

Kenny Pyatt's

Python Web Development

The following information is Copyright Kenny Pyatt.
Please feel free to share this with as many people as you like.

© Kenny Pyatt. All Rights Reserved.

Forward

I created this to help several of the interns at our office. The goal was to shortcut their learning cycle and get them up to speed. I wrote it in the wiki at the office over the course of a weekend and a couple of week nights. Several of the interns completed the tutorials and helped me catch errors, typos, etc. I hope that no new errors were created in my copy paste from the wiki into Open Office.

I have noticed there is some wrapping issues in some of the code blocks. I have attempted to edit those and hopefully I have managed to correct them.

There are 9 lessons total:

1. Hello, World via HTTP
2. HTML Template Hello World
3. Dynamic HTML Form
4. Database interaction
5. BREAD Intro
6. BREAD via one controller
7. Multiple BREAD interaction
8. A better class of program
9. File management via HTTP

Each lessons starts on a new page. Most people take about two weeks to complete the lessons. I don't cover setting up the server or any of the software I use. You should be able to Google for that.

Hello, World via HTTP

So you are going to be writing your first web based python program. Awesome! This first tutorial gets you into the basics of developing. It's more than just making a browser say "Hello, World!" It's about setting up your work environment. Getting used to working in our setup and framework.

IMPORTANT NOTE: Please read **everything** in this document (and every assignment after this). I promise I didn't write anything you don't need to know. I would also like to issue a warning. The wiki application will occasionally remove a) or ' so if you notice a syntax error from a copy paste check for those first. It is very possible they were typed in but wikitext formatted them out.

Prerequisite Knowledge

It is also valuable to read chapters two and three in [Dive Into Python](#) prior to starting these assignments.

If you are starting your internship, we understand that you may not have come from a programming background. Please check out the following resources on HTML (sorted in the order we recommend reading).

- <http://www.w3schools.com/html/default.asp>
- <http://www.htmlgoodies.com/primers/html/>
- <http://www.htmlcodetutorial.com/>

If you've never made a website before you probably should start with the HTML links and make a simple web page.

Administrator

The server administrator will need to setup a username and password for you for the development server (edison or franklin) and the MySQL database installed on the server your "play" account is created on.

Terminal

If you are going to do much work in the development team you need to start learning the terminal. If you're on a windows box look for putty.exe. If you are on a Linux machine then there will be a system terminal. In windows you will connect to edison via putty.exe.

For all of the assignments anything with a # in front of it is a linux command that you would issue in a server. Anything in a code block that doesn't have a # is a python sample. Of course I say that and the first line of every python program is `#!/usr/bin/python` or something similar. That is the exception. Also the **username** should be your login username and not 'username'.

In Linux you will type the following command:

```
#ssh username@edison
```

Once you are logged in you should be able to do the following command to change directory to the development spot we have created for you.

```
#cd /var/www/html/username
```

After you have changed to the directory please look and make sure

```
#pwd
```

This will tell you what directory you are in. Now let's look at the files in this directory (if it said /var/www/html/username).

```
#ls
```

At this point in the assignments you will probably have an empty directory. Let's make a new directory called lab1.

```
#mkdir lab1
```

Now if you type **#ls** it should give you one directory called lab1. You will repeat these steps for each assignment. As of this writing there were 10 assignments in total. Now let's move into the folder where we will be working in order to start programming.

```
#cd lab1
```

I normally **#ls** all the time. It should be empty now. The next step in this process is the first one where you choose which editor you are going to use.

Code Editor (Integrated Development Environment IDE)

Most beginners start with Windows Notepad or something similar. We have developers that use [Notepad++](#) and [VIM](#) at the office. There are also entire suites of programs for developers called IDEs that make your life easier. At least they all say they do. Probably the easiest editor to use on our server from the terminal is called **nano**. Nano is similar to Notepad and it is feature rich enough for this first lesson. If you already know how to use a different system you can but for the moment please use a terminal based editor so the following commands will work.

```
#nano hello.py
```

This will put you in the nano editor in a blank hello.py file.

Finally the Python script

Type (or copy) the following:

```
#!/usr/bin/python

# Tell the browser it is getting HTML
print "content-type: text/html\n"

# Print the html for the browser
print "Hello, World!"
```

When you are finished type **ctr+X** to exit the program. It will ask you if you want to save. You do! If you want to save the file without exiting nano you can do that with **ctr+O**.

Now we have a python script but we need to tell the server that we intend to run it as an executable. You do that by setting permissions.

```
#chmod a+x hello.py
```

That gives everyone permission to run the program. This isn't a security lesson but it is important to note that programs that allow anyone to run them should be safe to run. Please keep that in mind when programming.

Now let's test this program in the terminal before we check it in the browser.

```
#!/hello.py
```

Notice the ./ before hello.py. That tells the terminal you want the local version of the hello.py file. Just in case there is a hello.py file somewhere else on the server.

It should've printed out the following:

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

```
Hello, World!
```

The first line tells the browser what content is coming so it will know what to expect. You can literally send anything you want. For example if you wrote a python script to show server side real time scaled images (probably a bad idea) you would pass the image time as the content-type. In the case of a jpeg it would be **image/jpeg**.

Test it

Now you can try it in a browser. Just open your browser (hopefully Firefox with the Developer toolbar add-on) and visit <http://edison.xmsolution.com/username/lab1/hello.py>. Don't put the training period. If your browser says "Hello, World" and not Internal Server error then please move on to assignment two.

HTML Template Hello World

Hopefully that last assignment went well. You might need to refer back to it in order to do some of the things in this assignment. I will not be typing every command in this one. Create a new directory called `/var/www/html/username/lab2` and go ahead and change directory.

HTMLTMPL

We really don't want to include our HTML in our python. Primarily because we have graphics artists that develop the HTML at the same time we are working on the python. It makes for a good environment for separation of the design from the logic. Since we want to let the Graphics team make the design we need a way to plug values into their templates. We use [htmltmpl.py](#) to help us do that. To get a copy of `htmltmpl.py` in your `lab2` directory you can do the following:

```
#cp /var/www/html/labs/htmltmpl.py .
```

That will copy the `htmltmpl.py` file into your current directory. This is probably a good time to read some of the `htmltmpl` [documentation](#). If you get confused don't worry to much. It will slowly seap into your brain.

Make your first HTML template

Woot! Something easy. HTML is pretty straight forward. If you are not familiar with it you probably need to quickly Google for a tutorial. Especially since you are doing web development. And for the record, HTML is not a programming language. I know it stands for HyperText Markup Language. It is a **Markup** Language. Not a **programming** language. If you say I program in HTML you will not be allowed into Geek Heaven.

```
#nano main.html
```

Put the following (via copy/paste) into the nano screen.

```
<html>
<head>
  <title>Hello, World</title>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

Python control file

Now we are going to use python to import the html template (`main.html`) and print it to the browser.

```
#nano index.py
```

Now put the following code into the `index.py` file.

```
#!/usr/bin/python

# Import the HTML Template manager
from htmltmpl import TemplateManager, TemplateProcessor

def main():
    """ This is the controlling function """
```

```
# Load the Template processor
proc = TemplateProcessor(html_escape=0)

# Set the template
template = TemplateManager(precompile=0).prepare("main.html")

# Send the template to the browser as HTML
print "content-type: text/html\n"
print proc.process(template)

# Run the main
if __name__ == "__main__":
    main()
```

Don't just copy paste. Read the comments and lines and try to understand what is going on.

I kinda slipped somethings in on you there.

The **from htmltmpl** line gets code from the file `htmltmpl.py` and loads it in your code.

I also added a function called **main()**. Every program you ever write for XM solutions will have a main function. The only exception to this rule is when you write a class (covered in a later assignment). The comments in the program should explain everything going on in the `index.py`.

The **__name__** is a python thing. For now just include it in your program and be happy. If you are curious you could [google](#).

Your going to need to `chmod` it and test it from the command line. If your browser shows a slightly bigger "Hello, World!" than the last assigment then congratz. You can move on to assignment 3.

Dynamic HTML Form

Each lesson builds off of the previous one. If you were unable to get the previous lesson to work you will need to go back and ask someone for help. I will also warn against copy/pasting these examples and not studying them. In my experience I learn a lot more if I take the time to type them out and try to figure out each line. I know that takes longer but it will be worth it in the long run.

In Assignment 3 we are going to develop some dynamic screens using [CGI](#). This will allow us to get information from users via the browser. The HTML Form is the most common method of transmitting data from a browser to a server.

This project is going to be the start of a user manager. When you are finished with the next couple of assignments you will have created a simple tool to add, remove, edit, and browse users. We are going to create one python file and two HTML files for this assignment.

To start this assignment let's make a copy of the lab2 folder and call it lab3.

```
#cp -r lab2 lab3
```

Build the HTML Form

Now let's edit the main.html file and add an HTML form. The form I want will have 7 fields. First Name, Last Name, Email Address, Street Address, City State, and ZIP. Here is an example of a form with one of the fields.

```
<form method='post' action='index.py'>
  <input type='hidden' name='mode' value='results'>

  <p><label name="firstname">First Name:<br>
  <input type='text' name='firstname'></p>

  <input type='submit' value='Create User'>
</form>
```

The first line above tells the browser which python script gets the data when the form is submitted. In this case I chose **index.py**. After that is a hidden field named *mode* which will tell the python script which screen to display next. The next line has a label which is just a nice thing for the graphics team so they have a CSS hook to style the label. The first input is the data collection field for the Firstname. The second input type is submit and it creates the button that 'sends' the form to the server.

I would also like you to edit the title of the page from *Hello, World* to *Add User*.

To test this form you can pull it up in a browser directly by accessing <http://edison.xmsolution.com/username/lab3/main.html>. If you see a simple HTML form then you should be ready for the next step.

The results screen

Using HTML Template we will create another HTML file that will display the results of what the user submitted. This form is going to be fairly simple. We will call it results.html so if you remember from assignment 1 you will need to **#nano results.html**.

```
<html>

  <head>
    <title>New User Information</title>
  </head>
```

```

<body>
  <h1>New User Information</h1>
  Firstname: <TMPL_VAR firstname><br />
</body>

</html>

```

This screen introduces the `<TMPL_VAR>` tag. In `htmptmpl.py` there is a feature to replace `TMPL_VAR` tags with dynamic values. We will discuss how to do that in the next paragraph. For now understand that the `TMPL_VAR` tag will be removed and the users firstname will appear in it's place.

The python magic

We need to create a simple python file to handle two different things. The first thing is displaying the initial form (this is very similar to your last assignment). The next mode is to display the result of what they typed.

We are going to tell the python script which mode to be in based on the CGI variables we send it from the browser.

Take your python program from the last assignment (`index.py`) and edit it to look like the following code.

```

#!/usr/bin/python

# Import the HTML Template manager
from htmptmpl import TemplateManager, TemplateProcessor

# Import the CGI handler
import cgi

def main():
    """ This is the controlling function """

    # Create a CGI object called form
    form = cgi.FieldStorage()

    # Load the Template processor
    proc = TemplateProcessor(html_escape=0)

    # We need to determine which "mode" we need to be in
    # The mode will determine which template we load
    if form.getvalue("mode") == 'results':
        # we need to be on the results screen
        template = TemplateManager(precompile=0).prepare("results.html")

        # They sent us results so we need to load them in the template
        proc.set("firstname", form.getvalue("firstname"))
        # The rest of your variables would go here

    # This is the default mode
    else:
        # we need to be on the screen to gather information because the user isn't
        # sending us any
        template = TemplateManager(precompile=0).prepare("main.html")

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

# Run the main
if __name__ == "__main__":

```

```
main()
```

There is another neat feature in `htmltmpl.py` called `proc.set()`. The `proc.set` line allows you to load the values you want to replace the `TMPL_VAR` tags. In this case I am replacing "firstname" with the value from the user submitted HTML form. I get that via the `form.getvalue("firstname")`.

It is worth mentioning the `precompile=0` line in the above code. This is a feature which the author strongly recommends you use during development but that you turn on when you deploy the site. Precompiled load much faster. You can enable this feature by removing the entire `precompile=0`.

Testing

Now you should be able to test your program in both the browser and the terminal. The terminal offers a neat trick here to help you test your program. You can pass CGI variables to your python script as command line arguments.

For example:

```
#!/index.py mode=results
```

Should start your program in the results mode. You can pass multiple arguments with the `&` between them. Please notice the "s

```
#!/index.py "mode=results&firstname=Kenny"
```

You should be ready for the next assignment. Congrats on getting this far. You could probably write very basic web applications with the tools you've already learned. I recommend continuing because I am currently showing you ways to do things. Later I will show you better methods to achieve even cooler results.

Database interaction

XM Solutions choose MySQL for their database software. Primarily because it's very powerful. It scales well. A lot of people know how to use it already and because more than half of the 100 largest sites online are powered by a MySQL backend. It is pronounced My-S-Q-L according to popular legend because Microsoft managed to patent/trademark/copyright/own the Pronunciation of "Sequel Server". So the official way to pronounce it is to sound out the letters.

Using the database

Since you are working for XM solutions we already have a test database created. You just need to access it via your python script. You are probably not the first person to use the test database so we need to log in and create a table that will be your test table.

```
#mysql -u username -p
Enter password:
```

The password prompt doesn't echo (print) anything as you type your password. I am sure this is for security reasons. On the XM server we have configured the mysql prompt to display your username and the database your using. When you first login this will say **[username] (none)>**. You need to select a database to use. At the end of each MySQL command is a ; (and technically the enter character).

```
[username] (none)> use test;
Database changed
```

use databasename allows you to switch between databases. Since we are using the test database we said **use test**

Now we need to create a table for our users. In assignment 3 we discussed the user would have the following values: First Name, Last Name, Email Address, Street Address, City State, and ZIP. Databases are similar to a spreadsheet (at least in how you think about them). They have rows and columns. In a spreadsheet they have sheets (accessible via the tabs at the bottom). Sheets in MySQL are called tables. They are always named. Rows hold a single item. Columns define the types of data stored for each item.

```
+-----+
|  firstname  |  lastname  |  email  |
+-----+
| Kenny       | Pyatt      | kenny.pyatt@youwish.com |
+-----+
```

In the table above there would be one user.

Create the table

In this next section I am doing something for the columns that I would never do in a production site (production = live online). For the sake of ensuring that you don't have any weird problems during the learning phase of your programming journey the following create table syntax is very safe.

```
[username] test> create table usernameusers (
->   id int not null auto_increment,
->   primary key(id),
->   firstname varchar(255),
->   lastname varchar(255),
->   email varchar(255),
```

```
-> address varchar(255),
-> city varchar(255),
-> state varchar(255),
-> zip varchar(255));
Query OK, 0 rows affected (0.01 sec)
```

When you hit enter it will put the ->. You don't type that. This will create a **usernameusers** table with the 7 columns that you are going to save each of your users values in.

To exit mysql

```
[username] test> quit;
```

Now you need to go back to your lab folders and copy lab3 to a new folder called lab4.

Python with the Database

We are going to modify our last program to accept user input (via the HTML form we already have) and save it to the database.

```
#!/usr/bin/python

# Import the HTML Template manager
from htmltmpl import TemplateManager, TemplateProcessor

# Setup browser error reporting
import cgi; cgi.enable()

# Importing Database functionality
import MySQLdb

# Import the CGI handler
import cgi

# Create a Database connection
conn = MySQLdb.connect(
    host = 'localhost',
    db = 'test',
    user = 'username',
    passwd = 'password')
cursor = conn.cursor(MySQLdb.cursors.DictCursor)

def main():
    """ This is the controlling function """

    # Create a CGI object called form
    form = cgi.FieldStorage()

    # Load the Template processor
    proc = TemplateProcessor(html_escape=0)

    # We need to determine which "mode" we need to be in
    # The mode will determine which template we load
    if form.getvalue("mode") == 'results':
        # we need to be on the results screen
        template = TemplateManager(precompile=0).prepare("results.html")

        # They sent us results so we need to load them in the template
        proc.set("firstname", form.getvalue("firstname"))
        # The rest of your variables would go here

        # This is where we will save the value to the database
```

```

        if form.getvalue("firstname") != "":
            cursor.execute("INSERT INTO usernameusers(firstname) VALUES (%s)",
                form.getvalue("firstname"))

    # This is the default mode
    else:
        # we need to be on the screen to gather information because the user isn't
        # sending us any
        template = TemplateManager(precompile=0).prepare("main.html")

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

# Run the main
if __name__ == "__main__":
    main()

```

You will notice the `cursor.execute()` line that inserts a new user and sets their `firstname` in the database. This code is a little weak in the set of features I really need to show you database interaction but we get one line to add the `firstname`. For your code please include all 7 fields. There is an important difference in including one value vs. two or more.

```

        cursor.execute("INSERT INTO users(firstname, lastname) VALUES (%s, %s)",
            (form.getvalue("firstname"), form.getvalue("lastname")))

```

In the second example you don't pass a single variable as the 2nd argument to `cursor.execute()`. You pass a tuple of values. Each value lines up with the respective `%s` in the SQL query.

There are a couple of other lines worth discussing. I added **`import MySQLdb`** in order to bring the database features into python. I also added **`import cgitb; cgitb.enable()`** which turns on browser based error messages. At this stage of the learning process your goal is to get it working in the browser. If your browser gives an internal server error it is probably syntax related. You can test it by running the program in the terminal. Otherwise run the program in the browser. Error messages will be a pink and purple (I hate the colors) error screen that gives you a traceback of what went wrong.

Checking the database

If you got the item to save to the database your finished with this assignment. But how do you know if it worked. First it should have shown the results page. Second it should show up in the database.

```

[username] test> SELECT * FROM users;

```

This will show you the current state of the `users` table. Hopefully your new user is happily stored in the database.

If you want to clear the database and start over you can do that with the **`truncate users`** command. That basically starts the table over again as if it was brand new.

I will look forward to seeing you in Assignment 5 (like I can see via the wiki or something :-)

BREAD Intro

Overview

This is a pretty big concept. Sometimes users struggle to remember (or even understand) the interaction between multiple files, modes, and the database. It is understandable if this particular assignment takes some time. I would be surprised if a new programmer could finish it in one sitting. I am not trying to dissuade you. I just want you to understand this is a complicated topic and for most people it will be a challenge.

Design Concepts

Up to this point you have performed the basic operations required to create a simple database driven web application. That being said you are now equipped to start learning some of the concepts in designing software. One of the common concepts in software design is called [MVC architecture](#). If you didn't click the link click it now so I don't have to type out what MVC stands for. Thanks. Fundamentally there are three layers.

I know the MVC purists won't accept this structure but for the purposes of this tutorial I want you to think of the **Model** layer as the database and the python scripts that directly interact with them. The **View** will be the html templates that you create to show the data. The **Controller** will be the glue that takes the html template and the database information and loads them together to form a screen which is then sent to the user.

In "pure" MVC the Model is the data and the View is how you look at it so I feel like this interpretation is close enough for the real world but obviously your professor would laugh as he marked the large F on your paper.

BREADs are tasty

Let's say you've written a computer game. You want to setup a website with a simple user registration system where users can join. You're going to write the administration panel so you can ban the little annoying 7th graders and older men who act like little annoying 7th graders. You're going to need a pretty standard set of functions to manage your users. You will need to be able to Browse through them. You need to add new users. Edit them if they need their password reset. Delete them if they annoy you. You will need to see the details of the user that won your free game giveaway. We call that **set of functions** a **BREAD**.

- **B**rowse
- **R**ead
- **E**dit
- **A**dd
- **D**delete

I personally define three other modes that are important to us here at XM solutions.

- **C**reate
- **S**ave
- **C**onfirm

Create is the mode you go to after you run Add. Save is the mode you go to after you run edit. Confirm is the mode you run before you delete.

Here is a rough ASCII flow chart

- **B**rowse
- **R**ead
- **E**dit -> **S**ave

- Add -> Create
- Confirm -> **Delete**

HTML Template nav bar

There is another neat feature in `htmltmpl.py` that I feel was poorly implemented. That being said we will still use it because it is still useful.

Create a directory called **inc**.

```
#mkdir inc
```

now change into that directory

```
#cd inc
```

We are going to create one navigation bar that we will include in every html template we make for this lab. Create a file called `navbar.html`. If you have forgotten you would type **#nano navbar.html** in the terminal.

```
[<a href='browse.py'>Browse</a>]  
[<a href='add.py'>Add</a>]
```

There are only two modes that can be accessed without specifying the user we would want to interact with. You need to go down a directory to your base lab folder.

```
#cd ..
```

This will take you backwards (down) one directory. If you didn't already know this then the directory structure is often called a tree. The root is the base and it is represented with a `/`.

Create the Add mode first

We are going to start with the add function because it is a lot easier to see, edit, and delete users that have been added to the database. In the last lab you started making the add mode. Actually, I think you finished it. Copy your `lab4` folder to `lab5` and rename your `index.py` file to `add.py`.

```
#mv index.py add.py
```

Then rename your `main.html` to `add.html`. Rename `results.html` to `create.html`. You may want to edit your `create.html` to say things like "A new user was created" or even better "Created the new user John Smith". You will need to edit the `add.py` and change the name of the template files in `add.py`.

Ok the Add mode is finished! Good job.

Now let's write the Browse mode

Create a file called `browse.py` and put the following in the file.

```
#!/usr/bin/python  
  
# Import the HTML Template manager  
from htmltmpl import TemplateManager, TemplateProcessor  
  
# Setup browser error reporting  
import cgitb; cgitb.enable()  
  
# Importing Database functionality  
import MySQLdb
```

```

# Create a Database connection
conn = MySQLdb.connect(
    host = 'localhost',
    db = 'test',
    user = 'username',
    passwd = 'password')
cursor = conn.cursor(MySQLdb.cursors.DictCursor)

def main():
    """ This is the controlling function """

    # Load the Template processor
    proc = TemplateProcessor(html_escape=0)

    # Let's set the browse mode template
    template = TemplateManager(precompile=0).prepare("browse.html")

    # Let's get a list of the users in the database
    cursor.execute("SELECT id, firstname, lastname, email FROM usernameusers ORDER BY
lastname")
    userList = cursor.fetchall()

    # Thaw the tuple
    userList = list(userList)

    # Let's set the results in the HTML template
    if userList:
        proc.set("Users", userList)

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

# Run the main
if __name__ == "__main__":
    main()

```

The cursor.execute selects all of the users ordered by lastname. The line below it is the cursor.fetchall(). That returns a tuple of dictionaries. Because htmltmpl expects a list of dictionaries we need to use the list() method to "thaw" the dictionary.

I suspect you noticed that you only have the browse mode. In add you had the screen to gather data (Add) and the screen to show the results of the new user being created (Create). In the browse mode you open a template. Take a query of all the users. Shove them together and Woot! Easy Peezy. Now you have to make an HTML template that will show the list of the users. It needs to include a few links to help us read, edit, and delete users as well.

```

<html>
<head>
    <title>Browsing Users</title>
</head>
<body>
    <h1>Browsing Users</h1>

    <TMPL_INCLUDE navbar.html>

    <ul>
    <TMPL_LOOP Users>
    <li>
        <a href='view.py?id=<TMPL_VAR id>'><TMPL_VAR firstname>, <TMPL_VAR lastname>
</a>(<TMPL_VAR email>) -

```

```

        <a href='edit.py?id=<TMPL_VAR id>'>Edit</a> <a href='delete.py?id=<TMPL_VAR
id>'>Delete</a>
    </li>
</TMPL_LOOP>
</ul>
</body>
</html>

```

Since I don't think I fully explained this concept, I want to take a moment and explain the CGI link structure. If you notice in the above HTML I am building a link (`edit.py?id=<TMPL_VAR id>`) that will look like **edit?id=17** when it is rendered. What happens when you visit (click) this link is pretty straight forward. The browser will pass `id=17` to the python program. Your python program will then be able to perform the edit functions (since we called `edit.py`) on the user identified by user id 17. If you need to pass more than one variable you would separate them with the `&` character. So if I called **`save.py?id=17&firstname=Kenny&lastname=Pyatt`** that would pass three variables to the python script.

Delete mode

If you're like me you created dozens of user testing out your `add.py` script. Now you have a screen full of 'asdf', '1234', 'kenny', and other random strings your fingers managed to type. Now let's make a simple method of deleting them.

Copy your `add.py` to `delete.py` and get ready to edit.

Make your `add.py` look like this:

```

#!/usr/bin/python

# Import the HTML Template manager
from htmltmpl import TemplateManager, TemplateProcessor

# Setup browser error reporting
import cgi; cgi.enable()

# Importing Database functionality
import MySQLdb

# Import the CGI handler
import cgi

# Create a Database connection
conn = MySQLdb.connect(
    host = 'localhost',
    db = 'test',
    user = 'username',
    passwd = 'password')
cursor = conn.cursor(MySQLdb.cursors.DictCursor)

def main():
    """ This is the controlling function """

    # Create a CGI object called form
    form = cgi.FieldStorage()

    # Load the Template processor
    proc = TemplateProcessor(html_escape=0)

    # We need to determine which "mode" we need to be in
    # The mode will determine which template we load
    if form.getvalue("mode") == 'delete':

```

```

# we need to inform them that we deleted it
template = TemplateManager(precompile=0).prepare("delete.html")

# This is where we will save the value to the database
if form.getvalue("id"):
    cursor.execute("DELETE from usernameusers WHERE id = %s", form.getvalue("id"))

# This is the default mode
else:

    # Are they sure they want to delete this beautiful user?
    template = TemplateManager(precompile=0).prepare("confirm.html")

    # We need to set the userid of whom they are deleting.
    proc.set("id", form.getvalue("id"))

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

# Run the main
if __name__ == "__main__":
    main()

```

Now that you have the python lets' make some quick HTML to confirm with the user that he knows he is about to delete that user that he loves so much. We are saving so much loss and regret by adding this extra confirmation step. Make a file called **confirm.html**.

```

<html>
<head>
  <title>Delete User</title>
</head>
<body>
  <h1>Are you sure you want to delete the user that has id <TMPL_VAR id></h1>

  <a href='delete.py?mode=delete&id=<TMPL_VAR id>'>Yes</a>
  <a href='browse.py'>No</a>
</body>
</html>

```

Now I would make a **delete.html** that says "User has been deleted click here to go back to browse" or something similar.

Read

The read mode is a fairly simple mode. It has a couple of the features of the edit mode (covered below). Let's make a new file called read.html.

```

<html>
<head>
  <title>User Details</title>
</head>

<body>

<TMPL_INCLUDE navbar.html>

<h1>User Details</h1>
id: <TMPL_VAR id>
First name: <TMPL_VAR firstname>
Last name: <TMPL_VAR lastname>

```

```
<a href='edit.py?id=<TMPL_VAR id>'>Click here</a> to edit this user.  
</body>  
</html>
```

I don't think we introduced any new concepts here. Let's make a new file called **read.py**.

```
#!/usr/bin/python  
  
# Import the HTML Template manager  
from htmltmpl import TemplateManager, TemplateProcessor  
  
# Setup browser error reporting  
import cgitb; cgitb.enable()  
  
# Importing Database functionality  
import MySQLdb  
  
# Import the CGI handler  
import cgi  
  
# Create a Database connection  
conn = MySQLdb.connect(  
    host = 'localhost',  
    db = 'test',  
    user = 'username',  
    passwd = 'password')  
cursor = conn.cursor(MySQLdb.cursors.DictCursor)  
  
def main():  
    """ This is the controlling function """  
  
    # Create a CGI object called form  
    form = cgi.FieldStorage()  
  
    # Load the Template processor  
    proc = TemplateProcessor(html_escape=0)  
  
    # we need to be on the screen to gather information because the user isn't sending  
us any  
    template = TemplateManager(precompile=0).prepare("read.html")  
  
    # Get the values for this user from the database  
    cursor.execute("SELECT id, firstname, lastname FROM users WHERE id=%s",  
form.getvalue("id"))  
    userInfo = cursor.fetchone()  
  
    # Load the values in the template  
    if userInfo:  
        proc.set("id", userInfo['id'])  
        proc.set("firstname", userInfo['firstname'])  
        proc.set("lastname", userInfo['lastname'])  
  
    # Send the template to the browser as HTML  
    print "content-type: text/html\n"  
    print proc.process(template)  
  
# Run the main  
if __name__ == "__main__":  
    main()
```

The only new concept is the `cursor.fetchone()` and I explain that below in better detail.

Edit

The edit mode is almost identical to the add mode. The only difference is the fields on the template contain variables from the database. We will reference the users by their user id. So we will be passing the user id to the edit mode via the CGI variable **id**.

Let's start by copying the add template and editing it to suit our purposes. I will give you the copy command for the last time.

```
#cp add.html edit.html
```

Now let's edit the template as follows.

```
<html>
<head>
  <title>Edit User</title>
</head>1111

<body>

<TMPL_INCLUDE navbar.html>

<h1>Edit User <TMPL_VAR firstname> <TMPL_VAR lastname></h1>

<form method='post' action='edit.py'>
<input type='hidden' name='mode' value='save'>
<input type='hidden' name='id' value='<TMPL_VAR id>'>

<p><label name="firstname">First Name:<br>
<input type='text' name='firstname' value='<TMPL_VAR firstname>'></p>

<input type='submit' value='Save'>
</form>

</body>
</html>
```

Notice we created a line for a hidden **id** value and set the value to a **TMPL_VAR** so we can dynamically set it via the `edit.py` python script. We also used the same **TMPL_VAR** more than once in the same page. That is completely acceptable.

We also need a `save.html` file to show the results of the save.

```
<html>
<head>
  <title>Edit User</title>
</head>

<body>

<TMPL_INCLUDE navbar.html>

<h1>User Saved <TMPL_VAR firstname> <TMPL_VAR lastname>!</h1>

</body>
</html>
```

Now we will create a the `edit.py` by copying the `add.py` and editing it to load the values from the database and then perform an update instead of an insert. It is worth mentioning that in this simple program to handle a few hundred users you could potentially just delete the current user and insert a new one. Under some circumstances that may actually be faster. Those circumstances normally have really large databases spanning multiple hard

disks and are probably not going to be assigned to you in your near-term future.

```
#!/usr/bin/python

# Import the HTML Template manager
from htmltmpl import TemplateManager, TemplateProcessor

# Setup browser error reporting
import cgitb; cgitb.enable()

# Importing Database functionality
import MySQLdb

# Import the CGI handler
import cgi

# Create a Database connection
conn = MySQLdb.connect(
    host = 'localhost',
    db = 'test',
    user = 'username',
    passwd = 'password')
cursor = conn.cursor(MySQLdb.cursors.DictCursor)

def main():
    """ This is the controlling function """

    # Create a CGI object called form
    form = cgi.FieldStorage()

    # Load the Template processor
    proc = TemplateProcessor(html_escape=0)

    # We need to determine which "mode" we need to be in
    # The mode will determine which template we load
    if form.getvalue("mode") == 'save':
        # we need to be on the results screen
        template = TemplateManager(precompile=0).prepare("save.html")

        # They sent us results so we need to load them in the template
        proc.set("firstname", form.getvalue("firstname"))
        proc.set("lastname", form.getvalue("lastname"))
        proc.set("id", form.getvalue("id"))

        # This is where we will save the value to the database
        if form.getvalue("firstname") != "" and form.getvalue("id") != "":
            # since this is a longer query we are breaking it up into multiple lines
            cursor.execute("""UPDATE usernameusers
                            SET firstname=%s, lastname=%s
                            WHERE id=%s""",
                            (form.getvalue("firstname"), form.getvalue("lastname"),
form.getvalue("id")))

    # This is the default mode
    else:
        # we need to be on the screen to gather information because the user isn't
        sending us any
        template = TemplateManager(precompile=0).prepare("edit.html")

        # Get the values for this user from the database
        cursor.execute("SELECT firstname, lastname FROM users WHERE id=%s",
form.getvalue("id"))
        userInfo = cursor.fetchone()
```

```
# Load the values in the template
if userInfo:
    proc.set("firstname", userInfo['firstname'])
    proc.set("lastname", userInfo['lastname'])
    proc.set("id", userInfo['id'])

# Send the template to the browser as HTML
print "content-type: text/html\n"
print proc.process(template)

# Run the main
if __name__ == "__main__":
    main()
```

We did a bunch of new stuff in this code. We added a multi-line SQL query using the triple quotes. It is helpful if you are going to have a really complex query to line up the SQL keywords. You have to use a triple quote! Another thing we did was using the UPDATE SQL syntax. This tells MySQL to update **EVERY** user that has the id that was passed to us by the form. I want to stress this because I have seen a lot of rookie (and experienced) DBAs forget to put the WHERE clause on an update and accidentally update millions of rows with the same value. Could you imagine a bank administrator typing "UPDATE accounts SET balance=0" and emptying every customers bank accounts! Hope they had a back up.

I also introduce the **cursor.fetchone()**. This is a neat function that will give back just one entry from a database query. It returns a simple dictionary with the keys named the column names in the database.

Whew... that was a lot

Take a break if you haven't already... Drink some Dr. Pepper (the [oldest label soft drink](#) in the world) and relax for 5 minutes. You've earned it.

BREAD via one controller

Just imagine

What if you could put all those python files from your last lab into one file? Can you imagine how much easier that code would be to maintain. How crazy would it be if you had 10 or 15 breads in a web application and each mode had it's own python file. You would have 8 times 15 (120) files. We need a neat and tidy way to store them all in one place.

Setup this lab

Make a new folder called lab6 and copy all the HTML files from lab5. If your in the lab6 folder the following commands will copy the HTML.

```
#cp ../lab5/*.html .
#cp -r ../lab5/inc .
#cp ../lab5/htmltmpl.py .
```

I kinda slipped in copying htmltmpl.py in that folder. You will still need it. You are going to have to edit every link and HTML form action to point to **bread.py**. I normally do this as I write the python modes (covered below). If I were writing this I would open a second terminal so I could copy past a bunch between lab5 and lab6. For the entire lab6 project you will only edit the bread.py (except to configure your templates to point to bread.py).

You are going to need to change all the links in the HTML files to bread.py and add a mode. For example:

```
bread.py?mode=add
```

So the navbar.html would be

```
[<a href='bread.py?mode=browse'>Browse</a>]
[<a href='bread.py?mode=add'>Add</a>]
```

bread.py

Now we are going to make one python script called bread.py and put the following at the top of the file.

```
#!/usr/bin/python

# Import the HTML Template manager
from htmltmpl import TemplateManager, TemplateProcessor

# Setup browser error reporting
import cgi; cgi.enable()

# Add the cgi
import cgi

# Importing Database functionality
import MySQLdb

# Create a Database connection
conn = MySQLdb.connect(
    host = 'localhost',
    db = 'test',
    user = 'username',
    passwd = 'password')
```

```

cursor = conn.cursor(MySQLdb.cursors.DictCursor)

# Load the Template processor
proc = TemplateProcessor(html_escape=0)

# Create a CGI object called form
form = cgi.FieldStorage()

def main():
    """ This is the controlling function """

```

The header is pretty much identical to every other python script you made in the last assignment. We did set the **form** and **proc** as global variables this time. I am normally opposed to global variables in the way I have them in this program. The only reason I am doing it this way is because I am confident you will continue through these assignments and learn the much better way to write a BREAD covered in the next assignments. WAIT! Don't skip ahead yet or you will be crazy confused! I am teaching this to show you the interaction. I promise I am not wasting your time.

main() takes control

We are using the CGI variable **mode** to manage what mode we are in for add, edit, and confirm already. What if we did the same thing for all the modes? That would make it very clean and logical. Add this code to your main() function.

```

    # Get the mode from the browser
    mode = form.getvalue("mode")

    # Now let's control what the program does based on which mode we are supposed to run
in.
    if mode == "add":
        Add()
    elif mode == "read":
        Read()
    elif mode == "edit":
        Edit()
    elif mode == "delete":
        Delete()
    elif mode == "confirm":
        Confirm()
    elif mode == "save":
        Save()
    elif mode == "create":
        Create()
    else:
        Browse()

    # We should be finished
    return

```

It really doesn't matter which order you're modes are in as long as you put the browse mode in the else block.

Now we need to start adding some functions to run when the users enters the mode. We will start by copying the main() function in the lab5 add.py file and editing it to work in our new main. You may need to adjust the indentation of this code because the wiki makes it hard for the author to judge depth.

Add() comes to life

The add mode is the part of the if/else statement in add.py that displays the HTML template.

```

def Add():
    " Display the HTML form to add a new user "

    # we need to be on the screen to gather information because the user isn't sending
us any
    template = TemplateManager(precompile=0).prepare("add.html")

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # Finished so back to main()
    return

```

Now let's Read() a bit

We aren't breaking any new ground from the previous code. We just take a few things out of read.py and plug in a return for the end of our function.

```

def Read():
    " Show one users information in detail "

    # we need to be on the screen to gather information because the user isn't sending
us any
    template = TemplateManager(precompile=0).prepare("read.html")

    # Get the values for this user from the database
    cursor.execute("SELECT id, firstname, lastname FROM users WHERE id=%s",
form.getvalue("id"))
    userInfo = cursor.fetchone()

    # Load the values in the template
    if userInfo:
        proc.set("id", userInfo['id'])
        proc.set("firstname", userInfo['firstname'])
        proc.set("lastname", userInfo['lastname'])

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # All done here
    return

```

You still need to edit the code to add the extra fields for the user table. But you should notice this is pretty much a copy paste from the main() function in the read.py file.

Edit()'s making changes

```

def Edit():
    " Show the form to edit one user "

    # we need to be on the screen to gather information because the user isn't sending
us any
    template = TemplateManager(precompile=0).prepare("edit.html")

    # Get the values for this user from the database
    cursor.execute("SELECT firstname, lastname FROM users WHERE id=%s",
form.getvalue("id"))
    userInfo = cursor.fetchone()

```

```

# Load the values in the template
if userInfo:
    proc.set("firstname", userInfo['firstname'])
    proc.set("lastname", userInfo['lastname'])

# Send the template to the browser as HTML
print "content-type: text/html\n"
print proc.process(template)

# Return to the main()
return

```

Browse() shows the scoop

Browse is similar to the previous browse.py main(). This is a trend if you haven't noticed.

```

def Browse():
    " Show the list of all the users "

    # Let's set the browse mode template
    template = TemplateManager(precompile=0).prepare("browse.html")

    # Let's get a list of the users in the database
    cursor.execute("SELECT id, firstname, lastname, email FROM usernameusers ORDER BY
lastname")
    userList = cursor.fetchall()

    # Thaw the tuple
    userList = list(userList)

    # Let's set the results in the HTML template
    if userList:
        proc.set("Users", userList)

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # Back to the main() dude
    return

```

Create() makes stuff happen

This is the other part of the if/else from add.py.

```

def Create():
    " Create a new user in the database "

    # This is where we will save the value to the database
    if form.getvalue("firstname") != "":
        cursor.execute("INSERT INTO usernameusers(firstname) VALUES (%s)",
form.getvalue("firstname"))

    # The user should be added
    return Browse()

```

There is an important difference in this function at the very end. I am return a different function instead of just returning to main and exiting the program. The reason is two fold. One I want to show adding a user without

displaying anything (normally I would do a create.html file here). I also wanted to show that you could return a different function and make have that screen display in the browser.

Save() is my best friend

If you've ever lost work because you forgot to save then join the club. To quote a good friend, "Get a ladder from Home Depot and get over it". There is no reason to loose work due to saving Alzheimers. What was I doing... oh yeah... right.

```
def Save():
    " Save the changes from an edit (one user) "

    # we need to be on the results screen
    template = TemplateManager(precompile=0).prepare("results.html")

    # They sent us results so we need to load them in the template
    proc.set("firstname", form.getvalue("firstname"))
    proc.set("lastname", form.getvalue("lastname"))

    # This is where we will save the value to the database
    if form.getvalue("firstname") != "" and form.getvalue("id") != "":
        # since this is a longer query we are breaking it up into multiple lines
        cursor.execute("""UPDATE usernameusers
                        SET firstname=%s, lastname=%s
                        WHERE id=%s""",
                        (form.getvalue("firstname"), form.getvalue("lastname"),
form.getvalue("id")))

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # We did what we needed to do now let's boggie back to main()
    return
```

Confirm() is a safe step

```
def Confirm():
    " Confirm they want to delete this user "

    # Are they sure they want to delete this beautiful user?
    template = TemplateManager(precompile=0).prepare("confirm.html")

    # We need to set the userid of whom they are deleting in case they want to go
    through with it.
    proc.set("id", form.getvalue("id"))

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # Back to the main()
    return
```

Delete() cleans house

```
def Delete():
    " Remove this user "
```

```
# This is where we will save the value to the database
if form.getvalue("id"):
    cursor.execute("DELETE FROM usernameusers WHERE id = %s", form.getvalue("id"))

# We are finished
return Browse()
```

The delete mode also goes back to browse when it is finished.

The last thing

You still need to run the main program or this operation has been a waste of time. Place this at the bottom of your **bread.py**.

```
# Run the main
if __name__ == "__main__":
    main()
```

Multiple BREAD interaction

Getting good enough to be asked for help

So you've been coding for a couple of days now and your uncle calls you up. The conversation goes something like this:

Uncle: So I hear you're programming HTML websites now.

You: You don't really program HTML. That is a common misconception among the non-programming community. It's a markup language. You mark up text with it. You don't program it.

Uncle: Um... yeah....

* Awkward pause *

Uncle: So can you make me a website for my new music CD store. We are going to sell on ebay dot com and make truck loads of money. I can't afford to pay you right now but I promise I will remember what you did when I make the 1st... err... um... 2nd million.

You: I don't think I've ever written something that complicated before.

(nice try... you should've mentioned the fact that MP3s are killing CD sales)

Uncle: So you'll do it then... great! I will email you my artwork. Can't wait to see what you can do!

Whoa! So what are you going to do? This is so much more complicated than the last assignment you finished. You are going to let your uncle add Artists and then add the Albums for that Artist. You might even have to keep up with Record Labels if your uncle asks for you to sort by label. How do you write a program to do all that. You also have to make the shopping cart and the pages that display the CDs with their prices... AHHHHHH!!!! This is getting complicated quickly. What about uploading image files for the album art? What about pictures of the Artists? This is very complicated. So far you've only handled one type of data. Users. This is scary stuff.

Thinking like a programmer

In the last assignment you created a BREAD which is a simple set of functions, templates, etc. that you need for each type of data you want to track. It just happens in the last assignment you only wanted to manage users. When you start to tackle a problem you need to take it from the perspective of the big picture first.

What do you need to store?

- What types of data are there?
- How much data will there be? In 2 years? In 5 years?

Then you need to think how the user(s) interact with the data.

- Are there administrators and users?
- How many people will use this data? At the same time?

Then you need to think about the interface design.

- Is this a simple HTML website?
- Would it work better as a (insert something you probably haven't learned yet)?

Has someone else already written software to do this? If yes, is the software customizable? affordable? up to the task?

I never want to duplicate/recreate someone else's work. That is, if their work is affordable and they have given me access to use it.

Data types

If you determine that someone hasn't written it or for some reason you can't use it (cost, closed source that lacks a feature you need, etc.) then you start to define your data types.

This program has a few basic data types.

- Albums
- Artists
- Shopping Carts
- Users

Permissions

It has two different permission systems. The website visitors shouldn't be able to edit Albums (they would lower your prices). You are going to need an administration system and a user system.

This can start to get fairly complicated quickly. I recommend that you start writing a [design specification](#) if you are planning on writing this. For this assignment we are going to tackle a small part of the whole project. We are going to write the administration screens for adding Artists and albums. To breads that will work together.

First we need to define the database. I am going to simplify this for our tutorial purposes.

For Artist we need Artist Name and a link to their picture. For Album we need Album name, link to a picture, price, description, and quantity we have in stock.

So we need to create our two database tables. So refer back to Assignment 4 and log into MySQL. We are again going to call your tables your username + the table name. On a normal project we would just say the table name but since there may be more than one person using these tutorials it is safer to use your username. For example my username is rogue so my table would be named **rogueArtist**.

```
[username] test> create table usernameArtist (  
-> id int not null auto_increment,  
-> primary key(id),  
-> name varchar(255),  
-> picurl varchar(255));  
  
[username] test> create table usernameAlbums (  
-> id int not null auto_increment,  
-> primary key(id),  
-> name varchar(255),  
-> picurl varchar(255),  
-> price float,  
-> description text,  
-> artist int,  
-> qty int);
```

There is a neat feature that I want to teach about relationships in a database. Because we expect this to be a fairly low volume database (under 500,000 rows) we are going to do a little bit of a relational structure. It is important to note that every relationship you draw in the database will slow it down as your database grows but you need millions of rows before that is an issue.

In the **usernameAlbum** table there is an artist integer. That integer is going to "relate" to the id of the artist

table. This gives you the ability to query all the albums for a certain artists. For example, if your uncle's store had the Elvis record "Girls! Girls! Girls!" you would save "Girls! Girls! Girls!" with the artist id for Elvis. Sounds simple. Little harder to query. Let's say Elvis had the database id of 17. The following query would give you a list of the album name, price, picture url, and artist name for each album.

```
SELECT album.name, album.price, album.picurl, artists.name
FROM album, artist
WHERE album.artist = artists.id
AND artists.id=17
```

Notice the 'album.artists = artist.id'. This draws the 'relationship' between the two tables.

Workflow in the system

If it were me, I would open several terminals. I would also learn to use a better text editor than nano or at the very least [turn on syntax highlighting](#). Several XMers use gedit in linux, Notepad++ in windows (and putty.exe in windows) and several use vim. I personally choose vim because it was made for programmers and offers dozens of neat hot keys for editing code fast. Vim has a pretty steep learning curve but I think it is worth it.

Python

Now take the last lab assignment that you designed for users and open it in a place where you can do some copy paste. This is the first assignment that I won't give you copy/paste code blocks. You need to learn to do this one on your own.

Let's create two separate breads. One called artist.py and one called albums.py. You should be able to model the BREAD from the last assignment for both artists and albums. You will need to come up with a name/folder system for the html files. I recommend making a directory called templates and then creating two directories called artist and album in the template folder.

```
#mkdir templates
#cd templates
#mkdir artist
#mkdir album
```

You can reference them by saying 'templates/artists/browse.html' when you define your template for each mode.

Once you have them finished we will need to do a bit of integrating. The best way to do that is to write some functions that work every thing. Here is a list of the functions you should have once you finish the two breads.

artist.py

- Browse
- Read
- Edit
- Add
- Delete
- Save
- Create
- Confirm

album.py

- Browse
- Read
- Edit
- Add

- Delete
- Save
- Create
- Confirm

You should also have the HTML files neatly stored in their own folders.

Integration

You now have the structure you need to start integration. When you integrate data types you need to map out which data types line up with (relate) other data types. For example, the artists have albums. This is a simple one-to-many relationship. One artists will have many albums (unless they suck).

artists.py's Read mode

Your uncle is a fairly new computer user. Anything that would be hard for you is going to be impossible for him. Let's make his life easier by adding a feature to the artists.py's Read mode. Let's have it display a list of the albums for that artist.

Something like:

- Elvis Presley
 - Girls! Girls! Girls!
 - Elvis
 - Love Letters From Elvis

That way your uncle can click on 'Love Letters from Elvis' and edit it directly without having to search through all the albums in the album.py's browse mode. The read mode probably looks something like this.

```
def Read():
    " Show one artist information in detail "

    # we need to be on the screen to gather information because the user isn't sending
    us any
    template = TemplateManager(precompile=0).prepare("templates/artist/read.html")

    # Get the values for this user from the database
    cursor.execute("SELECT id, name, picurl FROM usernameArtist WHERE id=%s",
form.getvalue("id"))
    userInfo = cursor.fetchone()

    # Load the values in the template
    if userInfo:
        proc.set("id", userInfo['id'])
        proc.set("name", userInfo['name'])
        proc.set("picurl", userInfo['picurl'])

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # All done here
    return
```

We are going to modify it to add the albums.

```
def Read():
    " Show one artist information in detail "

    # we need to be on the screen to gather information because the user isn't sending
```

```

us any
    template = TemplateManager(precompile=0).prepare("templates/artist/read.html")

    # Get the values for this user from the database
    cursor.execute("SELECT id, name, picurl FROM artists WHERE id=%s",
form.getvalue("id"))
    userInfo = cursor.fetchone()

    # Load the values in the template
    if userInfo:
        proc.set("id", userInfo['id'])
        proc.set("name", userInfo['name'])
        proc.set("picurl", userInfo['picurl'])

    # Now let's add the albums
    cursor.execute("SELECT id, name, picurl, price, qty FROM albums WHERE artists.id
= %s", form.getvalue("id"))
    albumsList = cursor.fetchall()

    # Let's grab the list and plug it in the template if the query returned albums.
    if albumsList:
        proc.set("Albums", list(albumsList))

    # Send the template to the browser as HTML
    print "content-type: text/html\n"
    print proc.process(template)

    # All done here
    return

```

Now we can take the Album.py's browse.html template and copy the album list <TMPL_LOOP> into the artist.py's read.html page. We will want to edit it to remove the columns I didn't query (description and artists). Now he should be able to read/edit/delete albums directly from the artists page. If you add an add album link you could link directly to the album.py's new mode.

The Delete Mode

Another feature that would make his life better is having artist.py's delete mode remove the albums associated with that artist. Go ahead. You should have enough pieces of the puzzle to be able to make that happen.

Done

If your uncle wanted a store obviously you would still have a lot more work to do but for the purposes of learning this topic I feel like you're done. Most beginners really struggle on assignments 6 and 7. If you feel like you need to try them again then by all means please do. They teach the foundation concepts on LAMP development. The next chapter simply makes using these concepts a little cleaner. Unfortunately for new programmers it also adds a level of complexity.

A better class of program

Classes are a nice container to store all your variables and functions in a single place. From the classes you create a variable called an object that gives you access to all the variables.

Your Hello World Class

This is a hello world program built in a Python class.

```
#!/usr/bin/python

class hello():
    """ Simple class example """

    def __init__(self, helloStr = ' '):
        if helloStr == ' ':
            self.helloStr = "Hello, World"
        else:
            self.helloStr = helloStr

    def SayHello(self):
        print self.helloStr

def main():
    """ Controller for the hello class """

    # Load my class
    myHello = hello()

    # Say Hello
    myHello.SayHello()

# Now let's run this bad boy if we were run directly (not imported)
if __name__ == "__main__":
    # Run the main()
    main()
```

This is an amazingly complex structure to house your hello world program but it is simple enough to teach you the basics. To make a new class you just need to name it via the class key word. **class hello()** defines this as the **hello** class.

__init__

The **__init__()** is built into python as a reserved function for object initialization. It is built to run when the class is first used. This allows you to setup your class variables and functions from the very first moment your class is brought into scope.

I believe coming into scope is a sniper term for bringing the victim into focus. Just a fun tidbit.

self

The **self** for you c++ programmers is ***this**. For the non c++ programmers (aka the lucky ones) this concept may be foreign to you. Basically while you work on the class you need a way to reference everything that is in scope for that class. Kinda like class global variables. The **self** gives you access to your class. Or a fun way to say it, the **self** gives you access to yourself.

If I had a class that had a list of people I would reference that list anywhere in my class by saying `self.people`. I should define that people list in my `__init__()` although some people don't do that. In the above example I took the `helloStr` and set it in my `self` with the line `self.helloStr = helloStr`. If you notice in the else I also gave an option for the user to pass their own `helloStr` when they started the class. They could have done that by changing the line `myHello = hello()` to something like `myHello = hello("New Hello String")`

In every function definition the `self` is the first parameter passed to the function. This allows it to be a global object inside the class. You can access class member functions by saying `self.functionName()`.

main()

In the main we instantiate the class by saying `myHello = hello()`. The two parentheses tell Python to run the `init` and return everything in the `self` to `myHello`. Now you have access to the `SayHello()` function by typing `myHello.SayHello()`. Pretty complex but awesomely powerful when you are trying to organize large amounts of code.

destructors

For those of you coming from other languages you will be happy (and a tad concerned) that python handles the destructor for you. I have never had an issue with this. If you are new to classes and would like to know more you can look into python garbage collection. Python does have a `__del__` for this but I consider it an expert only tool and probably not for people reading this tutorial.

File management via HTTP

Managing files via http is a fairly simple task. Once you get everything perfect. That's like the saying, juggling knives is a safe occupation... If you already know how. It's learning to do it hurts. File upload requires that everything be perfect.

Sending files to a server via http is a need skill so let's dive right in and get started.

The Upgraded HTML Form

Create a folder called lab9. In there let's make file.html.

```
<html>
<head>
  <title>File upload script</title>
</head>
<body>
  <form method='post' action='upload.cgi' enctype='multipart/form-data'>
    File: <input type='file' name='myfile'>
    <input type='submit' value='Upload File'>
  </form>
</body>
</html>
```

The HTML file above has an important set of additional features in its HTML form. First notice the **enctype='multipart/form-data'**. This tells the web server (apache) that there is going to be an additional file stream outside the normal CGI POST variables. Second notice the **input type='file'**. This creates the file selection dialog when you click 'browse'.

Now Python's file magic

Hopefully you reviewed [the file handling tutorial](#) already. If not check it out because I am going to dive in pretty quickly.

```
#!/usr/bin/python

# Imports
import sys, os, cgi

# The CGI form object
form = cgi.FieldStorage()

# Where are we uploading?
uploadDir = '/tmp/username/'

# Get the file item from the CGI object
if form.has_key('myfile'):
    fileitem = form['myfile']
else:
    print "HTTP/1.x 400 Bad Request\n"
    quit()

# Open our file for writing
fout = open(uploadDir + "myfile", "wb")

# Download the filestream from the browser and save it to the temp file
while 1:
    chunk = fileitem.file.read(100000)
```

```
# If we are no longer getting a file we are done
if not chunk: break

# Try / Except blocks make everything safer
try:
    # let's try to write this part of the file to our file
    fout.write(chunk)
except:
    # If it didn't work let's tell the browser we had a bad error
    print "HTTP/1.x 400 Bad Request\n"

# Let's close the file since we are finished
fout.close()

# This probably should be an HTML Template but this is ok for testing
print "content-type: text/html\n"
print "Got your file"
```

I always assume that my comments clarify what is going on but I will try to explain. The browser starts to send a file stream. You are going to catch that in 100000 bit chunks and start writing it to the file. You want to keep writing as long as they are sending. It is important to note that web servers (like apache) have a timeout function. Normally around 300 seconds. If this timeout occurs prior to your file transfer completing then the transfer will stop early. You need to consider this when deciding how to get files from the user. Files that are unusually large may need to be sent via a different method. You can also ask the system administrator to up the timeout on the Apache server but that may not be possible.

Upload Folder Setup

You will need to make sure that there is a directory named **/tmp/username**

```
#cd /tmp
#mkdir username
```

Where **username** is your username.

If you don't have permissions to do that you can always create a folder in your `/var/www/html/username/lab9` folder called **uploads** and then `'chmod a+rw uploads'` to allow apache to write to that folder.

Permissions are the number one mistake made in file uploading. Apache normally runs as user 'nobody' on the server. You have to make sure that every user has permission to read and write to the folder you are trying to upload into. This is a critical step that often causes programmers trouble. Check your permissions!