members of the class C.

Thus it can be accessed directly.

What is wrong with the code

func([1,2,3]) # explicitly passing in a list

func()&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; # using a default empty list

def func(n = []):

#do something with n

print n

This would result in a NameError. The variable n is local to function func and can't be accessesd outside. So, printing it won't be possible.

What all options will work?

```
a.

n = 1

print n++ ## no such operator in python (++)


b.

n = 1

print ++n ## no such operator in python (++)


c.

n = 1

print n += 1 ## will work


d.

int n = 1

print n = n+1 ##will not work as assignment can not be done in print command like this


e.

n =1

n = n+1 ## will work
```

In Python function parameters are passed by value or by reference?

By value (check if you want to, I also did the same. It is somewhat more complicated than I have written here (Thanks David for pointing). Explaining all here won't be possible. Some good links that would really make you understand how things are:

Stackoverflow

Python memory management

Viewing the memory

Remove the whitespaces from the string.

s = 'aaa bbb ccc ddd eee'

```
''.join(s.split())

## join without spaces the string after splitting it
```

What does the below mean?

```
s = a + '[' + b + ':' + c + ']'
```

seems like a string is being concatenated. Nothing much can be said without knowing types of variables a, b, c. Also, if all of the a, b, c are not of type string, TypeError would be raised. This is because of the string constants ('[' , ']') used in the statement.

Optimize the below code

```
def append_s(words):

  new_words=[]

  for word in words:

    new_words.append(word + 's')

  return new_words


for word in append_s(['a','b','c']):

  print word
```

The above code adds a trailing s after each element of the list.

```
def append_s(words):

     return [i+'s' for i in words] ## another list comprehension


for word in append_s(['a','b','c']):

     print word
```

If given the first and last names of bunch of employees how would you store it and what datatype?

best stored in a list of dictionaries..
dictionary format: {'first_name':'Ayush','last_name':'Goel'}

How to find distance of two locations using google map's latitude and longitude with python?

```
from math import sin,cos,atan,acos,asin,atan2,sqrt,pi, modf


def getDistance(loc1, loc2):

   "aliased default algorithm; args are (lat_decimal,lon_decimal) tuples"

   return getDistanceByHaversine(loc1, loc2)
```

```
def getDistanceByHaversine(loc1, loc2):
   "Haversine formula - give coordinates as (lat_decimal,lon_decimal) tuples"
   earthradius = 6371.0

   lat1, lon1 = loc1
   lat2, lon2 = loc2

   # convert to radians
   lon1 = lon1 * pi / 180.0
   lon2 = lon2 * pi / 180.0
   lat1 = lat1 * pi / 180.0
   lat2 = lat2 * pi / 180.0

   # haversine formula
   dlon = lon2 - lon1
   dlat = lat2 - lat1
   a = (sin(dlat/2))**2 + cos(lat1) * cos(lat2) * (sin(dlon/2.0))**2
   c = 2.0 * atan2(sqrt(a), sqrt(1.0-a))
   km = earthradius * c
   return km

location1 = (13.01013,80.21122)
location2 = (13.03361,80.26861)

distance = getDistance(location1, location2)

print 'KM', distance

#Output: in KM is 6.74331391328
```