Writing CGI Programs in Python

By Preston Landers

Table of Contents

What is Python and what can it do for me?

Python is a powerful, free, open source, general purpose, interpreted programming language. Python runs on a wide variety of platforms including Linux, Microsoft Windows (95/98/NT), Macintosh (including OS X), virtually every flavor of Unix, and many other platforms. Python is roughly comparable to Perl or Java, though it has several significant strengths (and a few disadvantages) over each. Python makes it very easy to write clean, maintainable, and powerful programs for a variety of tasks with minimum hassle.

This article is not a Python advocacy piece nor a general Python tutorial. There are already several excellent resources on the net for both tasks. In fact, one of the main joys of using Python is the well organized and ever useful Python web site. Few languages can boast such a wealth of clear and concise documentation gathered in one location.

For instance, you can go here for a summary of Python's features and comparisons to other popular languages. For a much more thorough introduction to general purpose programming in Python, please see the tutorial. Finally, you can obtain Python for your system here. (That is a generic download page and there may be easy to install packages available for your specific distribution or OS. See the end of this article.)

Why should my next CGI project be in Python?

First of all, it should be stated that Python is a general purpose programming language. Though it is well suited for almost every type of web application, if you need a few simple SSI (server−side include) pages a language like PHP might be more suitable.

Perl is well known as the "King of CGI." Perl is uniquely suited as a text scanning and processing language, and its CGI related modules are extensive and well implemented. However, even an old−time Perl hacker will usually tell you that Perl really shines in one−person 'quickie' jobs that will never have to be maintained by another human being. The combination of Perl's line−noise−like syntax and "There's More Than One Way To Do It" philosophy often results in an unmaintainable mess of a program.

That's not to say that interesting and powerful applications can't be written in Perl. In addition, knowledge of the language itself is often necessary for cleaning up other people's messes. However, if you are starting a brand new web application, you should consider Python and its "There Should Be One Obvious Way To Do It" philosophy.

Python has a much cleaner syntax than most languages and object orientation is built right into the core. [Overt use of object features such as classes is entirely optional; Python can be used as a straightforward scripting language.] Python's clean, readable syntax is a blessing to new developers learning the language and also to people who have to maintain other people's code.

People developing with the Apache web server (and according to Netcraft, that's most of you) will be pleased to know about the mod_python module that embeds the Python interpreter directly into Apache, resulting in application execution speeds to rival Perl.

Lastly, Python has a very extensive, well documented and portable module library that provides a large variety of useful functions. The Internet−related collection is particularly impressive, with modules (Python function/class libraries) to deal with everything from parsing and retrieving URL's to retrieving mail from POP servers and everything in between. Other modules handle everything from GUI interfaces (using a variety of popular toolkits) to database access.

Your First CGI program in Python

In this section, I'm going to assume you have Python installed and have set up your web server to actually execute Python scripts (instead of, for example, sending the source code to the client.) There are so many variations of operating systems and web servers that I'm going to have to rely on you following the appropriate directions that came with your copy of Python.

A tip for Unix users: often a CGI script has to have the executable attribute set ("chmod +x script.py") and/or have a special extension or be in a special CGI directory. Consult your friendly local web administrator for details.

Please note that if you're using Apache, using mod_python is not absolutely necessary but is recommended for most applications. It doesn't affect anything in your program other than the (perceived) execution speed because it eliminates the start−up time for the interpreter with each connection.

Here is a (very) simple script to test your setup:

#!/usr/bin/python # Tell the browser how to render the text print "Content−Type: text/plain\n\n" print "Hello, Python!" # print a test string

---

Let's go over the script line−by−line to familiarize ourselves with the basic components of a Python script.

The first line is necessary for most Linux/Unix systems, but is harmlessly useless on all other systems. It is simply a signal to the shell that This Is A Python Program. You will need to change the path if Python is not installed in the usual place. (Sometimes it is in /usr/local/bin/python for example.) Other systems, such as Windows or Macintosh, will have to follow different procedures to inform the OS to execute this as a Python file. Since the first line begins with a # character, it is a comment and has no actual effect in the Python program itself.

Next we have another comment describing what the first line of actual code does: simply print a string (to "standard output") describing what kind of content follows. This is necessary for most browsers to know how to display the information. In this case it's just plain text, but we might want 'Content−Type: text/html' in the future. The '\n' is a special character that says "print a new line." Normally the print command will automatically add a newline for you, but in this case we need two (because that's what browsers expect.) The signals a special character. Use to actually print a slash.

Finally, we simply say hello to the world! Note that Python statements do not end with a semicolon or other punctuation, just a newline.

If everything went according to plan, visiting the URL of the above script should print "Hello, Python!" to your browser window.

Getting some real work done

Our little example script above didn't accomplish much real work, of course, but it was helpful (I hope) in making sure your web server is set up to run Python CGI scripts. Now we're going to delve into the nitty−gritty of writing a real CGI program in Python. It might be helpful to have the Python tutorial open in another window to refer to from time to time on matters of syntax or keywords.

The first thing that most web developers will want to know is how to get information out of HTML forms and do something with it.

Python provides an excellent module to deal with this situation. Coincidentally enough, the module is named cgi. So let me show you a simple example program that will gather information out of a browser form.

As you know, the data in HTML forms is referenced by the "name" attribute of each input element. For instance, this tag in a form: <INPUT TYPE=TEXT NAME=email SIZE=40> produces a 40 character wide text box with word "email" associated with it.

This information is passed to a CGI script by either one of two methods: GET or POST. However, the cgi module hides this complexity from the programmer and presents all form information as a Python fundamental data type; the ever−useful dictionary. Perl users will know this one as a "hash" (which sounds like more fun than a dictionary, but the word is less descriptive.) A dictionary is a type of array that is indexed by a string instead of a number.

For instance, if we had defined a dictionary called Bestiary, we could type something like this:

```
>>> Bestiary["Zebra"]
```

and Python might respond with:

```
"A black and white striped ungulate native to southern Africa."
```

[The little snippet above demonstrates an interesting and occasionally useful feature of Python: the ability to act as an interactive interpreter. ">>>" is the Python prompt. If you type an expression, as I did above, Python will print the return value. If you're only writing CGI scripts, you probably won't use the interactive interpreter much, but sometimes it does come in handy for debugging.]

Python's cgi module has a method (another name for a procedure or function) called FieldStorage that returns a dictionary containing all the form <INPUT>'s and their corresponding values.

If you know the specific item you want, you can easily access it:

```
import cgi The_Form = cgi.FieldStorage() print The_Form["email"].value
```

Note that just using `The_Form["email"]` returns a tuple containing both the field name and the value. When dealing with the dictionaries returned by cgi.FieldStorage, we need to be explicit in asking for the values in the forms.

Any generic method that works on dictionaries can be usefully applied. For instance, if you want to find out which keys are available on any given form, you might do something like this:

```
>>> The_Form.keys() ['name', 'email', 'address', 'phone']
```

or this:

```
>>> The_Form.values() ['Preston Landers', 'preston@askpreston.com', '1234 Main St.',
'555−1212']
```

(Please note that Python will not guarantee any particular order for the `keys()` or `values()` methods. If you want that, use `sort()`)

If you try to access a field that doesn't exist, this will generate an exception; an error status. If you don't catch the exception with an except: block, this will stop your script and the user will see the dreaded "Internal Server Error." Don't worry too much about it at this point because we'll be covering exceptions in great detail later on.

We can put these pieces together to write a simple program that prints out the name and value of each <INPUT> element passed to it. In this case, it's up to you to provide an actual web page that contains the form that points to this CGI script as its ACTION field. If you're not sure what I mean, don't worry, we'll go over forms again.

```
#!/usr/bin/python import cgi print "Content−Type: text/plain\n\n" The_Form =
cgi.FieldStorage() for name in The_Form.keys(): print "Input: " + name + " value: " +
The_Form[name].value + "<BR>" print "Finished!"
```

As you can see, we used a simple for loop to iterate over each key, printing the key name, its value, and an HTML break to separate each line. Notice that blocks of code are set off by tabs in Python, not { } curlies like in C or Perl. When the indentation changes, as in the line that prints "Finished!," that signals the end of that code block.

We've also demonstrated the + operation for strings; as you might expect, + will concatenate (stick together) two strings. If you're not sure if the elements you're concatenating will be strings at runtime, use the str() or repr() methods. For instance:

```
>>> NumberA = 55 >>> NumberB = 45 >>> print NumberA + NumberB 100 >>> print
str(NumberA) + str(NumberB) 5545
```

Defining a useful Display function

Since we've decided to build our Python application incrementally ("Oh have we?" you ask) rather than from some kind of master scheme, let's go ahead and define a function that might actually be useful to us. By the time this article series is complete, you will see how and why it is best to design an overall object framework for all but the simplest applications, rather than design from the ground−up like we're doing here for the purposes of instruction.

We'll need a routine to actually display the content that we're going to assemble. This routine should take care of administrivia like the Content−Type line and basic HTML syntax like the <BODY></BODY> tags. Also, ideally, we want as little actual HTML in our Python program as possible. In fact, we want all this formatting stuff in an external file that can be easily modified and updated without delving into actual program code.

Your site probably has a standard HTML "style sheet" (even if you're not using actual CSS) that you're expected to follow. It would be a real pain to embed this HTML directly into your code, only to have to change on a daily basis. There is also the danger of accidentally modifying important program code.

So, we'll go ahead and define a simple template file that your Python scripts' Display() function will use. Your actual template will probably be much more complicated but this will get you started.

```
template.html:
<!DOCTYPE HTML PUBLIC "−//IETF//DTD HTML//EN">
<html>
  <head>
    <META NAME="keywords" CONTENT="blah blah −− your ad here">
    <title>Python is Fun!</title>
  </head>
  <body>
                <!−− *** INSERT CONTENT HERE *** −−>
  </body>
</html>
```

The thing that should (hopefully) strike you about this template is the HTML comment `<!−− *** INSERT CONTENT HERE *** −−>`. Your Python program will search for this comment and replace it with its goodies.

To perform this feat of minor magic, we will introduce a couple of new methods including file operations and regular expressions. Old Perl Hackers will rejoice to learn that Python includes complete and powerful regular expression (RE) support. The actual RE syntax is almost identical to Perl's, but it is **not** built directly into the language (it is implemented as a module.)

Those of you who are not Old Perl Hackers are probably wondering at this point what exactly are regular expressions. A regular expression is like a little mini−language that specifies a precise set of text strings that match, or agree, with it. That set can be unlimited or finite in size depending on the regexp. When Python scans a string for a given regular expression, it searches the entire string for characters that *exactly* match the RE. Then, depending on the operation, it will either simply note the location or perform a substitution. RE's are extraordinarily powerful in a wide range of applications, but we will only be using a few basic RE features here.

Please see the Regular Expression HOWTO for an excellent guide to using regular expressions in Python.

The basic tactic we're going to use to implement template support in our CGI script is to read the entire template file in as one long string, then do a regular expression substitution, swapping our content for that "INSERT CONTENT HERE" marker.

Here is the code for our `Display(Content)` function:

```
import re  # make the regular expression module available

# specify the filename of the template file
TemplateFile = "template.html"

# define a new function called Display
# it takes one parameter - a string to Display
def Display(Content):
    TemplateHandle = open(TemplateFile, "r")  # open in read only mode
    # read the entire file as a string
    TemplateInput = TemplateHandle.read()
    TemplateHandle.close()                    # close the file

    # this defines an exception string in case our
    # template file is messed up
    BadTemplateException = "There was a problem with the HTML template."

    SubResult = re.subn("<!-- *** INSERT CONTENT HERE *** -->",
        Content,TemplateInput)
    if SubResult[1] == 0:
        raise BadTemplateException

    print "Content-Type: text/html\n\n"
    print SubResult[0]
```

Let's review some of the important features of this code snippet. First, reading the contents of a file into a string is fairly unremarkable.

Next we define a string called BadTemplateException that will be used in case the template file is corrupt. If the marker HTML comment cannot be found, this string will be 'raised' as a fatal exception and stop the execution of the script. This exception is fatal because it's not caught by an 'except' block. Also, if the template file simply can't be found by the `open()` method, Python itself will raise an (unhandled, fatal) exception. All the user will likely see is an "Internal Server Error" message. The traceback (what caused the exception) will be recorded in the web server's logs. We'll go into more detail later about exceptions and how to handle them intelligently.

Next, we have the actual meat of this function. This line uses the `subn()` method of the `re` (regular expression) module to substitute our Content string for the marker defined in the first parameter to the method. Notice that we had to backslash the `*` characters, because these have special meaning in RE's. (See the documentation for the re module for a complete list of special characters.) Our RE is relatively simple; it is simply an exact list of characters with no special attributes.

The second parameter is our `Content`, the thing we want to replace the marker with. This could be another function as long as that function returns a string (otherwise, you'll get an exception.) The final parameter is the string to operate on, in this case, the entire template file contained in TemplateInput.

`re.subn()` returns a tuple with the resulting string and a number indicating how many substitutions were made. A tuple is another fundamental Python data type. It is almost exactly like a list except that it can't be

changed; it is what Python calls "immutable." (From this we can deduce that lists **can** be modified "on the fly.") It is an array that is accessed by number subscripts. `SubResult[0]` contains the modified result string (remember, programmers start counting from 0!) and `SubResult[1]` contains the number of substitutions actually made.

This line also demonstrates the method of breaking up long lines. If you put a single character at the end of a line, the next line is considered a continuation of the same line. It's useful for making code readable, or fitting Python code into narrow HTML browser windows.

The next line of code says that if there weren't any substitutions made, this is a problem, so complain loudly by raising an (unhandled, fatal) exception. In other words, if our special "INSERT CONTENT HERE" marker was not found in the template file, this is a big problem!

Finally, since everything is okay at this point, go ahead and print out the special Content−Type line and then our complete string, with our Content inside our Template.

You can use this function in your own CGI Python programs. Call it when you've assembled a complete string containing all the special output of your program and you're ready to display it.

Later, we will modify this function to handle special types of Content data including objects that contain sub−objects that contain sub−objects and so on.

Putting the pieces together

Let's review what we've covered so far. We've learned how to get information from the user via a CGI form and we've learned how to print out information in a nicely formatted HTML page. Let's put these two separate bits of knowledge together into one program that will prompt the user with a form, and then print out the information submitted by that form.

We're going to go ahead and get fancy and use two templates this time ... one, the same generic site template as before, and the other template will contain our form that will be presented to the user.

```
form.html
```

```
<FORM METHOD="POST" ACTION="sample1.py"> <INPUT TYPE=HIDDEN
NAME="key" VALUE="process"> Your name:<BR> <INPUT TYPE=TEXT
NAME="name" size=60> <BR> Email: (optional)<BR> <INPUT TYPE=TEXT
NAME="email" size=60> <BR> Favorite Color:<BR> <INPUT TYPE=RADIO
NAME="color" VALUE="blue" CHECKED>Blue <INPUT TYPE=RADIO NAME="color"
VALUE="yellow">No, Yellow... <INPUT TYPE=RADIO NAME="color"
VALUE="swallow">What do you mean, an African or European swallow? <P>
Comments:<BR> <TEXTAREA NAME="comment" ROWS=8 COLS=60> </TEXTAREA>
<P> <INPUT TYPE="SUBMIT" VALUE="Okay"> </FORM>
```

We'll use the same generic template file from before.

Here is the program itself:

```
sample1.py:
```

```
#!/usr/bin/python import re import cgi # specify the filename of the template file
TemplateFile = "template.html" # specify the filename of the form to show the user FormFile
= "form.html" # Display takes one parameter − a string to Display def Display(Content):
TemplateHandle = open(TemplateFile, "r") # open in read only mode # read the entire file as
a string TemplateInput = TemplateHandle.read() TemplateHandle.close() # close the file #
this defines an exception string in case our # template file is messed up
BadTemplateException = "There was a problem with the HTML template." SubResult =
re.subn("<!−− *** INSERT CONTENT HERE *** −−>", Content,TemplateInput) if
SubResult[1] == 0: raise BadTemplateException print "Content−Type: text/html\n\n" print
SubResult[0] ### what follows are our two main 'action' functions, one to show the ### form,
and another to process it # this is a really simple function def DisplayForm(): FormHandle =
open(FormFile, "r") FormInput = FormHandle.read() FormHandle.close()
Display(FormInput) def ProcessForm(form): # extract the information from the form in easily
digestible format try: name = form["name"].value except: # name is required, so output an
error if # not given and exit script Display("You need to at least supply a name. Please go
back.") raise SystemExit try: email = form["email"].value except: email = None try: color =
form["color"].value except: color = None try: comment = form["comment"].value except:
comment = None Output = "" # our output buffer, empty at first Output = Output + "Hello, "
if email != None: Output = Output + "<A HREF="mailto:" + email + "">" + name +
"</A>.<P>" else: Output = Output + name + ".<P>" if color == "swallow": Output = Output
```

```
+ "You must be a Monty Python fan.<P>" elif color != None: Output = Output + "Your
favorite color was " + color + "<P>" else: Output = Output + "You cheated! You didn't
specify a color!<P>" if comment != None: Output = Output + "In addition, you said:<BR>" +
comment + "<P>" Display(Output) ### ### Begin actual script ### ### evaluate CGI request
form = cgi.FieldStorage() ### "key" is a hidden form element with an ### action command
such as "process" try: key = form["key"].value except: key = None if key == "process":
ProcessForm(form) else: DisplayForm()
```

Notice that we defined the function Display() again. This isn't really necessary; in fact, it's a bad idea to cut−n−paste code from one script to another. A far better solution (which I will show you later) is to put all your functions that are common to multiple scripts in separate file. That way, if you want to improve or debug the function later, you don't have to hunt for each occurrence of it in all of your scripts. However, we can get away with it for this learning exercise.

Note that since Python is a dynamically interpreted language, functions have to be defined before they can be used. That's why the first thing this program does is define its functions.

The DisplayForm() function is very simple and uses the same principle as the Display() function (which it, of course, calls upon to do most of the hard work.) Using these kinds of functions rather than embedding static HTML forms will save you an incredible amount of aggravation. Sometimes, of course, you'll need to generate HTML on−the−fly; we'll talk about that later.

The ProcessForm() function has two main parts. The first extracts values from the <INPUT> fields of the form and stores them in regular variables. This is the section of your program logic where you validate the information in the form to make sure it is kosher by your application's definition.

It is a VERY bad idea to use form data provided by random people on the web without validating it; especially if you're going to use that data to execute a system command or for acting on a database. Naively written CGI scripts, in any language, are a favorite target for malicious system crackers. At the minimum, make sure form information doesn't exceed a certain appropriate length. Passing huge strings to external programs where they aren't expected can cause buffer overflows which can lead to arbitrary code execution by a clever cracker. We will be discussing security in greater detail in the next article. Fortunately, Python makes it easy to write scripts securely.

In our program, we enclose these validation statements in try: and except: blocks because if the value isn't present, it will generate an exception, which could potentially stop your program. In this case, we catch the exception if a given field was left blank (or simply not in the original form.) That variable will be assigned the None value, which has special meaning in Python. (By definition, it is the state of not having a value, similar to NULL in other languages.) A special case is the "name" field; we've decided that that field is absolutely required. If "name" is not given, then the program will use its Display() method to print a helpful message to the user: "Please go back and try again." It then raises the exception SystemExit, which in effect, stops the script then and there.

Once we have validated all of our form input, we can act on it however we choose. In this case, we just want to print a simple message acknowledging receipt of the information. We set up an empty string to use as a buffer for our output, and we begin adding to it. We can conditionally include stuff in this buffer depending on whether or not a value is defined is None. Sharp observers will note that we cheated a bit and put a little actual HTML directly in our code. Again, this is not ideal, but we can get away with it in this learning

exercise. The next article in this series will address using objects to avoid the problems of embedding HTML directly in output statements.

The actual main part of the script follows the function definitions, and is very simple. We obtain the form, and conditionally execute one of our two action functions depending on the value of a hidden form element called "key". We know that if this key is present, and it is set to the value "process," we'll want to process the form. Otherwise, we'll just display the form. If you're not sure what I mean by hidden key, go back and look at our form.html template carefully.

Simple Database Access

The last thing we're going to learn about in this installment of our series on learning CGI programming in Python is how to make simple database queries. But instead of providing a complete demonstration program, I'm just going to give you a few code snippets for running SQL queries. You can combine these fragments with what you've learned about working with forms to make a useful web application in Python.

Python has a standard API (Application Programming Interface) for working with databases. The interface is the same for every database that is supported, so your program can work unmodified (in theory) with any common database. Currently, the Python DBI (Database Interface) supports most popular SQL relational databases such as mSQL, MySQL, Oracle, PostgreSQL, and Informix.

There is an excellent resource for people interested in interfacing Python program with databases: the Database SIG (Special Interest Group.)

This article will only cover simple read−only queries. In the next installment we'll cover more complex database operations.

While the programming *interface* is standardized across databases, it is still necessary to have the module that provides access to your particular database available to the Python interpreter. Often these modules are a separate download and are not included with the standard Python distribution. See the Database SIG mentioned above for instructions on obtaining the module for your database. The code that initializes the database connection will be somewhat dependent on a specific database, while the rest of the code that actually uses the database should work across all supported types of database.

Once you've installed the appropriate module on your system, make it available with an import statement.

```
>>> import Mysqldb
```

...loads the MySQL DBI module.

Now we want to initialize the database module by connecting to it. The database module must be given a string in a specific format containing the name of the database to use, your username, and so on. The format is "Database_Name @ Machine_Name User Password". If you don't know some of this information, you'll have to ask your friendly local database administrator.

For instance:

```
>>> import Mysqldb >>> SQLDatabase = "guestbook" >>> SQLHost = "localhost" >>>
SQLUser = "gbookuser" >>> SQLPassword = "secret!" >>> connectstring = SQLDatabase +
"@" + SQLHost + " " >>> connectstring = connectstring + SQLUser + " " + SQLPassword
>>> connection = Mysqldb.mysqldb(connectstring) >>> cursor = connection.cursor()
```

Notice that the last statement returned a thing called a cursor. Coincidentally, we've also named the object that holds that 'cursor', but just as easily it could have been named 'bilbobaggins'. All database action is performed through a cursor, which functions as an active connection to the database. The cursor object that we obtained has a number of methods including execute() and fetchall(). execute() is used to actually execute SQL statements. Use fetchall() to get the results of the previous execute() as a list of tuples. Each tuple represents a specific record/row of the database.

Once you have the cursor, you can perform any SQL statement your database will support with cursor.execute(statement). Here is a simple example that will fetch all rows of a guestbook and display them as an HTML list. [Again, it's not a good idea to embed HTML directly into code like this; it's just an example.]

```
# get all entries from gbook table, ordered by time stamp myquery = "SELECT * FROM
gbook ORDER BY stamp" handle.execute(myquery) Results = handle.fetchall() # fetch all
rows into the Results array total = len(Results) # find out how many records were returned #
we'll want a blank list to hold all the guestbook entries entries = [] if total < 1: print "There
weren't any guestbook entries!" ### do something else here else: for record in range(total):
entry = {} # a blank dictionary to hold each record entry["gid"] = Results[record][0] # field 0
= guestbook ID entry["stamp"] = Results[record][1] # field 1 = timestamp entry["name"] =
Results[record][2] # and so on... entry["email"] = Results[record][3] entry["link"] =
Results[record][4] entry["comment"] = Results[record][5] entries.append(entry) # add this
entry to the master list # we'll pretend we set up an HTML table here... ### parse variables
into table for entry in entries: print "<LI>" + entry["name"] + "@" + entry["email"] + " said: "
+ entry["comment"]
```

Notice that we copied the information out of the Results list into a list of dictionaries. It's not absolutely necessary to do this; in fact it will slow down your program a tiny bit. The benefit to doing that is that you can access each column of a record by name, rather than the number. If you use the record more than once, it becomes much easier to keep the mnemonic names straight than arbitrary numbers.

Other Resources and Links

There are several good books in print on Python, including:

Internet Programming with Python (Watters, van Rossum, Ahlstrom; MIS Press): this one I personally own. Though it is slightly outdated, covering Python 1.4, most of the information is still relevant and useful. Provides a good tutorial on Python and lots of sample code for Internet applications. One of the co−authors is Guido van Rossum, the inventor of Python. Recommended, but due for a new edition.

Programming Python (Lutz; Oreilly &Associates): also slightly outdated, but highly recommended by people who have it. More of a reference than a tutorial.

Python Documentation: the starting point for all official Python documentation including the Library Reference.

The Python Tutorial: a gentle but effective introduction to Python.

The Python FAQ: many questions and answers about specific features and problems. Come here when you can't find your question answered in any other documentation.

Language Comparisons: also known as Python vs. Perl vs. Tcl vs. Java vs. the world. Most articles here are reasonably objective and balanced comparisons. Always pick the right tool for the job. Often, Python is the right tool.

Downloading Python for your platform. Both source code and binaries for a wide variety of platforms can be found there. However, if you are running a Linux distribution such as Debian or Red Hat it will be easier for you to obtain the appropriate package for your system from your distributor.

Regular Expression HOWTO: mastering those tricky but useful regexp's in Python.

Database SIG: a gathering place for people interested in and information pertaining to using tabular databases with Python.