
Real Python for the Web

Release 2.0

Michael Herman

August 01, 2013

Acknowledgements

Writing is an intense, solitary activity that requires discipline and repetition.

Although much of what happens in that process is a mystery to me, I know that the myriad of people I have chosen to surround myself with played a huge role in the development of this course. I am immensely grateful to all those in my life for providing feedback, pushing me when I needed to be pushed, and just listening when I needed silent support.

At times I ignored many people close to me, despite their continuing support. You know who you are. I promise I will do my best to make up for it.

For those who wish to write, know that it can be a painful process for those around you. They make just as many sacrifices as you do, if not more. Be mindful of this. Take the time to be in the moment with those people, in any way that you can. You and your work will benefit from this.

Thank you

First and foremost, I'd like to thank Fletcher, author of the original Real Python course, for believing in me even when I did not believe in myself. He's a talented developer and a natural leader; I'm also proud to call him a friend. Thanks to all my close friends and family (Mom, Dad, Jeff) for all your support and kindness. Derek, Josh, Danielle, Lily, John (all three of you), and Travis - each of you helped in a very special way that I am only beginning to understand.

Thank you also to the immense support from the Python community. Despite not knowing much about me or my abilities, you welcomed me, supported me, and shaped me into a much better programmer. I only hope that I can give back as much as you have given me.

Thanks to all who read through drafts, helping to shape this course into something accurate, readable, and, most importantly, useful. Nina, you are a wonderful technical writer and editor. Stay true to your passions.

Thank you Massimo, Shea, and Mic. You are amazing.

For those who don't know, this course started as a KickStarter. To all my original backers and supporters: You have lead me as much as I hope I am now leading you. Keep asking for more. This is your course.

Finally, thank you to a very popular yet terrible API that forced me to develop my own solution to a problem, pushing me back into development. Permanently.

About the Author

Michael is a lifelong learner. Formally educated in computer science, philosophy, business, and information science, he continues to challenge himself by learning new languages and reading Infinite Jest over and over again. Since he developed his first project - a video game enthusiast website - back in 1999, he's been hacking away ever since.

Python is his tool of choice. He's founded and co-founded several startups and has written extensively on his experiences.

He loves libraries and other mediums for publicly available data. When not staring at a computer screen, he enjoys running, writing flash fiction, and making people feel uncomfortable with his dance moves.

About the Editor

Massimo Di Pierro is an associate professor at the School of Computing of DePaul University in Chicago, where he directs the Master's program in Computational Finance. He also teaches courses on various topics, including web frameworks, network programming, computer security, scientific computing, and parallel programming.

Massimo has a PhD in High Energy Theoretical Physics from the University of Southampton (UK), and he has previously worked as an associate researcher for Fermi National Accelerator Laboratory. Massimo is the author of a book on web2py, and more than 50 publications in the fields of Physics and Computational Finance, and he has contributed to many open source projects.

He started the web2py project in 2007, and is currently the lead developer.

0) Introduction

This is not a reference book. Instead, I've put together a series of tutorials and examples to highlight the power of using Python for web development. The purpose is to open doors, to expose you to the various options available, so you can decide the path to go down when you are ready to move beyond this course. Whether its moving on to more advanced materials, becoming more engaged in the Python development community, or building dynamic web applications of your own - the choice is yours.

This course moves fast, focusing more on practical solutions than theory and concepts. Take the time to go through each example. Ask questions on the [forum](#). Join a local [Meetup](#) group. Take advantage of the various online and offline resources available to you. Engage.

Regardless of whether you have past experience with Python or web development, I urge you to approach this course with a beginner's mind. The best way to learn this material is by challenging yourself. Take my examples further. Find errors in my code. And if you run into a problem, use the "Google-it-first" approach to find a relevant blog post or article to help answer your question. This is how "real" developers solve "real" problems.

By learning through a series of exercises that are challenging, you will screw up at times. Try not to beat yourself up. Instead, learn from your mistakes - and get better.

0.1) Why Python?

Python is a beautiful language. It's easy to learn and fun, and its syntax is clear and concise. Python is a popular choice for beginners, yet still powerful enough to back some of the world's most popular products and applications from companies like NASA, Google, IBM, Cisco, Microsoft, Industrial Light & Magic, among others. Whatever the goal, Python's design makes the programming experience feel almost as natural as writing in English.

With regard to web development, Python powers some of the world's most popular sites. Reddit, the Washington Post, Instagram, Quora, Pinterest, Mozilla, Dropbox, Yelp, and YouTube are all powered by Python.

Unlike Ruby, Python offers a plethora of frameworks from which to choose from, including bottle.py, Flask, CherryPy, Pyramid, Django, and web2py.¹ This freedom of choice allows *you* to decide which framework is best for your applications. You can start with a lightweight framework to get off the ground quickly, adding complexity as your site grows. Such frameworks are great for beginners who wish to learn the nuts and bolts that underlie web frameworks. Or if you're building an enterprise-level application, the higher-level frameworks bypass much of the monotony inherent in web development, enabling you to get an application up quickly and efficiently.

0.2) Who should take this Course?

The ideal reader should have some background in a programming language. If you are completely new to Python, you should consider starting with the original [Real Python](#) course to learn the fundamentals. The examples and tutorials in this course are written with the assumption that you already have basic programming knowledge.

Please be aware that learning both the Python language and web development at the same time will be confusing. Spend at least a week going through the original course before moving on to web development. Combined with this course, you will get up to speed with Python and web development more quickly and smoothly.

What's more, this book is built on the same principles of the original Real Python course:

*We will cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to problems that ordinary people actually want to solve.*²

0.3) How to use this Course

This course has three sections. Although each section does stand alone, the chapters within the sections build on one another. Each chapter is further divided into lessons. Please read the chapters in the order presented.

Each lesson contains conceptual information and hands-on, practical exercises meant to reinforce the theory and concepts; many chapters also include a homework assignment that further reinforces the material and begins preparing you for the next chapter. A number of videos are [included](#) as well, covering many of the exercises and homework assignments.

The first section runs through the basics that underlie web frameworks. The more you work through the exercises and homework problems in that section, the better off you will be when you start working with web frameworks. Much automation happens behind the scenes with web frameworks. This will be confusing at first, which is why the first section details much of this automation. Work hard to get through the first section to really learn the material, and you will have a much easier time learning the web frameworks in the subsequent sections.

Learning by doing

Since the underlying philosophy is learning by doing, do just that: Type in each and every code snippet presented to you. **Do not copy and paste.** The lessons work as follows: After I present the main theory, you will type out and then run a small program. I will then provide feedback on how the program works, focusing specifically on new concepts presented in the lesson.

Finish all review exercises and give each homework assignment and the larger development projects a try on your own before getting help from outside resources. You may struggle, but that is just part of the process. You will learn better that way. If you get stuck and feel frustrated, take a break. Stretch. Re-adjust your seat. Go for a walk. Eat something. Do a one-armed handstand. And if you get stuck for more than a few hours, check out the support forum on the Real Python [website](#). There is no need to waste time. If you continue to work on each chapter for as long

as it takes to at least finish the exercises, eventually something will *click* and everything that seemed hard, confusing, and beyond comprehension will suddenly seem easy.

With enough practice, you will learn this material - and hopefully have fun along the way!

0.4) Errata

I welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Or did you find an error in the text or code? Did I omit a topic you would love to know more about. Whatever the reason, good or bad, please send in your feedback.

You can find my contact information on the Real Python [website](#).

The code found in this course has been tested in Mac OS X v. 10.8, Windows XP, Windows 7, and Linux Mint 14.

0.5) License

This e-book and course are copyrighted and licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. This means that you are welcome to share this book and use it for any non-commercial purposes so long as the entire book remains intact and unaltered. That being said, if you have received this copy for free and have found it helpful, I would very much appreciate it if you purchased a copy of your [own](#).

The example Python scripts associated with this book should be considered open content. This means that anyone is welcome to use any portion of the code for any purpose.

1. <http://wiki.python.org/moin/WebFrameworks>
2. <http://www.realpython.com>

1) Getting Started

1.1) Python Review

Before we begin, you should already have Python installed on your machine. Although Python 3.x has been available since 2008, we'll be using 2.7.4 instead. The majority of web frameworks do not yet support 3.x, because many popular libraries and packages have not been ported from Python version 2 to 3. Fortunately, the differences between 2.7.4 and 3.x are minor.

If you do not have Python installed, please refer to Appendix A for a basic tutorial.

To get the most out of this course, I assume you have at least an understanding of the basic building blocks of the Python language:

- Data Types
- Numbers
- Strings
- Lists
- Tuples
- Dictionaries
- Loops
- Functions
- Modules

Again, if you are completely new to Python or just need a brief review, please start with the original Real Python [course](#).

1.2) Development Environments

Once Python is installed, take some time familiarizing yourself with the three environments in which we will be using Python with: The command line, the Python

Shell, and an advanced Text Editor called [gedit](#). If you are already familiar with these environments, you can skip ahead to the lesson on SQLite.

The Command Line

We will be using the command line, or terminal, extensively throughout this course. If you've never used the command line before, please familiarize yourself with the following commands:

| Windows | Unix | Action |
|-----------------|------------------|--------------------------------------|
| cd | pwd | show the current path |
| cd <dir_name> | cd <dir_name> | move up one directory level |
| cd .. | cd.. | move down one directory level |
| dir | ls | output contents of current directory |
| cls | clear | clear the screen |
| del <file_name> | rm <file_name> | delete a file |
| md <dir_name> | mkdir <dir_name> | create a new directory |
| rd <dir_name> | rmdir <dir_name> | remove a directory |

For simplicity, all command line examples use the Unix-style prompt:

```
$ python big_snake.py
```

(The dollar sign is not part of the command.)

Windows equivalent:

```
C:\> python big_snake.py
```

The Python Shell

The Shell can be accessed through the terminal by typing `python` and then pressing enter. The Shell is an interactive environment: You type code directly into it, sending code directly to the Python Interpreter, which then reads and responds to the entered code. The `>>>` symbols indicate that you're working within the Shell.

Try accessing the Shell through the terminal and print something out:

```
>>> print "The bigger the snake, the bigger the prey"  
The bigger the snake, the bigger the prey  
>>>
```

To exit the Shell from the Terminal, press CTRL-Z + Enter within Windows, or CTRL-D within Unix.

The Shell gives you an immediate response to the code you enter. It's great for testing, but you can really only run one statement at a time. To write longer scripts, we will be using gedit.

gedit

Again, for much of this course, we will be using a basic yet powerful text editor built for writing code called gedit. Like Python, it's cross-compatible with many operating systems. gedit works well with Python and offers excellent syntax highlighting - applying colors to certain parts of programs such as comments, keywords, and variables - making the code more readable.

If you're using Linux, chances are gedit is already installed. If not, use the package manager to install it. You can download the latest versions for Windows and Mac OS X [here](#).

1.3) How to use gedit

Once installed, open gedit. To customize for Python, you then need to access the View tab, which is dependent on your gedit version:

- Mac: Click gedit => Preferences
- Linux: Click Edit => Preferences
- Windows: Click Edit => Preferences

It's common practice to use four spaces for whitespace within a code block rather than a single tab. gedit actually treats tabs as individual spaces, so in the Editor tab, within the Preferences menu, decrease the tab width to 4 and check all options. Now instead of hitting the spacebar four times, you can just hit the tab button.

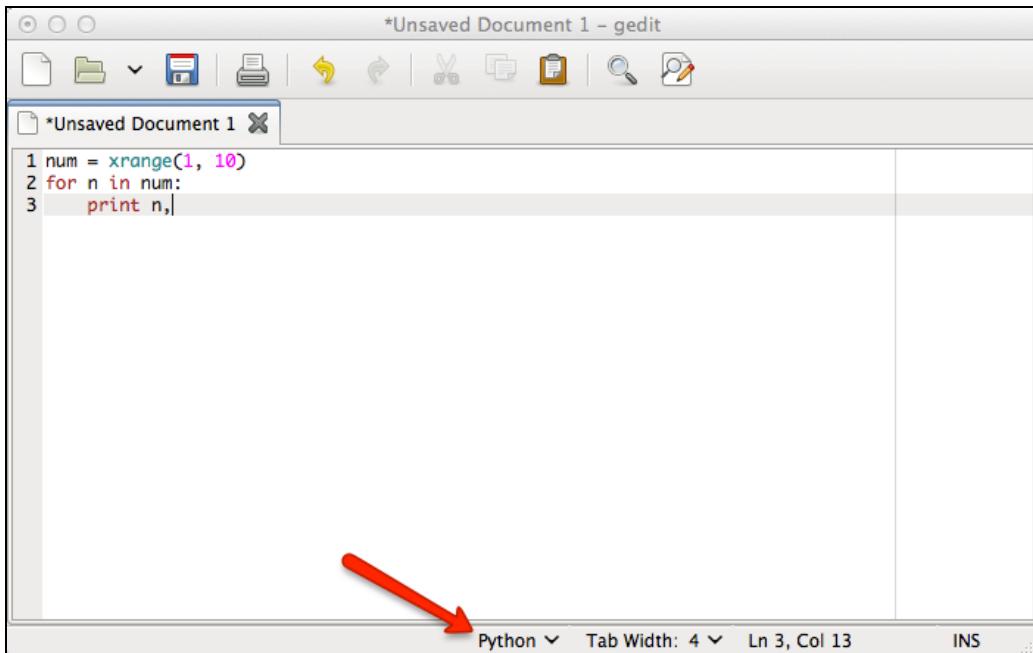
Close the window, and we're now ready to take gedit for a quick test drive. Open a new file.

Type the following code below.

Remember=> Type this out exactly. Do not copy and paste. :)

```
num = xrange(1, 10)
for n in num:
    print n,
```

Now, change the syntax on the bottom drop-down list from "Plain Text" to "Python" so that the correct syntax highlighting will be applied:



Notice the color differences between each object. That's syntax highlighting. It makes it easier to read the code and spot errors.

Save the file as `gedit.py` to your desktop. Get in the habit of saving your programs with a `.py` extension. Most of the basic development environments and editors won't automatically add the extension. If you don't save the file with the `.py` extension, the Python interpreter may not recognize the file as a Python program.

Now to run it, open up your terminal. Navigate to your desktop and then type `python gedit.py`. If you got an error, go back and correct the code. Save the file, and then try to run it again. Once you get it to run, close your terminal. That's it. Pretty simple, right? We're going to have lots of fun with gedit.

If you have your own text editor you currently use, feel free to continue using it. Notepad++ for Windows and TextWrangler for Mac OS X are excellent as well. But, if you're new to programming or just want to give something new a try, go with gedit. It's powerful and easy to use. Just like Python.

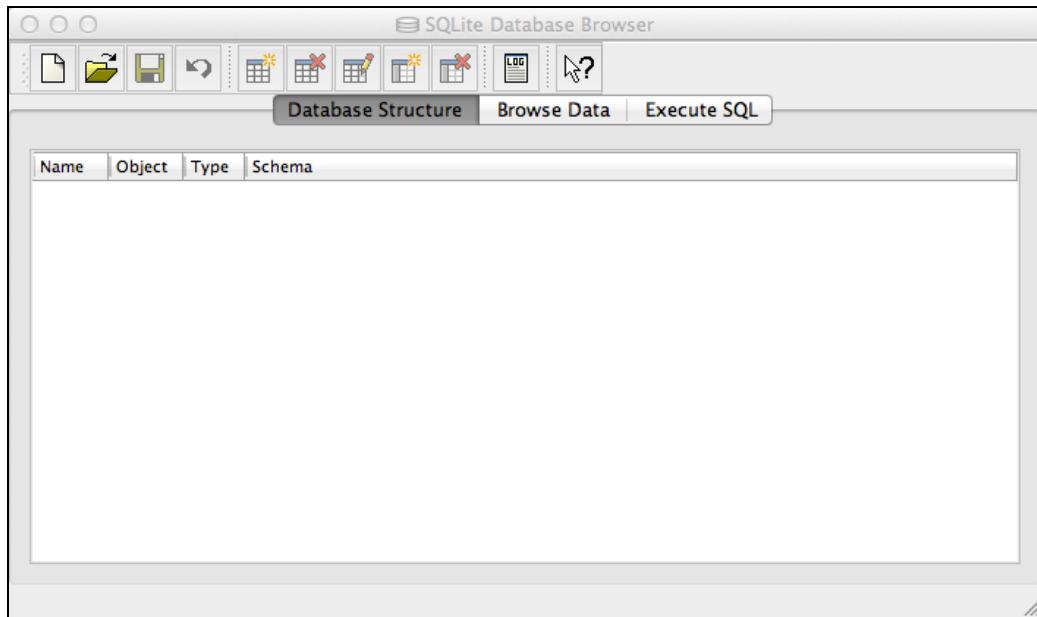
*If you do decide use a different text editor, be aware that mixing tabs with spaces can cause problems with your code, even if the spacing **looks** the same.*

Homework

- Create a directory using your terminal within your "Documents" or "My Documents" directory called "realpython". All the code from your exercises and homework assignments will be saved in this directory.
- See if you can figure out how to change the default starting directory in your terminal. In other words, when you load the terminal you want it to open directly to your "realpython" directory so that you can quickly run a program rather than having to navigate to that directory first. Use Google for assistance.

1.4) Installing SQLite

In the next chapter we will begin working with Python database programming. You will be using the SQLite database because it's simple to set up and great for beginners who need to learn the SQL syntax. Python includes the SQLite library. We just need to install the SQLite Database Browser:



Regardless of the operating system, you can download the SQLite Database Driver from [Sourceforge](#). Installation for Windows and Mac OS X environments are relatively the same. As for Linux, installation is again dependent upon which Linux flavor you are using.

Homework

- Learn the basics of the SQLite Database Browser [here](#).

1.5) Installing easy_install and pip

Both easy_install and pip are Python Package Managers. These essentially make it much easier to install and upgrade Python packages (and package dependencies) by automating the process ¹. In other words, they are used for installing third party packages and libraries that other Python developers create.

easy_install

With easy_install you can simply use the filename of the package you wish to download and install:

```
$ easy_install numpy
```

The above command installs numpy or if you already have it installed, it will attempt to install the new version (if available). If you need a specific version of a package, you can use the following command:

```
$ easy_install numpy==1.6.2
```

To delete packages, run the following command and then manually delete the package directory from within your python directory:

```
$ easy_install -mxN PackageName numpy
```

To download easy_install, go to the [Python Package Index](#) (PyPI), which is also commonly referred to as the "Cheeseshop" in the Python community. You need to download setuptools, which includes easy_install. Download the executable (.exe) file for Windows operating systems, or the package egg (.egg) for Unix operating systems (Mac and Linux). You can install it directly from the file. You must have Python installed first before you can install setuptools.

pip

Pip, meanwhile, is similar to easy_install, but there are a few added features:

1. Better error messages
2. Ability to uninstall a package directly from the command line

Also, while easy_install will start to install packages before it's done downloading, pip waits until the download is complete. So, if your connection is lost during download, you won't have a partially installed package.

Now, since pip is a wrapper that relies on easy_install, you *must* have easy_install setup and working first before you can install pip. Once easy_install is setup, run the following command to install pip:

```
$ easy_install pip
```

To install a package:

```
$ pip install numpy
```

To download a specific version:

```
$ pip install numpy==1.6.1
```

To uninstall a package:

```
$ pip uninstall numpy
```

If you are in a **Unix environment** you will probably need to use `sudo` before each command in order to execute the command as the root user: `sudo easy_install pip`². You will have to enter your root password to install.

Video

Please watch the video [here](#) for assistance with installing setup_tools and pip.

1.6 Installing virtualenv

It's common practice to use a virtualenv (virtual environment) for your various projects, which is used to create isolated Python environments (also called "sandboxes")³. Essentially, when you work on one project, it will not affect any of your other projects.

It's best to work in a virtualenv so that you can keep all of your Python versions, libraries, packages, and dependencies separate from one another.

Some examples:

1. Simultaneous applications you work on may have different dependency requirements for one particular package. One application may need version 1.3.1 of package X, while another could require version 1.2.3. Without virtualenv, you would not be able to access each version concurrently.
2. You have one application written in Python 2.7 and another written in Python 3.0. Using virtualenv to separate the development environments, both applications can reside on the same machine without creating conflicts.
3. One project uses Django version 1.2 while another uses version 1.5. Again, virtualenv allows you to have both versions installed in isolated environments so they don't affect each other.

Python will work fine without virtualenv. But if you start working on a number of projects with a number of different libraries installed, you will find virtualenv an absolute necessity. Once you understand the concept behind it, virtualenv is easy to use. It will save you time (and possibly prevent a huge headache) in the long run.

Some Linux flavors come with virtualenv pre-installed. You can run the following command in your terminal to check:

```
$ virtualenv --version
```

If installed, the version number will be returned:

```
$ virtualenv --version  
1.8.4
```

Use pip to install virtualenv on your system:

```
$ pip install virtualenv
```

Unless otherwise specified, keep all your virtual environments separated by chapter for chapters two, three, and four:

1. Navigate to your "realpython" directory.
2. Run the following command to set up a virtualenv for Chapter 2 within that directory:

```
virtualenv --no-site-packages chapter2
```

This created a new directory, "chapter2", and set up a virtualenv within that directory. The `--no-site-packages` flag truly isolates your work environment from the rest of your system as it does not include any packages or modules already installed on your system. Thus, you have a completely isolated environment, free from any previously installed packages.

3. Now you just need to activate the virtualenv, enabling the isolated work environment:

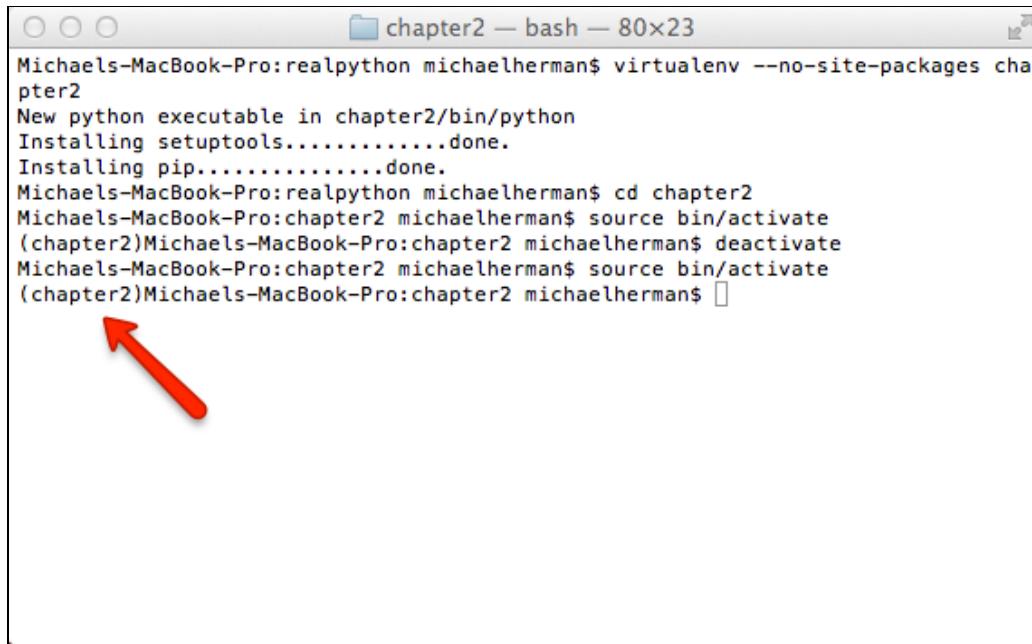
Unix:

```
source bin/activate
```

Windows:

```
scripts\activate
```

This changes the context of your terminal in that this folder, chapter2, acts as the root of your system. Now you can work with Python in a completely isolated environment. You can tell when you're working in a virtualenv by the directory surrounded by parentheses to the left of the path in your command-line:



```
Michaels-MacBook-Pro:realpython michaelherman$ virtualenv --no-site-packages chapter2
New python executable in chapter2/bin/python
Installing setuptools.....done.
Installing pip.....done.
Michaels-MacBook-Pro:realpython michaelherman$ cd chapter2
Michaels-MacBook-Pro:chapter2 michaelherman$ source bin/activate
(chapter2)Michaels-MacBook-Pro:chapter2 michaelherman$ deactivate
Michaels-MacBook-Pro:chapter2 michaelherman$ source bin/activate
(chapter2)Michaels-MacBook-Pro:chapter2 michaelherman$ 
```

When you're done working, you can then exit the virtual environment using the `deactivate` command. And when you're ready to develop again, simply navigate back to that same directory and reactivate the virtualenv.

Video

Please watch the video [here](#) for assistance.

Homework

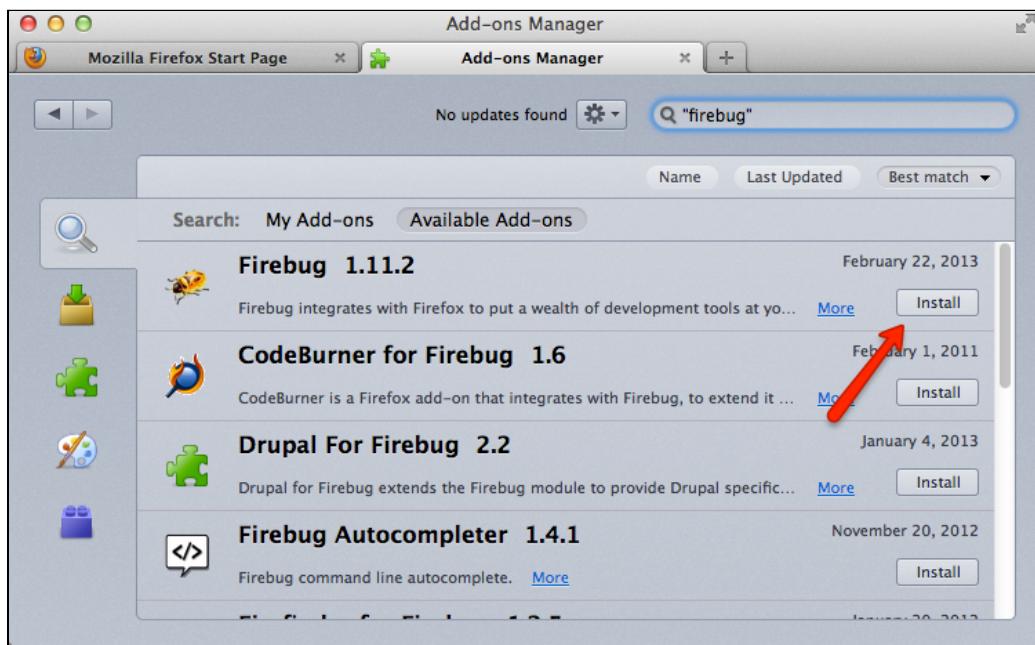
- Create separate virtualenvs for chapters two, three, and four of this course.

1.7) Firebug

You can use whichever web browser you'd like during this course; however, there will be times that we'll need to use the Firebug add-on for Firefox. Firebug is a must for any web developer. It allows you to inspect and analyze various elements that make up a webpage. Other web browsers do come with similar development add-ons, but Firebug is the best in terms of ease of use and available features.

So, go ahead and [download](#) the latest version of Firefox if you don't already have it. Then follow these steps to add Firebug:

1. On the Menu Bar, click "Tools" => "Add ons"
2. In the "Get Add-ons" tab, search for "firebug".
3. It should be the first search result. Click Install.
4. You may have to restart Firefox for the installation to complete.



1.8 Integrated Development Environments

Although we will be using a text editor, gedit, for the majority of this course, you will eventually want to move to an IDE (Integrated Development Environment) as you begin to develop more complicated applications. Text editors are great for smaller applications; however, when you start working with bigger applications that have multiple files and tree structures, IDEs will make your development experience much easier.

Some features of popular IDEs:

- Automatic code completion
- Auto-indentation
- Internal Documentation
- Python Method Organizer
- Syntax highlighting
- Templates for common code
- Built-in support (help/tips/definitions)
- Source control support (such as mercurial or git)
- Integration with various web frameworks
- Multi-language support
- Collaborative functions
- Logical project structure
- Code output preview
- Debugging support
- Keyboard macros

If you've never used an IDE before, I'd recommend starting with the cross-platform tool, Sublime Text 2, which is part text editor, part IDE, after familiarizing yourself with gedit. (You will be introduced to Sublime in later chapters.) From there, you can move on to some of the more advanced Python IDEs, such as NetBeans, Komodo, PyScripted, PyDev, and Wing.

Finally, the web hosting platform, PythonAnywhere also doubles as an IDE. This cloud-based service allows you to write programs and web applications on their web servers, which you can easily access through your desktop computer, laptop, iPad, and iPhone. You can also collaborate with people on projects. It's a really cool environment:

**PythonAnywhere makes it easy to create and run Python programs in the cloud. You can write your programs in a web-based editor or just run a console session from any modern web browser. There's storage space on our servers, and you can preserve your session state and access it from anywhere, with no need to pay for, or configure, your own server. Start work on your work desktop, then later pick up from where you left off by accessing exactly the same session from your laptop.⁴*

We will be using PythonAnywhere later in this course when we move into developing web applications.

That's it. Let's go learn how to use Python for web development!

1. <http://www.pip-installer.org/en/latest/index.html>
2. <https://gist.github.com/mjhea0/5692708>
3. <http://www.virtualenv.org/en/1.9/>
4. <https://www.pythonanywhere.com/>

2) Fundamentals: Database Programming

2.1) SQL and SQLite Basics

A database is a structured set of data. Besides flat files, the most popular types of databases are relational databases. These organize information into tables, similar to a basic spreadsheet, with columns called fields and rows called records.

Records from one table can be linked to records in another table to create relationships.

Most relational databases use the SQL (Structured Query Language) language to communicate with the database. SQL is a fairly easy language to learn, and one worth learning. In this course I will only be providing a high-level overview to get you started. To achieve the goals of the course, you need to understand the four basic SQL commands: SELECT, UPDATE, INSERT, and DELETE.

| Command | Action |
|---------|----------------------------------|
| SELECT | retrieves data from the database |
| UPDATE | updates data from the database |
| INSERT | insert data into the database |
| DELETE | deletes data from the database |

Although SQL is a simple language, you will find an even easier way to interact with databases when we start working with web frameworks. In essence, instead of working with SQL directly, you will work with Python objects, which many Python programmers are more comfortable with. We'll cover these methods in later chapters. For now, we'll cover SQL, as it's important to understand how SQL works for when you have to troubleshoot or conduct difficult queries that require SQL.

Numerous libraries and modules are available for connecting to relational database management systems. Such systems include SQLite, MySQL, PostgreSQL, Microsoft Access, SQL Server, and Oracle. Since the language is essentially the same across these systems, choosing the one which best suits the needs of your application depends on the application's current and expected size. In this chapter, we will focus on SQLite, which is ideal for simple applications.

SQLite is great. It gives you most of the database structure of the larger, more powerful relational database systems without having to actually use a server. Again, it is ideal for simple applications as well as for testing out code. Lightweight and fast, SQLite requires little administration. It's also already included in the Python standard library. Thus, you can literally start creating and accessing databases without downloading any additional dependencies. [1](#)

Homework

- Spend thirty minutes reading more about the basic SQL commands highlighted above from the official [SQLite documentation](#). If you have time, check out W3schools.com's [Basic SQL Tutorial](#) as well. This will set the basic ground work for the rest of the chapter.
- Also, if you have access to the Real Python [course](#), brush through chapter 9 again.

2.2) Creating Tables

Let's begin. Start by activating your virtualenv.

```
# ex2.2a.py - Create a SQLite3 database and table

# import the sqlite3 library
import sqlite3

# create a new database if the database doesn't already exist
conn = sqlite3.connect("new.db")

# get a cursor object used to execute SQL commands
cursor = conn.cursor()

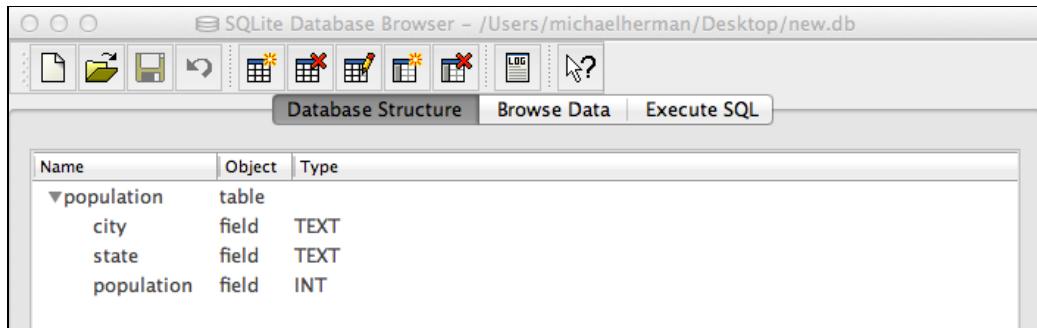
# create a table
cursor.execute("""CREATE TABLE population
                (city TEXT, state TEXT, population INT)
                """)

# close the database connection
conn.close()
```

Save the file as *ex2.2a.py* in Chapter 2's directory. Run the file from your terminal:

```
$ python ex2.2a.py
```

As long as you didn't receive an error, the database and table were created inside a new file called *new.db*. You can verify this by launching the SQLite Database Browser and then opening up the "new" database, which will be located in the same directory where you saved the file. Under the Database Structure tab you should see the "population" table. You can then expand the table and see the "city", "state", and "population" fields (also called table columns):



So what exactly did we do here?

First, we imported the `sqlite3` library to communicate with SQLite. Next, we created a new database named `new.db`. This same command is also used to connect to an existing database. Since a database didn't exist in this case, one was created for us. We then created a cursor object, which lets us execute a SQL query or command against data within the database. Finally, we created a table named "population" using the SQL statement `CREATE TABLE` that has two text fields, "city" and "state", and one integer field, "population".

You can also use the `"":memory:"` string to create a database in memory only:

```
conn = sqlite3.connect(":memory:")
```

Keep in mind, though, that as soon as you close the connection the database will disappear.

You can see that working with databases is simple! No matter what you are trying to accomplish, you will usually follow this basic workflow:

1. Create a database connection
2. Get a cursor
3. Execute your SQL query
4. Close the connection

What happens next depends on your end goal. You may insert new data (`INSERT`), modify (`UPDATE`) or delete (`DELETE`) current data, or simply extract

data in order to output it the screen or conduct analysis (SELECT). Go back and look at the SQL statements, from the beginning of the chapter, for a basic review.

Homework

- Create a new database called "cars", and add a table called "inventory" that includes the following fields: "Make", "Model", and "Quantity". Don't forget to include the proper data-types.

2.3) Inserting Data

Now that we have a table created, we can populate it with some actual data.

```
# ex2.3a.py - INSERT Command

# import the sqlite3 library
import sqlite3

# create the connection object
conn = sqlite3.connect("new.db")

# get a cursor object used to execute SQL commands
cursor = conn.cursor()

# insert data
cursor.execute("INSERT INTO population VALUES('New York City',
'NY', 8200000)")
cursor.execute("INSERT INTO population VALUES('San Francisco',
'CA', 800000)")

# commit the changes
conn.commit()

# close the database connection
conn.close()
```

Save the file as `ex2.3a.py` and then run it. Again, if you did not receive an error, then you can assume the code ran correctly. Open up the SQLite Database Browser again to ensure that the data was added. After you load the database, click the second tab, "browse data", and you should see the new values that were inserted.

As in the example from Lesson 2.2, we imported the `sqlite3` library, established the database connection, and created the `cursor` object. We then used the `INSERT INTO` SQL command to insert data into the "population" table. Note how each item has single quotes around it, while the entire statement is enclosed in double quotes. Many relational databases only allow objects to be enclosed in single quotes rather than double quotes. This can get a bit more complicated when you have items that include single quotes in them. There is a workaround, though - the `executemany()` method which you will see in the next example.

The `commit()` method executes the SQL statements and inserts the data into the table. Anytime you make changes to a table via the `INSERT`, `UPDATE`, or `DELETE` commands, you need to run the `commit()` method before you close the database connection. Otherwise, the values will only persist temporarily in memory.

That being said, if you rewrite your script using the `with` keyword, your changes will automatically be saved without having to use the `commit()` method, making your code more compact.

Let's look at the above code re-written using the `with` keyword:

```
import sqlite3
with sqlite3.connect("new.db") as connection:
    c = connection.cursor()
    c.execute("INSERT INTO population VALUES('New York City',
'NY', 8200000)")
    c.execute("INSERT INTO population VALUES('San Francisco',
'CA', 800000)")
```

Using the executemany() method

If you need to run many of the same SQL statements you can use the `executemany()` method to save time and eliminate unnecessary code. This can be helpful when initially populating a database with data.

```
# ex2.3b.py - executemany() method

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # insert multiple records using a tuple
    cities = [
        ('Boston', 'MA', 600000),
        ('Chicago', 'IL', 2700000),
        ('Houston', 'TX', 2100000),
        ('Phoenix', 'AZ', 1500000)
    ]

    # insert data into table
    c.executemany('INSERT INTO population VALUES(?, ?, ?)', cities)
```

Save the file then run it. Double check in the SQLite Database Browser that the values were added.

In this example, the question marks (?) act as placeholders (called parameterized statements) for the tuple instead of string substitution (%s). Parametrized statements should always be used when communicating with a SQL database due to potential SQL injections that could occur from using string substitutions. Essentially, a SQL injection is a fancy term for when a user supplies a value that *looks* like SQL code but really causes the SQL statement to behave in unexpected ways. Whether accidental or malicious in intent, the statement fools the database

into thinking it's a real SQL statement. In some cases, a SQL injection can reveal sensitive information or even damage or destroy the database. Be careful.

Importing data from a CSV file

In many cases, you may need to insert thousands of records into your database, in which case the data is probably contained within an external CSV file – or possibly even from a different database. Use the `executemany()` method again.

Use the `employees.csv` file for this exercise, which is located within the chapter two directory in the exercises folder.

```
# ex2.3c.py - import from CSV

# import the csv library
import csv

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # open the csv file and assign it to a variable
    employees = csv.reader(open("employees.csv", "rU"))

    # create a new table called employees
    c.execute("CREATE TABLE employees(firstname, lastname)")

    # insert data into table
    c.executemany("INSERT INTO employees(firstname, lastname)
values (?, ?)", employees)
```

2.4) Searching

Let's now look at how to retrieve data:

```
# ex2.4a.py - SELECT statement

import csv
import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # use a for loop to iterate through the database, printing
    # the results line by line
    for row in c.execute("SELECT firstname, lastname from
employees"):
        print row
```

Notice in the output the `u` character. This just stands for a Unicode string. Unicode is an international character encoding standard for displaying characters [2](#). This outputted because we printed the entire string rather than just the values.

Let's look at how to output the data with just the values by removing the unicode characters altogether:

```
# ex2.4b.py - SELECT statement, remove unicode characters

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("SELECT firstname, lastname from employees")

    # fetchall() retrieves all records from the query
    rows = c.fetchall()

    # output the rows to the screen, row by row
    for r in rows:
        print r[0], r[1]
```

First, the `fetchall()` method retrieved all records from the query and stored them as a tuple; or, more precisely: tuples within a tuple. We then assigned the records to the "rows" variable.

Finally, we printed the values using index notation, `print r[0], r[1]`.

2.5) Updating and Deleting

This lesson covers how to use the UPDATE and DELETE SQL commands.

```
# ex2.5a.py - UPDATE and DELETE statements

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # update data
    c.execute("UPDATE population SET population = 9000000 WHERE
    city='New York City'")

    # delete data
    c.execute("DELETE FROM population WHERE city='Boston'")

    print "\nNEW DATA:\n"

    c.execute("SELECT * FROM population")

    rows = c.fetchall()

    for r in rows:
        print r[0], r[1], r[2]
```

In this example, we used the UPDATE command to change a specific field from a record and the DELETE command to delete an entire record. We then displayed the results using the SELECT command. We also introduced the WHERE clause,

which is used to filter the results by a certain characteristic. You can also use this clause with the SELECT statement.

For example:

```
SELECT city from population WHERE state = 'CA'
```

This statement searches the database for cities where the state is CA. All other states are excluded from the query.

Homework

We covered a lot of material in the past few lessons. Please be sure to go over it as many times as necessary before moving on.

Use three different scripts for these homework assignments:

- Using the "inventory" table from the homework in Lesson 2.2, add 5 records (rows of data) to the table. Make sure 3 of the vehicles are Fords while the other 2 are Hondas. Use any model and quantity for each.
- Update the quantity on two of the records, and then output all of the records from the table.
- Finally output only records that are for Ford vehicles.

2.6) Working with Multiple Tables

Now that you understand the basic SQL statements - SELECT, UPDATE, INSERT, and DELETE - let's apply some of them while working with multiple tables. Before we begin, though, we need to add more records to the population table, as well as add one more table to the database.

```
# ex2.6a.py - executemany() method

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # insert multiple records using a tuple
    # (you can copy and paste the values)
    cities = [
        ('Boston', 'MA', 600000),
        ('Los Angeles', 'CA', 38000000),
        ('Houston', 'TX', 2100000),
        ('Philadelphia', 'PA', 1500000),
        ('San Antonio', 'TX', 1400000),
        ('San Diego', 'CA', 130000),
        ('Dallas', 'TX', 1200000),
        ('San Jose', 'CA', 900000),
        ('Jacksonville', 'FL', 800000),
        ('Indianapolis', 'IN', 800000),
        ('Austin', 'TX', 800000),
        ('Detroit', 'MI', 700000)
    ]

    c.executemany("INSERT INTO population VALUES(?, ?, ?)", cities)

    c.execute("SELECT * FROM population WHERE population > 1000000")

    rows = c.fetchall()

    for r in rows:
        print r[0], r[1], r[2]
```

Did you notice the WHERE clause again? In this example, we chose to limit the results by only outputting cities with populations greater than one million. Next, let's create a new table to use:

```
# ex2.6b.py - Create a table and populate it with data

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("""CREATE TABLE regions
                (city TEXT, region TEXT)
                """)

    # (again, copy and paste the values if you'd like)
    cities = [
        ('New York City', 'Northeast'),
        ('San Francisco', 'West'),
        ('Chicago', 'Midwest'),
        ('Houston', 'South'),
        ('Phoenix', 'West'),
        ('Boston', 'Northeast'),
        ('Los Angeles', 'West'),
        ('Houston', 'South'),
        ('Philadelphia', 'Northeast'),
        ('San Antonio', 'South'),
        ('San Diego', 'West'),
        ('Dallas', 'South'),
        ('San Jose', 'West'),
        ('Jacksonville', 'South'),
        ('Indianapolis', 'Midwest'),
        ('Austin', 'South'),
        ('Detroit', 'Midwest')
    ]

    c.executemany("INSERT INTO regions VALUES(?, ?)", cities)

    c.execute("SELECT * FROM regions ORDER BY region ASC")

    rows = c.fetchall()
```

```
for r in rows:  
    print r[0], r[1]
```

We created a new table called "regions" that displayed the same cities with their respective regions. Notice how we used the ORDER BY clause in the SELECT statement to display the data in ascending order by region.

Open up the SQLite Browser to double check that the new table was in fact created and populated with data.

SQL Joins

Let's write some code that will use data from both the "population" and the "regions" tables.

Code:

```
# ex2.6c.py - JOINing data from multiple tables  
  
import sqlite3  
  
with sqlite3.connect("new.db") as connection:  
    c = connection.cursor()  
  
    # retrieve data  
    c.execute("""SELECT population.city, population.population,  
              regions.region FROM population, regions  
              WHERE population.city = regions.city""")  
  
    rows = c.fetchall()  
  
    for r in rows:  
        print r[0], r[1], r[2]
```

Take a look at the SELECT statement. Since we are using two tables, fields in the SELECT statement must adhere to the following format: table_name.column_name

(i.e., `population.city`). In addition, to eliminate duplicates, as both tables include the city name, we used the WHERE clause as seen above.

Finally, let's organize the outputted results and clean up the code so it's more compact:

```
#ex2.6d.py - JOINing data from multiple tables - cleanup

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    c.execute("""SELECT DISTINCT population.city,
population.population,
regions.region FROM population, regions WHERE
population.city = regions.city ORDER by
population.city ASC""")

    rows = c.fetchall()

    for r in rows:
        print "City: " + r[0]
        print "Population: " + str(r[1])
        print "Region: " + r[2]
        print
```

Homework

- In the examples above, I made a mistake on the actual SQL statements. In essence, I JOINed the tables using Cartesian joins, which join every row from one table to every row of another table. If, for example, you have a table with 100 rows and you join it with another table with 1,000 rows, this will return 100,000 rows. You can easily crash your server using this method. Avoid this!

Instead, you will want to follow this syntax:

```
SELECT orders.orderid, clients.name, orders.date FROM
orders INNER JOIN clients ON orders.clientid =
clients.clientid;
```

Rewrite the previous examples using the correct syntax. Verify that your results are correct by comparing them to the previous examples' results.

- Add another table to accompany your "inventory" table called "orders". This table should have the following fields: "make", "model", and "order_date". Make sure to only include makes and models for the cars found in the inventory table. Add 15 records (3 for each car), each with a separate order date (YYYY-MM-DD).
- Finally output the car's make and model on one line, the quantity on another line, and then the order_dates on subsequent lines below that.

2.7 SQL Functions

SQLite has many built-in functions for aggregating and calculating data. In this lesson, we will be working with the following functions:

| Function | Result |
|----------|---|
| AVG() | Returns the average value from a group |
| COUNT() | Returns the number of rows from a group |
| MAX() | Returns the largest value from a group |
| MIN() | Returns the smallest value from a group |
| SUM() | Returns the sum of a group of values |

```
# ex2.7a.py - SQLite Functions

import sqlite3

with sqlite3.connect("new.db") as connection:
    c = connection.cursor()

    # create a dictionary of sql queries
    sql = {'average': "SELECT avg(population) FROM population",
            'maximum': "SELECT max(population) FROM population",
            'minimum': "SELECT min(population) FROM population",
            'sum': "SELECT sum(population) FROM population",
            'count': "SELECT count(city) FROM population"}

    # run each sql query item in the dictionary
    for keys, values in sql.iteritems():

        # run sql
        c.execute(values)

        # fetchone() retrieves one record from the query
        result = c.fetchone()

        # output the result to screen
        print keys + ":", result[0]
```

Essentially, we created a dictionary of SQL statements and then looped through the dictionary, executing each statement. Next, using a for loop, we printed the results of each SQL query.

Homework

- Using the COUNT() function, calculate the total number of orders for each make and model.
- Output the car's make and model on one line, the quantity on another line, and then the order count on the next line. The latter is a bit difficult, but please try it first before looking at the code. **Remember:** Google-it-first!

2.8) Example Application

We're going to end our discussion of the basic SQL commands by looking at an extended example. Please try the assignment first before reading the solution. The hardest part will be breaking it down into small bites that you can handle. You've already learned the material; we're just putting it all together. Spend some time mapping out the workflow as a first step.

In this application we will be performing aggregations on 100 integers.

Criteria:

1. Add 100 random integers, ranging from 0 to 100, to a new database called "newnum.db".
2. Prompt the user whether he or she would like to perform an aggregation (AVG, MAX, MIN, or SUM) or exit the program altogether.

Break this assignment into two scripts. Name them *assignment2a.py* and *assignment2b.py*.

Now stop for a minute and think about how you would set this up. Take out a piece of paper and *actually* write it out. Create a box for the first script and another box for the second. Write the criteria at the top of the page, and then begin by writing out exactly what the program should do in plain English in each box. These sentences will become the comments for your program.

Script 1

1. Import libraries (we need the random library because of the random variable piece):

```
import sqlite3  
import random
```

2. Establish connection, create "newnum.db" database:

```
with sqlite3.connect("newnum.db") as connection:
```

3. Open the cursor:

```
c = connection.cursor()
```

4. Create table, "numbers", with value "num" as an integer (the DROP TABLE command will remove the entire table if it exists so we can create a new one):

```
c.execute("DROP TABLE if exists numbers")
c.execute("CREATE TABLE numbers(num int)")
```

5. Use a for loop and random function to insert 100 random values (0 to 100):

```
for i in range(100):
    c.execute("INSERT INTO numbers
VALUES(?)", (random.randint(0,100),))
```

Full Code:

```
# Assignment 2a - insert random data

# import the sqlite3 library
import sqlite3
import random

with sqlite3.connect("newnum.db") as connection:
    c = connection.cursor()

    # delete database table if exist
    c.execute("DROP TABLE if exists numbers")

    # create database table
    c.execute("CREATE TABLE numbers(num int)")

    # insert each number to the database
    for i in range(100):
        c.execute("INSERT INTO numbers
VALUES (?)", (random.randint(0,100),))
```

Script 2

Code:

```
# Assignment 2b - prompt the user

# import the sqlite3 library
import sqlite3

# create the connection object
conn = sqlite3.connect("newnum.db")

# create a cursor object used to execute SQL commands
cursor = conn.cursor()

prompt = """
Select the operation that you want to perform [1-5]:
1. Average
2. Max
3. Min
4. Sum
5. Exit
"""

# loop until user enters a valid operation number [1-5]
while True:
    # get user input
    x = raw_input(prompt)

    # if user enters any choice from 1-4
    if x in set(["1", "2", "3", "4"]):
        # parse the corresponding operation text
        operation = {1: "avg", 2:"max", 3:"min",
4:"sum"}[int(x)]

        # retrieve data
        cursor.execute("SELECT {}(num) from
numbers".format(operation))

        # fetchone() retrieves one record from the query
        get = cursor.fetchone()
```

```
# output result to screen
print operation + ":  %f" % get[0]

# if user enters 5
elif x == "5":
    print "Exit"

# exit loop
break
```

We asked the user to enter the operation they would like to perform (numbers 1-4), which queried the database and displayed either the average, minimum, maximum or sum (depending on the operation chosen). The loop continues forever until the user chooses 5 to break the loop.

This chapter provided a brief summary of SQLite and how to use Python to interact with relational databases. There's a lot more you can do with databases that are not covered here. If you'd like to explore relational databases further, there are a number of great resources online, like [ZetCode](#) and [tutorialspoint](#)'s Python MySQL Database Access.

-
1. [http://www.sqlite.org/about.html ↵](http://www.sqlite.org/about.html)
 2. [http://www.unicode.org/standard/standard.html ↵](http://www.unicode.org/standard/standard.html)

3) Fundamentals: Client-Side Programming

3.1) Introduction

This chapter focuses on web client programming. Web clients are simply web browsers that access documents from other servers. Web server programming, on the other hand - covered in the next chapter - deals with, well, web servers. Put simply, when you browse the Internet, your web client (i.e., browser) sends a request to a remote server, which responds back to the web client with the requested information.

In this chapter, we will navigate the Internet using Python programs to:

- gather data,
- access and consume web services,
- scrape web pages, and
- interact with web pages.

I'm assuming you have some familiarity with HTML (HyperText Markup Language), the primary language of the Internet. If you need a quick brush up, the first 17 chapters of W3schools.com's [Basic HTML Tutorial](#) will get you up to speed quickly. They shouldn't take more than an hour to review. Make sure that at a minimum, you understand the basic elements of an HTML page such as the `<head>` and `<body>` as well as various HTML tags like `<a>`, `<div>`, `<p>`, `<h1>`, ``, `<center>`, and `
`.

These tags are used to differentiate between each section of a web site or application.

For example:

```
<h1>This is a headline</h1>
<p>This is a paragraph.</p>
<a href="http://www.realpython.com">This is a link.</a>
```

Finally, to fully explore this topic, you can gain an understanding of everything from sockets to various web protocols and become a real expert on how the Internet works. We will not be going anywhere near that in-depth in this course. Our focus, rather, will be on the higher-level functions, which are practical in nature and which can be immediately useful to a web development project. I will provide the required concepts, but it's more important to concern yourself with the actual programs and coding.

Homework

- Do you know the difference between the Internet and the Web? Did you know that there is a difference?

First, the Internet is a gigantic system of decentralized, yet interconnected computers that communicate with one another via protocols. Meanwhile, the web is what you see when you view web pages. In a sense, it's just a layer that rests on top of the Internet.

The Web is what most people think of as the Internet, which, now you know is actually incorrect.

- Read more about the differences between the Internet and the Web via Google. Look up any terminology that you have questions about, most of which we will be covering in this Chapter.

3.2) Retrieving Web Pages

The requests library is used for interacting with web pages [1](#). For straightforward situations, requests is very easy to use. You simply use the get() function, specify the URL you wish to access, and the web page is retrieved. From there, you can crawl or navigate through the web site or extract specific information.

Let's start with a basic example. But first, don't forget to activate your virtualenv.

Install requests: `pip install requests`

Code:

```
# ex3.2a.py - Retrieving a web page

import requests

# retrieve the web page
r = requests.get("http://www.python.org/")

print r.content
```

As long as you are connected to the Internet this script will pull the HTML source code from the Python Software Foundation's website and output it to the screen. It's a mess, right? Can you recognize the header (`<head> </head>`)? How about some of the other basic HTML tags?

Let me show you an easier way to look at the full HTML output.

Code:

```
# ex3.2b.py - Downloading a web page

import requests

r = requests.get("http://www.python.org/")

# write the content to test_request.html
with open("test_requests.html", "wb") as code:
    code.write(r.content)
```

Save the file as `ex3.2b.py` and run it. If you don't see an error, you can assume that it ran correctly. Open the new file, `test_requests.html` in gedit. Now it's much easier

to examine the actual HTML. Go through the file and find tags that you recognize. Google any tags that you don't. This will help you later when we start web scraping.

"wb" stands for write binary, which downloads the raw bytes of the file. Thus, the file is downloaded in its exact format.

Did you notice the get() function in those last two programs? Computers talk to one another via HTTP methods. The two methods you will use the most are GET and POST. When you view a web page, your browser uses GET to fetch that information. When you submit a form online, your browser will POST information to a web server. Make them your new best friends. *More on this later.*

Let's look at an example of a POST request.

Code:

```
# ex3.2d.py - Submitting to a web form

import requests

url = 'http://httpbin.org/post'
data = {'fname': 'Michael', 'lname': 'Herman'}

# submit post request
r = requests.post(url, data=data)

# display the response to screen
print r
```

Output:

```
<Response [200]>
```

Using the requests library, you created a dictionary with the field names as the keys `fname` and `lname`, associated with values `Michael` and `Herman` respectively.

`requests.post` initiates the POST request. In this example, you used the website httpbin.org, which is specifically designed to test HTTP requests and received a response back in the form of a code, called a status code.

Common status codes:

- 200 OK
- 300 Multiple Choices
- 301 Moved Permanently
- 302 Found
- 304 Not Modified
- 307 Temporary Redirect
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 410 Gone
- 500 Internal Server Error
- 501 Not Implemented
- 503 Service Unavailable
- 550 Permission denied

There are actually many [more](#) status codes that mean various things, depending on the situation.

You should have received a "200" response to the POST request above.

Modify the script to see the entire response by appending `.content` to the end of the print statement:

```
print r.content
```

You should see the data you sent within the response:

```
"form": {  
    "lname": "Herman",  
    "fname": "Michael"  
},
```

3.3) Web Services Defined

Web services can be a difficult subject to grasp. Take your time with this. Learn the concepts in order to understand the exercises. Doing so will make your life as a web developer much easier. In order to understand web services, you first need to understand APIs.

APIs

An API (Application Programming Interfaces) is a type of protocol used as a point of interaction between two independent applications with the goal of exchanging data. Protocols define the type of information that can be exchanged. In other words, APIs provide a set of instructions, or rules, for applications to follow while accessing other applications.

One example that you've already seen is the SQLite API, which defines the SELECT, INSERT, UPDATE, and DELETE requests discussed in the last chapter. The SQLite API allows the end user to perform certain tasks, which, in general, are limited to those four functions.

HTTP APIs

HTTP APIs, also called web services and web APIs, are simply APIs made available over the Internet, used for reading (GET) and writing (POST) data. GET and POST, as well as UPDATE and DELETE, along with SELECT, INSERT, UPDATE, and DELETE are all forms of CRUD:

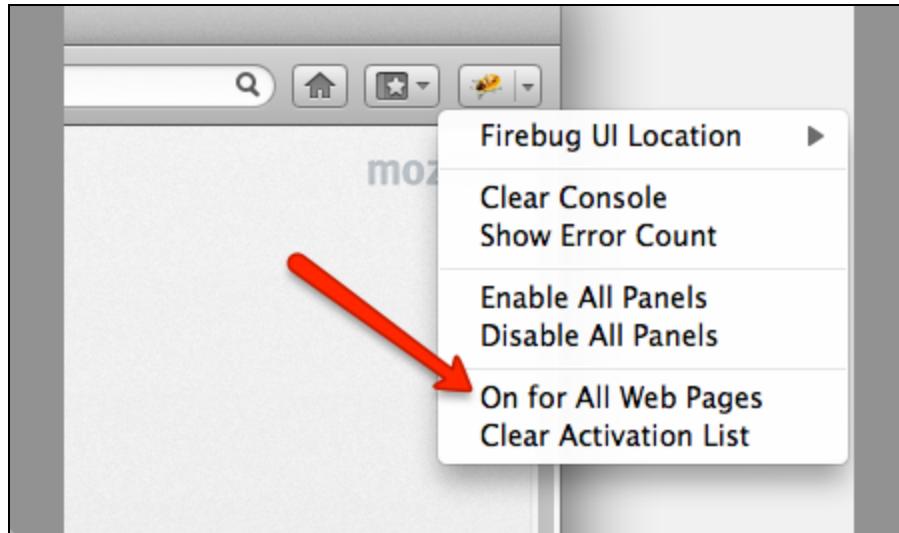
| Operation (CRUD) | HTTP | SQL |
|------------------|--------|--------|
| CREATE | POST | INSERT |
| READ | GET | SELECT |
| UPDATE | PUT | UPDATE |
| DELETE | DELETE | DELETE |

So, the SELECT command, for example, is equivalent to the GET HTTP method, which corresponds to the Read CRUD Operation.

Anytime you browse the Internet, you are constantly sending HTTP requests. For example, WordPress reads (GETs) data from Facebook to show how many people have "liked" a blog article. Then, if you "liked" an article, data is sent (POST) to Facebook (if you allow it), showing that you liked that article. Without web services, these interactions between two independent applications would be impossible.

Let's look at an example in Firebug:

1. Open FireFox
2. Click on the drop down arrow in the top right of the screen, next to the Firebug logo
3. Select the option "On for all Web Pages":



4. Then click on the Firebug logo itself to open the Firebug console
5. With the console open, click the Net panel, and set it to enabled.
6. Now, navigate in your browser to a site you need to login at. Watch the activity in your console. Do you see the GET requests? The most important one is the top one, which loaded the web page.
7. Now click the tab to Log In. If you're already logged in, go ahead and log out. Enter your login credentials. Pay close attention to the console, and click "Log In".
8. Did you see the POST request? It was only there for a seconds. Basically, you sent a POST request with your login credentials to the server.

Check out some other web pages. Try logging in to some of your favorite sites to see more POST requests. Perhaps POST a comment on a blog or message forum.

Applications can access APIs either directly, through the API itself, or indirectly, through a client library. The best means of access depends on a number of factors. Access through client libraries can be easier, especially for beginners, as the code is already written. However, you still have to learn how the client library works and integrate the library's code base into your overall code. Also, if you do not first take

the time to learn how the client library works, it can be difficult to debug or troubleshoot. Direct access provides greater control, but beginners may encounter more problems understanding and interpreting the rules of the specific API.

We will be looking at both methods.

Not all web services rely on HTTP requests to govern the allowed interaction. Only RESTful APIs use POST, GET, PUT, and DELETE. This confuses a lot of developers. Just remember that web RESTful APIs, or HTTP APIs, are just one type of web service.

Summary

In summary, APIs:

- Facilitate the exchange of information,
- Speak a common language, and
- Can be accessed either directly or indirectly through client libraries.

Although web services have brought much order to the Internet, the services themselves are still fairly chaotic. There are no standards besides a few high-level rules, REST, associated with HTTP requests. Documentation is a big problem too, as it is left to the individual developers to document how their web services work. If you start working more with web services, which I encourage you to do so, you will begin to see not only just how different each and every API is but also how terribly documented many of them are.

If you'd like more information on web APIs, check out [this](#) great crash course from Codecademy.

Fortunately, data exchanged via web services is standardized in text-based formats and thus, are both human and machine-readable. Two popular formats used today are XML and JSON, which we will address in the next two chapters.

Homework

- Read [this](#) article providing a high-level overview of standards associated with RESTful APIs.

3.4) Working with XML

XML (eXtensible Markup Language) is a highly structured language, designed specifically for transferring information. The rigid structure makes it perfect for reading and interpreting the data (called parsing) found within an XML file. It's both human and machine-readable.

Let's look at an example of an XML file:

```
<?xml version="1.0"?>
<CARS>
    <CAR>
        <MAKE>Ford</MAKE>
        <MODEL>Focus</MODEL>
        <COST>15000</COST>
    </CAR>
    <CAR>
        <MAKE>Honda</MAKE>
        <MODEL>Civic</MODEL>
        <COST>20000</COST>
    </CAR>
    <CAR>
        <MAKE>Toyota</MAKE>
        <MODEL>Camry</MODEL>
        <COST>25000</COST>
    </CAR>
    <CAR>
        <MAKE>Honda</MAKE>
        <MODEL>Accord</MODEL>
        <COST>22000</COST>
    </CAR>
</CARS>
```

There's a declaration at the top, and the data is surrounded by opening and closing tags. One useful thing to remember is that the purpose of XML is much different than HTML. While HTML is used for displaying data, XML is used for transferring data. In itself, an XML document is purposeless until it is read, understood, and parsed by an application. It's about what you *do* with the data that matters.

With that in mind, let's build a quick parser. There are quite a few libraries you can use to read and parse XML files. One of the easiest libraries to work with is the `ElementTree` library, which is part of Python's standard library. Use the `cars.xml` file found in your Chapter 3 folder with this example.

Code:

```
# ex3.4a.py - XML Parsing 1

from xml.etree import ElementTree as et

# parses the file
doc = et.parse("cars.xml")

# outputs the first MODEL in the file
print doc.find("CAR/MODEL").text
```

Output:

Focus

In this program you read and parsed the file using the `find` function and then outputted the data between the first `<MODEL>` `</MODEL>` tags. These tags are called element nodes, and are organized in a tree-like structure and further classified into parent and child relationships.

In the example above, the parent is `<CARS>`, and the child elements are `<CAR>`, `<MAKE>`, `<MODEL>`, and `<COST>`. The `find` function begins looking for elements that are children of the parent node, which is why we started with the first child when we outputted the data, rather than the parent element:

```
print doc.find("CAR/MODEL").text
```

The above line is equivalent to:

```
print doc.find("CAR[1]/MODEL").text
```

See what happens when you change the code in the program to:

```
print doc.find("CAR[2]/MODEL").text
```

The output should be:

```
Civic
```

See how easy that was. That's why XML is both machine *and* human readable.

Let's take it a step further and add a loop to extract all the data.

Code:

```
# ex3.4b.py - XML Parsing 2

from xml.etree import ElementTree as et

doc = et.parse("cars.xml")

# outputs the make, model and cost of each car to the screen
for element in doc.findall("CAR"):
    print (element.find("MAKE").text + " " +
           element.find("MODEL").text +
           ", $" + element.find("COST").text)
```

You should get the following results:

```
Ford Focus, $15000
Honda Civic, $20000
Toyota Camry, $25000
Honda Accord, $22000
```

This program follows the same logic as the previous one, but you just added a FOR loop to iterate through the XML file, pulling all the data.

In this last example, we will use a GET request to access XML found on the web.

Code:

```
# ex3.4c.py - XML Parsing 3

from xml.etree import ElementTree as et
import requests

# retrieve an xml document from a web server
xml = requests.get("http://www.w3schools.com/xml/
cd_catalog.xml")

with open("test.xml", "wb") as code:
    code.write(xml.content)

doc = et.parse("test.xml")

# outputs the album, artist and year of each CD to the screen
for element in doc.findall("CD"):
    print "Album: ", element.find("TITLE").text
    print "Artist: ", element.find("ARTIST").text
    print "Year: ", element.find("YEAR").text, "\n"
```

Again, this program follows the same logic. You just added an additional step by importing the requests library and downloading the XML file before reading and parsing the XML.

3.5) Working with JSON

JSON (JavaScript Object Notation) is a lightweight format used for transferring data. Like XML, it's both human and machine readable, which makes it easy to generate and parse, and it's used by thousands of web services. Its syntax differs from XML though, which many developers prefer because it's faster and takes up less memory. Because of this, JSON is becoming the format of choice for web services. It's derived from JavaScript and, as you will soon see, resembles a more complex Python dictionary.

Let's look at a quick example:

```
{
    "CARS": [
        {
            "MAKE": "Ford",
            "MODEL": "Focus",
            "COST": "15000"
        },
        {
            "MAKE": "Honda",
            "MODEL": "Civic",
            "COST": "20000"
        },
        {
            "MAKE": "Toyota",
            "MODEL": "Camry",
            "COST": "25000"
        },
        {
            "MAKE": "Honda",
            "MODEL": "Accord",
            "COST": "22000"
        }
    ]
}
```

Although the data looks very similar to XML, there are many noticeable differences. There's less code, no start or end tags, and it's easier to read. Also, because JSON operates much like a Python dictionary, it is very easy to work with with Python.

Basic Syntactical rules:

1. Data is found in key/value pairs (i.e., `"MAKE": "FORD"`).
2. Data is separated by commas.
3. The curly brackets contain dictionaries, while the square brackets hold lists.

JSON decoding is the act of taking a JSON file, parsing it, and turning it into something usable.

Without further ado, let's look at how to decode and parse a JSON file. Use the `cars.json` file found in your Chapter 3 folder for this example.

Code:

```
# ex3.5a.py - JSON Parsing 1

import json

# decodes the json file
output = json.load(open('cars.json'))

# display output to screen
print output
```

Output:

```
[{u'CAR': [{u'MAKE': u'Ford', u'COST': u'15000', u'MODEL':
u'Focus'}, {u'MAKE': u'Honda', u'COST': u'20000', u'MODEL':
u'Civic'}, {u'MAKE': u'Toyota', u'COST': u'25000', u'MODEL':
u'Camry'}, {u'MAKE': u'Honda', u'COST': u'22000', u'MODEL':
u'Accord'}}}]
```

You see we have four dictionaries inside a list, enclosed within another dictionary, which is finally enclosed within another list. *Repeat that to yourself a few times.* Can you see that in the output?

If you want to print just the value "Focus" of the "MODEL" key within the first dictionary in the list, you could run the following code:

```
# ex3.5b.py - JSON Parsing 2

import json

# decodes the json file
output = json.load(open('cars.json'))

# display output to screen
print output[0]["CAR"][0]["MODEL"]
```

Let's look at the `print` statement in detail:

1. `[0]["CAR"]` - indicates that we want to find the first car dictionary. Since there is only one, there can only be one value - `0`.
2. `[0]["MODEL"]` - indicates that we want to find the first instance of the `model` key, and then extract the value associated with that key. If we changed the number to `1`, it would find the second instance of `model` and return the associated value: `Civic`.

Finally, let's look at how to POST JSON to an API.

Code:

```
# ex3.5d.py - POST JSON Payload

import json
import requests

url = "http://httpbin.org/post"
payload = {"colors": [
    {"color": "red", "hex": "#f00"}, 
    {"color": "green", "hex": "#0f0"}, 
    {"color": "blue", "hex": "#00f"}, 
    {"color": "cyan", "hex": "#0ff"}, 
    {"color": "magenta", "hex": "#f0f"}, 
    {"color": "yellow", "hex": "#ff0"}, 
    {"color": "black", "hex": "#000"}]}
headers = {"content-type": "application/json"}

# post data to a web server
response = requests.post(url, data=json.dumps(payload),
                           headers=headers)

# output response to screen
print response.status_code
```

Output:

```
200 OK
```

In some cases you will have to send data (also called a Payload), to be interpreted by a remote server to perform some action on your behalf. For example, you could send a JSON Payload to Twitter with a number Tweets to be posted. I have seen situations where in order to apply for certain jobs, you had to send a Payload with your name, telephone number, and a link to your online resume.

If you ever run into a similar situation, make sure to test the Payload before actually sending it to the real URL. You can use sites like [JSON Test](#) or [Echo JSON](#) for testing purposes.

3.6) Working with Web Services

Now that you've seen how to communicate with web services (via HTTP methods) and how to handle the resulting data (either XML or JSON), let's look at some examples.

Youtube

Google, the owner of YouTube, has been very liberal when it comes to providing access to their data via web APIs, allowing hundreds of thousands of developers to create their own applications.

Download the GData library for this next example: `pip install gdata` [2](#)

Code:

```
# ex3.6a.py - Basic web services example

# import the client libraries
import gdata.youtube
import gdata.youtube.service

# YouTubeService() used to generate the object so that we can
# communicate with the YouTube API
youtube_service = gdata.youtube.service.YouTubeService()

# prompt the user to enter the Youtube User ID
playlist = raw_input("Please enter the user ID: ")

# setup the actual API call
url = "http://gdata.youtube.com/feeds/api/users/"
playlist_url = url + playlist + "/playlists"

# retrieve Youtube playlist
video_feed =
youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)

print "\nPlaylists for " + str.format(playlist) + ":\n"

# display each playlist to screen
for p in video_feed.entry:
    print p.title.text
```

Test the program out with my Youtube ID, "hermanmu". You should see a listing of my Youtube playlists.

In the above code, we started by importing the required libraries, which are for the Youtube Python client library. We then established communication with Youtube, prompted the user for a user ID, and then made the API call to request the data.

You can see that the data was returned to the variable `video_feed`, which we looped through and pulled out certain values. Let's take a closer look.

Comment out the loop and just output the `video_feed` variable. Look familiar? You should be looking at an XML file. It's difficult to read, though. So, how did I know which elements to extract? Well, I went and looked at the API [documentation](#).

Navigate to that URL.

First, you can see the URL for connecting with (calling) the API and extracting a user's information with regard to playlists:

```
http://gdata.youtube.com/feeds/api/users/username/playlists
```

Try replacing `username` with my `username`, `hermanmu`, then navigate to this URL in your browser. You should see the same XML file that you did just a second ago when you printed just the `video_feed` variable.

Again, this is such a mess we can't read it. At this point, we could download the file, like we did in lesson 2.2, to examine it. But first, let's look at the documentation some more. Perhaps there's a clue there.

Scroll down to the "Retrieving playlist information", and look at the code block. You can see that there is sample code there for iterating through the XML file:

```
# iterate through the feed as you would with any other
for playlist_video_entry in playlist_video_feed.entry:
    print playlist_video_entry.title.text
```

With enough experience, you will be able to just look at this and know that all you need to do is append `title` and `text`, which is what I did in the original code to obtain the required information:

```
for p in video_feed.entry:
    print p.title.text
```

For now, while you're still learning, you have one of two options:

1. Trial and error, or

2. Download the XML file using the requests library and examine the contents

Always look at the documentation first. You will usually find something to work with, and again this is how you learn. Then if you get stuck, use the "Google-it-first" philosophy, then if you still have trouble, go ahead and download the XML.

By the way, one of my readers helped with developing the following code to pull not only the title of the playlist but the list of videos associated with each playlist as well.

```
# import the client libraries
import gdata.youtube
import gdata.youtube.service

# YouTubeService() used to generate the object so that we can
# communicate with the YouTube API
youtube_service = gdata.youtube.service.YouTubeService()

# prompt the user to enter the Youtube User ID
user_id = raw_input("Please enter the user ID: ")

# setup the actual API call
url = "http://gdata.youtube.com/feeds/api/users/"
playlist_url = url + user_id + "/playlists"

# retrieve Youtube playlist and video list
playlist_feed =
youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
print "\nPlaylists for " + str.format(user_id) + ":\n"

# display each playlist to screen
for playlist in playlist_feed.entry:
    print playlist.title.text
    playlistid = playlist.id.text.split('/')[-1]
    video_feed =
youtube_service.GetYouTubePlaylistVideoFeed(playlist_id =
playlistid)
    for video in video_feed.entry:
        print "\t"+video.title.text
```

Does this make sense? Notice the nested loops. What's different about this code?

Twitter

Like Google, Twitter provides a very open API. I use the Twitter API extensively for pulling in tweets on specific topics, then parsing and extracting them to a CSV file for analysis. One of the best client libraries to use with the Twitter API is Tweepy:

```
pip install tweepy .3
```

Before we look at example, you need to obtain access codes for authentication.

1. Navigate to <https://dev.twitter.com/apps>
2. Click the button to create a new application
3. Enter dummy data
4. After you register, you will be taken to your application where you will need the following access codes:
 - consumer_key
 - consumer_secret
 - access_token
 - access_secret
5. Make sure to create your access token to obtain the access_token and access_secret.

Now, let's look at an example:

```
# ex3.6b.py

import tweepy

consumer_key = "<get_your_own>"
consumer_secret = "<get_your_own>"
access_token = "<get_your_own>"
access_secret = "<get_your_own>

auth = tweepy.auth.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_secret)
api = tweepy.API(auth)

tweets = api.search(q='web2py')

# display results to screen
for t in tweets:
    print t.created_at, t.text, "\n"
```

If done correctly, this should output the tweets and the dates and times they were created. Essentially, the search results were returned to a list, and then you used a For loop to iterate through that list to extract the desired information.

How did I know I wanted the keys `created at` and `text`? Again, trial and error. I started with the [documentation](#), then I did some Google searches on my own. You will become quite adept at knowing the types of reliable sources to use when trying to obtain information about an API. Try this on your own. See what other information you can extract.

Google Directions API

With the Google Directions API, you can obtain directions between two points for a number of different modes of transportation. Start by looking at the documentation [here](#). In essence, the documentation is split in two. The first part describes the type of information you can obtain from the API, and the second part details how to obtain said information.

Let's get walking directions from Central Park to Times Square..

1. Use the following URL to call (or evoke) the API:

```
http://maps.googleapis.com/maps/api/directions/  
output?parameters
```

2. You then must specify the output. We'll use JSON since it's easier to work with.
3. Also, you must specify some parameters. Notice in the documentation how some parameters are required while others are optional. Let's use these parameters:

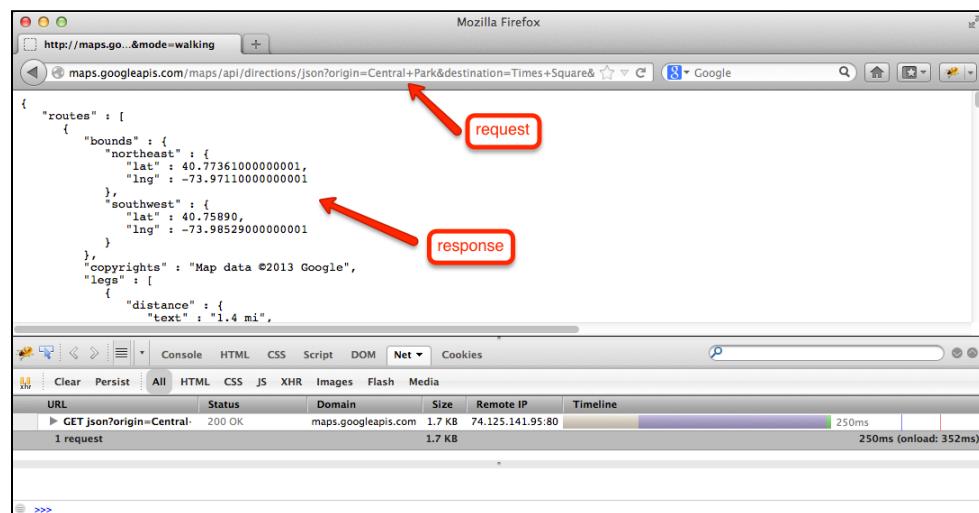
```
origin=Central+Park  
destination=Times+Square  
sensor=false  
mode=walking
```

4. You could simply append the output as well as the parameters to the end of

the URL - `http://maps.googleapis.com/maps/api/directions/`

`json?origin=Central+Park&destination=Times+Square&sensor=false&mode=walking`

- and then call the API directly from your browser:



However, there's a lot more information there than we need. Let's call the API directly from the Python Shell, and then extract the actual driving directions:

```
Desktop — Python — 117x28
>>> import json, requests
>>> url = "http://maps.googleapis.com/maps/api/directions/json?origin=Central+Park&destination=Times+Square&sensor=false&mode=walking"
>>> data = requests.get(url)
>>> binary = data.content
>>> output = json.loads(binary)
>>> print output['status']
OK
>>> for route in output['routes']:
...     for leg in route['legs']:
...         for step in leg['steps']:
...             print step['html_instructions']
...
Head <b>south</b> on <b>The Mall</b>
Turn <b>right</b> toward <b>Central Park Driveway</b>
Turn <b>left</b> toward <b>Central Park Driveway</b>
Slight <b>right</b> toward <b>Central Park Driveway</b>
Turn <b>left</b> toward <b>Central Park Driveway</b>
Turn <b>right</b> toward <b>Central Park Driveway</b>
Slight <b>left</b> onto <b>Central Park Driveway</b>
Turn <b>right</b> toward <b>Central Park Driveway</b>
Turn <b>left</b> onto <b>Central Park Driveway</b>
Turn <b>left</b> onto <b>West Dr</b>
Turn <b>right</b> to stay on <b>West Dr</b>
Continue onto <b>7th Ave</b>
Turn <b>right</b> onto <b>W 47th St</b>
Turn <b>left</b> onto <b>Broadway</b><div style="font-size:0.9em">Destination will be on the left</div>
>>>
```

All right. Let's breakdown the for loops:

```

for route in output['routes']:
    for leg in route['legs']:
        for step in leg['steps']:
            print step['html_instructions']

```

Compare the loops to the entire output. You can see that for each loop we're just moving in one level:

```

{
  "routes": [
    {
      "bounds": {
        "northeast": {
          "lat": 40.77361000000001,
          "lng": -73.97110000000001
        },
        "southwest": {
          "lat": 40.75890,
          "lng": -73.98529000000001
        }
      },
      "copyrights": "Map data ©2013 Google",
      "legs": [
        {
          "distance": {
            "text": "1.4 mi",
            "value": 2285
          },
          "duration": {
            "text": "28 mins",
            "value": 1676
          },
          "end_address": "Times Square, 1560 Broadway #800, New York, NY 10036, USA",
          "end_location": {
            "lat": 40.75890,
            "lng": -73.98529000000001
          },
          "start_address": "Central Park, 14 East 60th Street, New York, NY 10022, USA",
          "start_location": {
            "lat": 40.77361000000001,
            "lng": -73.97110000000001
          },
          "steps": [
            {
              "distance": {
                "text": "0.2 mi",
                "value": 328
              },
              "duration": {
                "text": "4 mins",
                "value": 241
              },
              "end_location": {
                "lat": 40.77079000000001,
                "lng": -73.97220
              },
              "html_instructions": "Head \u003cb\u003esouth\u003c/b\u003e on \u003cb\u003eThe Mall\u003c/b\u003e"
            }
          ]
        }
      ]
    }
  ]
}

```

So, if I wanted to print the `start_address` and `end_address`, I would just need two for loops:

```

for route in output['routes']:
    for leg in route['legs']:
        print leg['start_address']
        print leg['end_address']

```

Homework

- Using the Google Direction API, pull driving directions from San Francisco to Los Angeles in XML. Extract the step-by-step driving directions.

3.7) Rotten Tomatoes API

Before moving on to web scraping, let's look at an extended example of how to use web services to obtain information. In the last lesson we used client libraries to connect with APIs; in this lesson we'll establish a direct connection. You'll grab data (GET) from the Rotten Tomatoes API, parse the relevant info, then upload the data to a SQLite database.

Start by navigating to this URL in your browser:

<http://developer.rottentomatoes.com/docs/read/Home> ⁴

Whenever you start working with a new API, you always, always, ALWAYS want to start with the documentation. Again, all APIs work a little differently because few universal standards or practices have been established. Fortunately, the Rotten Tomatoes API is not only well documented but also easy to read and follow.

In this example, we want to grab a list of all movies currently playing in theaters. According to the documentation, we need to register for an API key to access the API. We then can grab the needed data based on a series of endpoints. Endpoints are the actual connection points for accessing the data. In other words, they are the specific URLs used for calling an API to GET data. Each endpoint is generally associated with a different type of data, which is why endpoints are often associated with groupings of data (e.g., movies playing in the theater, movies opening on a certain date, top rentals, and so on). Go ahead and click the link for "In Theaters Movies". Immediately, you'll see the URL (or endpoint):

```
http://api.rottentomatoes.com/api/public/v1.0/lists/movies/  
in_theaters.json?apikey=[your_api_key]
```

You use this URL to connect with the API.

Notice how you need an API key to access the API. The majority of web APIs require users to go through some form of authentication in order to access their services. There are a number of different means of going through authentication. It's less important that you understand how each method works than to understand *how* to obtain authentication to access the web service. Always refer to the web service provider's documentation to obtain this information.

In this particular case, we just need to register for a developer's key. To register, click the "Register" link in the top right corner of the web page. It's fairly straightforward. Make sure to enter a working email address, as the authentication key will be emailed to you. When you get to the point in the process where you're asked the name and URL of the application you are creating, simply enter dummy data:

The screenshot shows a registration form with a title bar 'Register Your New Application'. Below it are two input fields. The first field is labeled 'Name of your application (you can change it later)' and contains the text 'this is top secret'. The second field is labeled 'Application URL' and contains the text 'http://its-a-secret.com'.

Once you register for a key, DO NOT share it with anyone. You do not want someone else using that key to obtain information and possibly use it in either an illegal or unethical manner.

Once you have your key, go ahead and test it out. Use the URL from above and replace "[your_api_key]" with the generated key. Now test it in your browser. You should see a large JSON file full of data. If not, there may be a problem with your key. **Make sure you copied and pasted the entire key and appended it correctly to the URL.** If you continue having problems, post a message to the Real Python [message forum](#).

Now comes the fun part: building the program to actually GET the data, parsing the relevant data, and then dumping it into a database.

Let's start by writing a script to create the database and table.

Code:

```
# ex3.7a.py - Create a SQLite3 database and table

import sqlite3

with sqlite3.connect("movies.db") as connection:
    c = connection.cursor()

    # create a table
    c.execute("""CREATE TABLE new_movies
                (title TEXT, year INT, rating text,
                 release text, runtime INT,
                 critics_review INT,
                 audience_review INT)""")
```

We need one more script now to pull the data again and dump it directly to the database:

```
# ex3.7b.py - GET data from Rotten Tomatoes, parse, and write
# to database

import json
import requests
import sqlite3

YOUR_OWN_KEY = 'GET_YOUR_OWN_KEY'
url = requests.get("http://api.rottentomatoes.com/api/public/
v1.0/lists/movies/in_theaters.json?apikey=%s" %
(YOUR_OWN_KEY,))

# convert data from feed to binary
binary = url.content

# decode the json feed
output = json.loads(binary)

# grab the list of movies
movies = output["movies"]

with sqlite3.connect("movies.db") as connection:
    c = connection.cursor()

    # iterate through each movie and write to the database
    for movie in movies:
        c.execute("INSERT INTO new_movies VALUES(?, ?, ?, ?, ?, ?, ?, ?)",
                  (movie["title"], movie["year"],
                   movie["mpaa_rating"],
                   movie["release_dates"]["theater"],
                   movie["runtime"],
                   movie["ratings"]["critics_score"],
                   movie["ratings"]["audience_score"]))

    # retrieve data
    c.execute("SELECT * FROM new_movies ORDER BY title ASC")
```

```
# fetchall() retrieves all records from the query
rows = c.fetchall()

# output the rows to the screen, row by row
for r in rows:
    print r[0], r[1], r[2], r[3], r[4], r[5], r[6]
```

We grabbed the data URL with a GET request, converted the data to binary, decoded the JSON feed, then used a for loop to write the data to the database, and finally we grabbed the data. Nice, right?

3.8) Web Scraping and Crawling with Scrapy

Since data is not always accessible through a manageable format (i.e., XML or JSON), via web APIs, we sometimes need to get our hands dirty to access the data that we need. So, we need to turn to web scraping.

Web scraping is an automated means of retrieving data from a web page. Essentially, we grab unstructured HTML and parse it into usable data that Python can work with. Most web page owners and many developers do not view scraping in the highest regard. The question of whether it's illegal or not often depends on *what* you do with the data, not the actual act of scraping. If you scrape data from a commercial website, for example, and resell that data, there could be serious legal ramifications. The act of actual scraping, if done ethically, is *generally* not illegal, if you use the data for your own personal use.

That said, most developers will tell you to follow these two principles:

1. Adhere to ethically scraping practices by not making numerous repeated requests to a website's server, which may use up bandwidth, slowing down the website for other users and potentially overloading the server; and
2. Always check a website's acceptable use policy before scraping its data to see if accessing the website by using automated tools is a violation of its terms of use or service.

Example terms of service from Ebay, explicitly banning scraping: [5](#)

Access and Interference

eBay's sites contain robot exclusion headers. Information on our sites is subject to constant updates and changes. Much of the information on the sites is also proprietary or is licensed to eBay by our users or third parties. You agree that you will not use any **robot, spider, scraper**, or other automated means to access our sites for any purpose without our express handwritten permission.

Additionally, you agree that you will not:

- take any action that imposes or may impose (to be determined in our sole discretion) an unreasonable or disproportionately large load on our infrastructure;
- copy, reproduce, reverse engineer, modify, create derivative works from, distribute, or publicly display any content (except for your information) from our sites, services, applications, or tools without the prior express written permission of eBay and the appropriate third party, as applicable;
- interfere or attempt to interfere with the proper working of our sites, services, applications, or tools, or any activities conducted on or with our sites, services, applications, or tools; or
- bypass our robot exclusion headers or other measures we may use to prevent or restrict access to our sites.

It's absolutely vital to adhere to ethical scraping. You could very well get yourself banned from a website if you scrape millions of pages using a loop. With regard to the second principle, there is much debate about whether accepting a website's terms of use is a binding contract or not. This is not a course on ethics or law, though. So, the examples covered will adhere to **both** principles.

Finally, it's also a good idea to check the *robots.txt* file before scraping or crawling. Found in the root directory of a web site, *robots.txt* establishes a set of rules, much like a protocol, that web crawlers or robots *should* adhere to.

Let's look at an example. Navigate to the HackerNews' *robots.txt* file:

<https://news.ycombinator.com/robots.txt>:

```
User-Agent: *
Disallow: /x?
Disallow: /vote?
Disallow: /reply?
Disallow: /submitted?
Disallow: /submitlink?
Disallow: /threads?
Crawl-delay: 30
```

1. The User-Agent is the robot, or crawler, itself. Nine times out of ten you will see a wildcard * used as the argument, specifying that *robots.txt* applies to all robots.
2. Disallow parameters establish the directories or files - "Disallow: /folder/" or "Disallow: /file.html" - that robots must avoid.
3. The Crawl-delay parameter is used to indicate the minimum delay (in seconds) between successive server requests. So, in the HackerNews' example, after scraping the first page, a robot must wait thirty seconds before crawling to the next page and scraping it, and so on.

Regardless of whether a Crawl-delay is established or not, it's good practice to wait five to ten seconds between each request to avoid putting unnecessary load on the server. Again, exercise caution. You do not want to get banned from a site.

And with that, let's start scraping.

There are a number of great libraries you can use for extracting data from websites. If you are new to web scraping, start with [Beautiful Soup](#). It's easy to learn, simple to use, and the documentation is great. That being said, there are plenty of examples of using Beautiful Soup in the original Real Python [course](#).
Start there. We're going to be looking at a more advanced library called Scrapy.⁶

Let's get scrapy installed: `pip install Scrapy`

If you are using Windows there are additional steps and dependencies that you need to install. Please follow the video accompanying this lesson for details.

HackerNews (BaseSpider)

In this first example, let's scrape [HackerNews](#).

Once Scrapy is installed, open your terminal and navigate to your Chapter 3 directory, and then start a new Scrapy project: `scrapy startproject hackernews`

This will create a "hackernews" directory with the following contents:

```
hackernews/
    scrapy.cfg
    hackernews/
        __init__.py
        items.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
```

In this basic example, we're only going to worry about the `items.py` file and creating a spider, which is the actual Python script used for scraping.

First, open up the `items.py` file and edit it to define the fields that you want extracted. Let's grab the title and url of each posting:

```
from scrapy.item import Item, Field

class HackernewsItem(Item):
    # define the fields for your item here like:
    title = Field()
    url = Field()
```

Now, let's create the actual spider:

```
# spider.py

from scrapy.spider import BaseSpider
from scrapy.selector import HtmlXPathSelector
from hackernews.items import HackernewsItem

class MySpider(BaseSpider):

    # name the spider
    name = "hackernews"

    # allowed domains to scrape
    allowed_domains = ["news.ycombinator.com/"]

    # urls the spider begins to crawl from
    start_urls = ["https://news.ycombinator.com/"]

    # parses and returns the scraped data
    def parse(self, response):
        hxs = HtmlXPathSelector(response)
        titles = hxs.select('//td[@class="title"]')
        items = []
        for title in titles:
            item = HackernewsItem()
            item["title"] = title.select("a/text()").extract()
            item["url"] = title.select("a/@href").extract()
            items.append(item)
        return items
```

Save the file as `spider.py` in the "spiders" directory (`/hackernews/hackernews/spiders/`). Then, navigate to the main directory (`/hackernews/hackernews`) and run the following command: `scrapy crawl hackernews`. This will scrape all the data to the screen. If you want to create a CSV so the parsed data is easier to read, run this command instead: `scrapy crawl hackernews -o items.csv -t csv`.

All right. So what's going on here?

Essentially, you used XPath to parse and extract the data using HTML tags:

1. `//td[@class="title"]` - finds all `<td>` tags where `class="title"`.
2. `a/text` - finds all `<a>` tags within each `<td>` tag, then extracts the text
3. `a/@href` - again finds all `<a>` tags within each `<td>` tag, but this time it extracts the actual url

How did I know which HTML tags to use?

1. Open the start url in Firefox: <https://news.ycombinator.com/>
2. Right click on the first article link and select "Inspect Element"
3. In the Firebug console, you can see the HTML that's used to display the first link:

```

Y Hacker News new | comments | ask | jobs | submit
1. ▲ The Handshake Deal Protocol (ycombinator.com)
185 points by pg 1 hour ago | 104 comments
2. ▲ Bulletproof demos using Chrome's playback mode (hubspot.com)
100 points by zackblum 2 hours ago | 17 comments

<html>
  <head>
  <body>
    <center>
      <table width="85%" cellspacing="0" cellpadding="0" border="0" bgcolor="#f6f6ef">
        <tbody>
          <tr>
            <td>
              <table cellspacing="0" cellpadding="0" border="0">
                <tbody>
                  <tr>
                    <td class="title" valign="top" align="right">1.</td>
                    <td>
                      <a href="http://ycombinator.com/hdp.html">The Handshake Deal Protocol </a>
                      <span class="comhead"> (ycombinator.com) </span>
                    </td>
                  </tr>
                </tbody>
              </table>
            </td>
          </tr>
        </tbody>
      </table>
    </center>
  </body>
</html>

```

4. You can see that everything we need, text and url, is located between the `<td class="title">` `</td>` tag:

```

<td class="title">
  <a href="http://ycombinator.com/hdp.html">The Handshake
  Deal Protocol</a>
  <span class="comhead"> (ycombinator.com) </span>
</td>

```

And if you look at the rest of the document, all other postings fall within the same tag.

5. Thus, we have our main XPath: `titles = hxs.select('//td[@class="title"]')`.
6. Now, we just need to establish the XPath for the title and url. Take a look at the HTML again:

```
<a href="http://ycombinator.com/hdp.html">The Handshake  
Deal Protocol</a>
```

7. Both the title and url fall within the `<a>` `` tag. So our XPath must begin with those tags. Then we just need to extract the right attributes, `text` and `@href` respectively.

Need more help testing XPath expressions? Try Scrapy Shell.

Scrapy Shell

Scrapy comes with an interactive tool called Scrapy Shell which easily tests XPath expressions. It's already included with the standard Scrapy installation.

The basic format is `scrapy shell <url>`. Open your terminal and type `scrapy shell http://news.ycombinator.com`. Assuming there are no errors in the URL, you can now test your XPath expressions.

1. Start by using Firebug to get an idea of what to test. Based on the Firebug analysis we conducted a few lines up, we know that `//td[@class="title"]` is part of the XPath used for extracting the title and link. If you didn't know that, you could test it out in Scrapy Shell.
2. Type `hxs.select('//td[@class="title"]').extract()` in the Shell and press enter.
3. It's hard to see, but the URL and title are both part of the results. We're on the right path.
4. Add the `a` to the test:

```
hxs.select('//td[@class="title"]/a').extract()
```

- Now you can see that just the title and URL are part of the results. Now, just extract the text and then the href:

```
hxs.select('//td[@class="title"]/a/text()').extract()
```

and

```
hxs.select('//td[@class="title"]/a/@href').extract()
```

Scrapy Shell is a valuable tool for testing whether your XPath expressions are targeting the data that you want to scrape.

Try some more XPath expressions:

- The "library" link at the bottom of the page:

```
hxs.select('//span[@class="yclinks"]/a[3]/@href').extract()
```

- The comment links and URLs:

```
hxs.select('//td[@class="subtext"]/a/@href').extract()
```

and

```
hxs.select('//td[@class="subtext"]/a/text()').extract()
```

See what else you can extract.

If you need a quick primer on XPath, check out the W3C [tutorial](#). Scrapy also has some great [documentation](#). Also, before you start the next section, read [this](#) part of the Scrapy documentation. Make sure you understand the difference between the `BaseSpider` and `CrawlSpider`.

Wikipedia (BaseSpider)

In this next example, we'll be scraping a listing of new movies from Wikipedia:

http://en.wikipedia.org/wiki/Category:2013_films

First, check the terms of use and the *robots.txt* file and answer the following questions:

- Does scraping or crawling violate their terms of use?
- Are we scraping a portion of the site that is explicitly disallowed?
- Is there an established crawl delay?

All no's, right?

Start by building a scraper to scrape just the first page. Grab the movie title and URL. This is a slightly more advanced example than the previous one. Please try it on your own before looking at the code.

1. Start a new Scrapy project in your Chapter 3 directory:

```
scrapy startproject wikipedia
```

2. Create the *items.py* file:

```
from scrapy.item import Item, Field

class WikipediaItem(Item):
    title = Field()
    url = Field()
```

3. Setup your crawler. You can setup a skeleton crawler using the following command:

```
scrapy genspider -d basic
```

The results are outputted to the screen:

```
class $classname(BaseSpider):
    name = "$name"
    allowed_domains = ["$domain"]
    start_urls = (
        'http://www.$domain/',
    )

    def parse(self, response):
        pass
```

4. Copy and paste the output into gedit, and then finish coding the scraper:

```
# wikibase.py - basespider

from scrapy.spider import BaseSpider
from scrapy.selector import HtmlXPathSelector

from wikipedia.items import WikipediaItem

class MySpider(BaseSpider):
    name = "wiki"
    allowed_domains = ["en.wikipedia.org"]
    start_urls = [
        "http://en.wikipedia.org/wiki/Category:2013_films"
    ]

    def parse(self, response):
        hxs = HtmlXPathSelector(response)
        titles = hxs.select('//tr[@style="vertical-align: top;"]//li')
        items = []
        for title in titles:
            item = WikipediaItem()
            item["title"] = title.select("a/text()").extract()
            item["url"] =
            title.select("a/@href").extract()
            items.append(item)
        return(items)
```

Did you notice the XPath?

```
hxs.select('//tr[@style="vertical-align: top;"]//li')
```

is equivalent to

```
hxs.select('//tr[@style="vertical-align: top;"]/td/ul/li')`
```

Since `` is a child element of `<tr style="vertical-align: top;">`, you can bypass the elements between them by using two forward slashes, `//`.

Navigate to the main project directory:

```
cd wikipedia/wikipedia
```

This time, output the data to a JSON file:

```
scrapy crawl wiki -o wiki.json -t json
```

Take a look at the results. We now need to change the relative URLs to absolute by appending `http://en.wikipedia.org` to the front of the URLs.

First, import the `urlparse` library - `from urlparse import urljoin` - then update the for loop:

```
for title in titles:
    item = WikipediaItem()
    url = title.select("a/@href").extract()
    item["title"] = title.select("a/text()").extract()
    item["url"] = urljoin("http://en.wikipedia.org", url[0])
    items.append(item)
return(items)
```

Delete the JSON file and run the scraper again. You should now have the full URL.

Socrata (CrawlSpider and Item Pipeline)

In this next example, we'll be scraping a listing of publicly available datasets from Socrata: <https://opendata.socrata.com/>.

Create the project:

```
scrapy startproject socrata
```

Start with the BaseSpider. We want the title, URL, and the number of views for each listing. Do this on your own.

1. *items.py*:

```
from scrapy.item import Item, Field

class SocrataItem(Item):
    text = Field()
    url = Field()
    views = Field()
```

2. *socrata_base.py*:

```
# socrata_base.py - basespider

from scrapy.spider import BaseSpider
from scrapy.selector import HtmlXPathSelector

from socrata.items import SocrataItem

class MySpider(BaseSpider):
    name = "socrata"
    allowed_domains = ["opendata.socrata.com"]
    start_urls = [
        "https://opendata.socrata.com"
    ]

    def parse(self, response):
        hxs = HtmlXPathSelector(response)
        titles = hxs.select('//tr[@itemscope="itemscope"]')
        items = []
        for t in titles:
            item = SocrataItem()
            item["text"] = t.select("td[2]/a/
text()").extract()
            item["url"] =
t.select("td[2]/a/@href").extract()
            item["views"] = t.select("td[3]/span/
text()").extract()
            items.append(item)
        return(items)
```

3. Release the spider:

```
scrapy crawl socrata -o socrata.json
```

4. Make sure the JSON looks right.

CrawlSpider

Moving on, let's now look at how to crawl a website as well as scrape. Basically, we'll start at the same starting URL, scrape the page, follow the first link in the pagination links at the bottom of the page. Then we'll start over on that page. Scrape. Crawl. Scrape. Crawl. Scrape. Etc.

Earlier, when you looked up the difference between the `BaseSpider` and `CrawlSpider`, what did you find? Do you feel comfortable setting up the `CrawlSpider`? Give it a try.

1. First, there's no change to `items.py`. We will be scraping the same data on each page.
2. Make a copy of the `BaseSpider`. Save it as `socrata_crawl.py`.
3. Update the imports:

```
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors.sgml import
SgmlLinkExtractor
from scrapy.selector import HtmlXPathSelector

from socrata.items import SocrataItem
```

4. Add the rules:

```
rules = (Rule
(SgmlLinkExtractor(allow=("browse\\?utf8=%E2%9C%93&page=\\d*",
)), callback="parse_items", follow= True),)
```

5. Final code:

```
# socrata_crawl.py - crawlspider

from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors.sgml import
SgmlLinkExtractor
from scrapy.selector import HtmlXPathSelector

from socrata.items import SocrataItem

class MySpider(CrawlSpider):
    name = "socrata2"
    allowed_domains = ["opendata.socrata.com"]
    start_urls = [
        "https://opendata.socrata.com"
    ]

    rules = (Rule
(SgmlLinkExtractor(allow=("browse\\?utf8=%E2%9C%93&page=\\d*",
)), callback="parse_items", follow= True),)

    def parse_items(self, response):
        hxs = HtmlXPathSelector(response)
        titles = hxs.select('//tr[@itemscope="itemscope"]')
        items = []
        for t in titles:
            item = SocrataItem()
            item["text"] = t.select("td[2]/a/
text()").extract()
            item["url"] =
t.select("td[2]/a/@href").extract()
            item["views"] = t.select("td[3]/span/
text()").extract()
            items.append(item)
        return(items)
```

As you can see, the only new part of the code, besides the imports, are the rules, which define the crawling portion of the spider:

```
rules = (Rule
(SgmlLinkExtractor(allow="browse\?utf8=%E2%9C%93&page=\d*",
)), callback="parse_items", follow= True),)
```

Please read over the [documentation](#) regarding rules quickly before you read the explanation. Also, it's important that you have a basic understanding of regular expressions. Please refer to the original Real Python course for a high-level overview.

So, the `SgmlLinkExtractor` is used to specify the links that should be crawled. The `allow` parameter is used to define the regular expressions that the URLs must match in order to be crawled.

Take a look at some of the URLs:

- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=2>
- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=3>
- <https://opendata.socrata.com/browse?utf8=%E2%9C%93&page=4>

What differs between them? The numbers on the end, right? So, we need to replace the number with an equivalent regular expression, which will recognize any number. The regular expression `\d` represents any number, 0 - 9. Then the `*` operator is used as a wildcard. Thus, any number will be followed, which will crawl every page in the pagination list.

We also need to escape the question mark (?) from the URL since question marks have special meaning in regular expressions. In other words, if we don't escape the question mark, it will be treated as a regular expression as well, which we don't want because it is part of the URL. Thus, we are left with this regular expression:

```
browse\?utf8=%E2%9C%93&page=\d*
```

Make sense?

All right. Remember how I said that we need to crawl "ethically"? Well, let's put a 10-second delay between each crawl/scrape combo. This is very easy to forget to do. Once you get yourself banned from a site, though, you'll start to remember.

*I cannot urge you enough to be **careful**. Only crawl sites where it is 100% legal at first. If you start venturing into gray area, do so at your own risk. These are powerful tools you are learning. Act responsibly. Or getting banned from a site will be the least of your worries.*

Speaking of which, did you check the terms of use and the *robots.txt* file? If not, do so now.

To add a delay, open up the *settings.py* file, and then add the following code:

```
DOWNLOAD_DELAY = 10
```

Item Pipeline

Finally, instead of dumping the data a JSON file, let's feed it to a database.

1. Create the database within the "socrata\socrata" directory from your shell:

```
import sqlite3

conn = sqlite3.connect("project.db")
cursor = conn.cursor()
cursor.execute("""CREATE TABLE data
    (text TEXT, url TEXT, views TEXT)
""")
```

2. Update the *pipelines.py* file:

```
import sqlite3

class SocrataPipeline(object):
    def __init__(self):
        self.conn = sqlite3.connect('project.db')
        self.cur = self.conn.cursor()

    def process_item(self, item, spider):
        self.cur.execute("insert into data (text, url,
views) values(?, ?, ?)", (item['text'][0], item['url'][0],
item['views'][0]))
        self.conn.commit()
        return item
```

3. Add the pipeline to the `settings.py` file:

```
ITEM_PIPELINES = ['socrata.pipelines.SocrataPipeline']
```

4. Test this out with the BaseSpider first:

```
scrapy crawl socrata -o project.db -t sqlite
```

Look good? Go ahead and delete the data using the SQLite Browser. Save the database.

Ready? Fire away:

```
scrapy crawl socrata2 -o project.db -t sqlite
```

This will take a while. In the meantime, read about using Firebug with Scrapy from the official Scrapy [documentation](#). Still running? Take a break. Stretch.

Once complete, open the database with the SQLite Browser. You should have about ~20,000 rows of data. Make sure to hold onto this database we'll be using it later.

Homework

- Use your knowledge of BeautifulSoup, which, again, was taught in the original Real Python course, as well as the requests library, to scrape and parse all links from the web2py [homepage](#). Use a for loop to output the results to the screen. Refer back to the main course or the BeautifulSoup [documentation](#) for assistance.

Use the following command to install BeautifulSoup: `pip install beautifulsoup4`

3.9) Web Interaction

Web interaction and scraping go hand in hand. Sometimes, you need to fill out a form to access data or log into a restricted area of a website. In such cases, Python makes it easy to interact in real-time with web pages. Whether you need to fill out a form, download a CSV file weekly, or extract a stock price each day when the stock market opens, Python can handle it. This basic web interaction combined with the data extracting methods we learned in the last lesson can create powerful tools.

Let's look at how to download a particular stock price.

```
# ex3.8a.py Download stock quotes in CSV

import requests
import time

i = 0

# obtain quote once per minute for the next 2 minutes
while (i < 2):

    base_url = 'http://download.finance.yahoo.com/d/quotes.csv'

    # retrieve data from web server
    data = requests.get(base_url,
                         params={'s': 'GOOG', 'f': 's11d1t1c1ohgv', 'e': '.csv'})

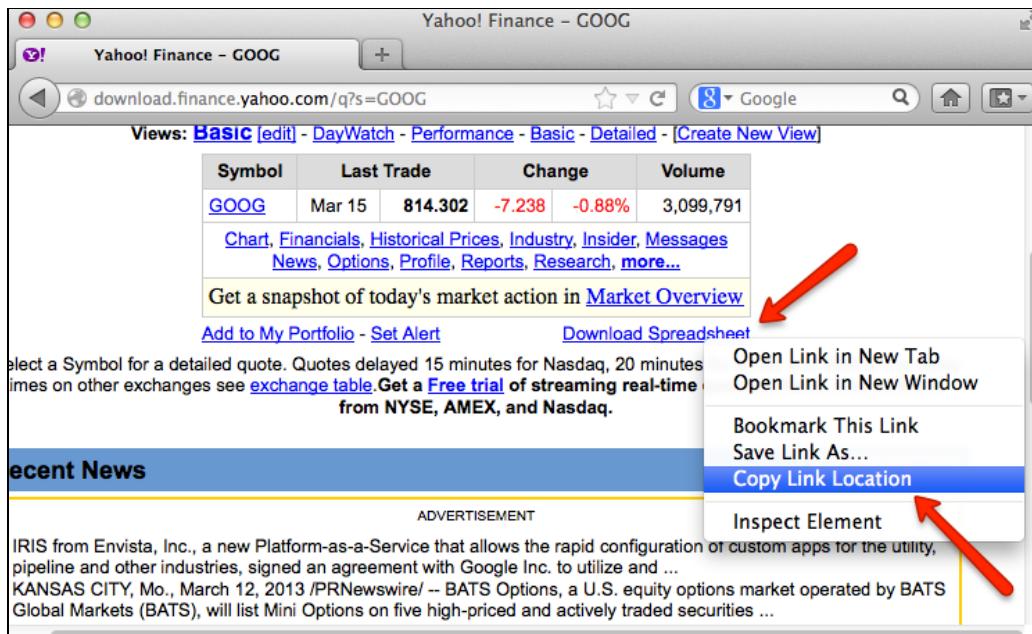
    # write the data to csv
    with open("stocks.csv", "a") as code:
        code.write(data.content)
    i+=1

    # pause for 3 seconds
    time.sleep(3)
```

Save this file as *ex3.8a.py* and run it. Then load up the CSV file after the program ends to see the stock prices. You could change the sleep time to 60 seconds so it pulls the stock price every minute or 360 to pull it every hour.

Let's look at how I got the parameters: `params={'s': 'GOOG', 'f': 's11d1t1c1ohgv', 'e': '.csv'})`

Open <http://download.finance.yahoo.com/> in your browser and search for the stock quote for Google: GOOG. Then copy the url for downloading the spreadsheet:



```
http://download.finance.yahoo.com/d/
quotes.csv?s=goog&f=s11d1t1c1ohgv&e=.csv
```

So to download the CSV, we need to input parameters for `s`, `f`, and `e`, which you can see in the above URL. The parameters for `f` and `e` are constant, which means you could include them in the base_url. So it's just the actual stock quote that changes.

How would you then pull prices for a number of quotes using a loop?

1. <http://docs.python-requests.org/en/latest/>
2. https://developers.google.com/youtube/1.0/developers_guide_python
3. <http://tweepy.github.com/>
4. <http://developer.rottentomatoes.com/docs/read/Home>

5. <http://pages.ebay.com/help/policies/user-agreement.html>
6. <http://scrapy.org/>
7. <http://doc.scrapy.org/en/latest/topics/shell.html>

4) Fundamentals: Server-Side Programming

4.1 Introduction

Python is one of the most widely-used programming languages for building and driving websites. From small scripts used to process HTML forms to entire, enterprise-level websites controlled by web frameworks, Python serves a breadth of roles with regards to web development. Python first made its appearance on the Internet in the form of CGI Scripting. Programs written using CGI are simple yet slow and should thus only be used for small applications - such as form handling. Despite its limitations and the rise of faster methods of getting Python to communicate with web servers, we will be addressing CGI scripts in detail, as they embody the main principles of web frameworks.

To test our CGI scripts, we must use a web server in order to serve the web pages to the web browser. Python comes with a number of built-in web servers, two of which we will be using: SimpleHTTPServer and CGIHTTPServer.¹ These servers essentially turn any directory on your local system into a mini web server. Both are ideal for training purposes as well as debugging code. Perhaps the best part about them, though, is that you can just run them directly from the command line. You don't even need a script!

4.2) SimpleHTTP Server

You'll see in the exercise directory a folder named "simplehttp". Navigate to that directory in your command line. Within it, you should see an *index.html* file.

Now go ahead and run this command: `python -m SimpleHTTPServer 8000`

The `-m` switch tells Python to run a module (`SimpleHTTPServer`, in this case) as a script.

Next, open up a web browser and navigate to the following url:

<http://localhost:8000>. (Local host simply refers to your computer). You should see the content of the `index.html` page:

```
Hello, World!!!! :)
```

That's it. Hit CTRL-C on your keyboard back in the terminal to close the server.

If you received a "socket.error: [Errno 13] Permission denied" message then port 8000 is in use, you can add another port to the end of the command: `python -m SimpleHTTPServer 9000`, for example.

This is a quick way to get a web server up with little configuration and without eating up system resources. I personally use this quick method for testing and sharing files. If you run the command from a directory without an `index.html` file in it, files in that directory will be displayed, and can then be downloaded from another computer within your network. You can access those files by going to `http://your_ip_address:8000`.

To find your IP address:

- Unix: Type `ifconfig en1` in your terminal. The IP address is right after the "inet" descriptor.
- Windows: Type `ipconfig /all` in your terminal. (If you're using Windows 7 or 8, the IP address is called "IPv4".)

You can also write a custom script to do the same thing. The advantage of going this route is that you have the freedom to customize the server. You could set up GET or POST handling for testing purposes:

```
# ex4.2a.py Local web server

import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler

# establish a simple web server
server = BaseHTTPServer.HTTPServer

# provide a request handler
handler = SimpleHTTPRequestHandler

# set HTTP as the protocol
protocol = "HTTP/1.0"

# grab the specified port or use port 8000
if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000

# set the IP address and port
address = ('127.0.0.1', port)
handler.protocol_version = protocol

# create the server
httpd = server(address, handler)
sa = httpd.socket.getsockname()
print "Serving HTTP on", address[0], "port", address[1], "..."

# place the program in an infinite loop
httpd.serve_forever()
```

Run this from the `simplehttp` directory, and you should see the same output as in the last example. You can also establish the port from the command line. Otherwise, it will default to 8000:

```
$ python ex4.2a.py 5000
```

Once again, hit CTRL-C on your keyboard from the terminal to close the server.

4.3) CGI Programming

CGI (Common Gateway Interface) provides a set of standards, which define how information is exchanged between a web server and a custom script. It is one of the simplest methods for generating dynamic web content. Popular in the late 1990s and early 2000s, CGI is still being used today to create very basic, dynamic websites where speed and scalability are of little or no concern.

I've personally used Python CGI Programming to understand some of the basics of creating a dynamic website. Many of the popular high-level frameworks, including web2py and Django, employ various application shortcuts (often called "magic") to speed up the development process. Starting a basic application from scratch is vital for understanding how things work behind the scenes, at a low level, within those frameworks. This will ultimately make you a better developer.

When a user accesses a CGI script from a browser, data is received by the web server (called a request), which, in turn, passes it to the CGI script. This script performs some action and then sends the data back (called a response).

First, let's look at a quick example.

Code:

```
# ex4.3b.py - hello cgi example

#!/usr/bin/env python
print "Content-Type: text/html"
print
print """
<html>
<body>
<h2>Hello, World!</h2>
</body>
</html>
"""
```

Note: When working with CGI, do not include the header at the top of the file - that is, `# ex4.3b.py - hello cgi example`

Save this file as `ex4.3b.py` in the "cgihttp/cgi-bin" directory within the Chapter 4 "exercises" folder. If you are running on a **Unix** machine, you need to make sure both the directory and file are executable before you run the file. To do this, within the terminal, navigate to the "cgihttp" directory, then run the following command to update the user permissions:

```
$ sudo chmod 755 cgi-bin
```

Then, CD into the cgi-bin and run this code:

```
$ sudo chmod 755 ex4.3b.py
```

If you are on an Unix operating system, you will need to set the permissions for each script in this chapter.

This setup is for running locally only, using the test servers provided by Python. If you are working with a web server, please check with your hosting provider to see how to set up your user permissions. You'll want to set up the permissions so that everyone can read and (possibly) execute the file, but only you can write to it.

Navigate to your "cgihttp" directory within the terminal and then fire up the server:

```
$ python -m CGIHTTPServer
```

Navigate in your browser to <http://localhost:8000/cgi-bin/ex4.3b.py>

Output:

```
Hello, World!
```

If you have problems running this file, jump ahead to the end of the lesson where debugging techniques are discussed.

The first two characters of the first line of the script, `#!`, are called a "shebang", which indicates that the file is to be executed by the Python interpreter specified by the path, `/usr/bin/env python`. This is applicable only for Unix-based systems, as Windows handles this process differently. That said, you can run a Python program without it. **Unix users** - Create a new file called "shebang.py" and add the following code. **Windows Users** - Although you can't partake in this particular example, please read it over as it will ultimately pertain to you as well.

```
#!/usr/bin/env python
print "shebang"
```

Run it as usual. Then run the program again using this command: `./shebang.py`.

It should run the same. Now go back and remove the "shebang" and the Python path. Run the program again using both methods. You should have seen a

`./shebang.py: line 2: print: command not found` error using the latter, less verbose method of execution, because the Terminal did not know what to do with the `print` statement. **Since CGI scripts are meant to run on a web-server, you must add the "shebang" line so the server will know to execute the file using the Python interpreter.**

Next, you need to set the HTTP header to `text/html`, which lets the server know that it should expect HTML to be sent. The blank line that follows is required, signaling

the end of the header. You now have the actual HTML. If you view the page source, you will only see the HTML.

Now for a more complicated example: form handling. One of the more common uses for CGI still used today is for processing HTML forms.

HTML form code:

```
<html>
    <title>CGI Form</title>
    <body>
        <form method="POST" action="/cgi-bin/ex4.3c.py">
            <p>Please enter your first name:</p>
            <input type="text" name="name"><input type="submit"
value="Submit">
        </form>
    </body>
</html>
```

Save this file as *ex4.3c.html* in the `cgihttp` directory within the `exercises` folder:

```
# ex4.3c.py - cgi form handling

#!/usr/bin/env python
import cgi
form = cgi.FieldStorage()

# retrieve the name entered in the form; set to empty string if
# none
name = cgi.escape(form['name'].value) if 'name' in form else ''
print "Content-Type: text/html\n"
htmlFormat = """
<html>
  <body>
    <p>Hi, {name}</p>
  </body>
</html>
"""
print htmlFormat.format(name=name)
```

Back on the terminal, navigate to the `cgihttp` directory, then run the following command:

```
$ python -m CGIHTTPServer
```

Open another web browser and navigate to <http://localhost:8000/ex4.3c.html>. You should see the form there. Go ahead and type your name, hit submit, and if all goes well, it should be outputted back:

```
Hi, *<your name>*
```

If you go back to the terminal you can see the messages from the server:

```
Serving HTTP on 0.0.0.0 port 8000 ...
1.0.0.127.in-addr.arpa - - [29/Jan/2013 16:02:51] "GET
/ex4.3c.html HTTP/1.1" 200 -
1.0.0.127.in-addr.arpa - - [29/Jan/2013 16:02:57] "POST
/cgi-bin/ex4.3c.py HTTP/1.1" 200 -
```

Notice the HTTP requests, GET and POST, and the response message 200. We sent the server a GET request when we navigated to *ex4.3c.html*. When we submitted the form, a POST request was sent to the server. Both requests worked, as we received a response 200 for each. When our programs get more complicated, looking at the server responses is a good way to debug our programs and narrow down the problems.

The `FieldStorage()` class is used to fetch inputted values. In the example above, we used the key `name` to output its associated value.

Now, let's take this a step further and process some data that a user inputs via an HTML form:

```
<html>
    <title>Square Root</title>
    <body>
        <form method="POST" action="/cgi-bin/ex4.3d.py">
            <p>Hello. Please enter a number so we can provide the
            square root. Thanks!</p>
            <br/>
            <input type="text" name="sqrt"><input type="submit"
            value="Submit">
        </form>
    </body>
</html>
```

Save this as *ex4.3d.html* in your cgihttp directory:

```
# ex4.3d.py - cgi square root

#!/usr/bin/env python
import cgi, math
form = cgi.FieldStorage()

# retrieve the number entered in the form; set to 0 if none
value = cgi.escape(form['sqrt'].value) if 'sqrt' in form else 0
sqrt = str(math.sqrt(float(value)))

print "Content-Type: text/html\n"
htmlFormat = """
<html>
  <body>
    <p>Hi. The square root of {value} equals {sqrt}</p>
  </body>
</html>
"""
print(htmlFormat.format(value=value, sqrt=sqrt))
```

Within your `cgihttp` directory, run the command `python -m CGIHTTPServer`. Then navigate to <http://localhost:8000/ex4.3d.html> in your browser and you should see the form. Test the form out.

This works very similarly to the last example. In this example, however, we took the additional step of processing the data by using the `math` module to calculate the square root of the value.

Let's look at a simple login example:

```
<html>
    <title>Login</title>
    <body>
        <form method="POST" action="/cgi-bin/ex4.3e.py">
            <p>Please enter your username and password to access the
            restricted site</p>
            <p>Username: <input type="text" name="user" value=""></p>
            <p>Password: <input type="password" name="password"
            value=""></p>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

Save this as *ex4.3e.html* in your cgihttp directory:

```
# ex4.3e.py - login

#!/usr/bin/env python

import cgi

form = cgi.FieldStorage()

# retrieve the user and password entered in the form; set to
empty string if none
user = cgi.escape(form['user'].value) if 'user' in form else ''
passwd = cgi.escape(form['password'].value) if 'password' in
form else ''

username = "admin"
password = "admin"
if user == username and passwd == password:
    msg = "Welcome {username}. You are now logged
in.".format(username=username)
else:
    msg = "Incorrect username or password. Please try again."

print "Content-Type: text/html\n"
htmlFormat = """
<html>
    <body>
        <p>{msg}</p>
    </body>
</html>
"""
print htmlFormat.format(msg=msg)
```

Try both the correct username and password, "admin" and "admin", as well as an invalid username and/or password.

Can you figure out what these statements do:

```
user = form['user'].value  
passwd = form['password'].value
```

In the final example, let's look at how to access a database.

```
# ex4.3f.py - cgi database access

#!/usr/bin/env python
import sqlite3
import cgi

print "Content-Type: text/xml"
print

with sqlite3.connect("names.db") as connection:
    c = connection.cursor()

    c.executescript("drop table if exists names")
    c.execute("create table names(first TEXT, last TEXT)")

    # insert multiple records using a tuple
    names = [
        ('John', 'Bell'),
        ('Michael', 'Sloane'),
        ('Rachel', 'Peterson')
    ]

    # insert data into table
    c.executemany('INSERT INTO names VALUES(?, ?)', names)

    c.execute("select first, last from names")

    # retrieve data
    rows = c.fetchall()

    # build xml string
    print "<names>"
    for r in rows:
        print "\t<name>"
        print "\t\t<first>%s</first>" % (r[0])
        print "\t\t<last>%s</last>" % (r[1])
        print "\t</name>"
    print "</names>"
```

Fire up the server. Open another web browser and navigate to <http://localhost:8000/cgi-bin/ex4.3f.py>. You should see the following output:

```
<names>
  <name>
    <first>John</first>
    <last>Bell</last>
  </name>
  <name>
    <first>Michael</first>
    <last>Sloane</last>
  </name>
  <name>
    <first>Rachel</first>
    <last>Peterson</last>
  </name>
</names>
```

In this example, we indicated to the server that it should expect an XML file. You should recognize the SQL syntax from the Databases chapter. If not, go back and refresh your memory, as we will be covering a lot more SQL in the following chapters.

Debugging CGI

You will often need to debug your script in order to correct errors. This can be a difficult task, depending on the issues.

Possible issues:

1. Incorrect Python code
2. CGI script in the wrong directory
3. Incorrect script and/or directory permissions
4. Incorrect "shebang" line

Start by checking the Python code by running the file on its own (e.g., `python ex4.3b.py`). There could be a syntax error, for example. Next, enable the `cgitb`

module, which is part of Python's standard library, to display errors. Let's look at a quick example.

Code:

```
# ex4.3g.py - cgitb module

#!/usr/bin/env python
import cgitb
cgitb.enable()
print "Content-Type: text/html"
print
print 1/0
```

Load the server, and then navigate to <http://localhost:8000/cgi-bin/ex4.3g.py> and you should see the following output:

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

You can see that there is a division error. Comment out `cgitb.enable()`, run again, and see the difference. You can add this module to any of your scripts to help with debugging.

You never want your end users to see such errors, so make sure to disable cgitb before deploying to a web server accessible by end users

Unix users: To find the correct path for the "shebang" line, type the following command in your terminal: `which python`

4.4) Old School Web Application

At this point, we have gone through much of the basic web fundamentals, both on the client and server-side. We're now going to put much of what you've learned to use by creating a dynamic blog, backed by a SQLite database.

This exercise is meant to be difficult. Working with Python via CGI is very rare in the real world. However, this teaches you the lowest level of web development so that you will understand the basics before adding in web frameworks. Try your best on this, and if you get stuck and can't finish, you will be fine as long as you understand the overall concepts.

Application workflow: When a user logs in, they'll be presented with all of the blog posts. The user can then add new text-only blog entries from the same screen, read the entries themselves, or logout. That's it. Before we even think about coding, though, we need to define the application's workflow from a user's perspective. You can setup a matrix that defines all the actions, or requests, a user can make as well as the subsequent server response in terms of CRUD:

Login Page

| User Action | HTTP Request | Server Response | Response Code |
|--|--------------|------------------------|---------------|
| Navigates to login page | GET | HTML is rendered | 200 |
| Attempts to log in using correct credentials | POST | Redirects to Main Page | 302 |
| Attempts to log in using incorrect credentials | POST | Login is denied | 401 |

Main Page

| User Action | HTTP Request | Server Response | Response Code |
|--|--------------|---|---------------|
| After login, user is redirected to Main page | GET | HTML (including blog posts) is rendered | 200 |
| Posts a blog article | POST | Article accepted, added to database | 200 |
| Logs out | POST | Redirects to login page | 302 |

There are a number of additional Requests and Responses besides those listed. For now, let's focus on the actual one's that affect the user directly.

Before looking at my code, see how much of this you can plan and implement on your own. Like before, start by mapping the processes out on a piece of paper to determine the number of scripts and HTML files you'll need. From there, write out short statements of what each script will do, and then turn those statements into the comments for your program. You will then be ready to start writing code.

All right. Start by creating a directory called "blog" within your Chapter 4 folder. Then, in that directory, create a sub-directory called "cgi-bin". **Unix users:** Don't forget to set the proper executables on the "cgi-bin" directory.

Database

Since databases are the core of most applications, it's best to start there. It can also be difficult to make changes to a database management system (i.e., SQLite, MySQL, SQL Server) once configured, so database design is important. Even though we will be using SQLite, which is easy to configure and modify (as you now know), always take database design seriously.

Our database will have one table called *posts*, with two fields - *title* and *post*.

Let's create the database and populate the table with some dummy data:

```
# ex4.4_blog1.py - Create a SQLite3 table and populate it with
# data

import sqlite3

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # create the table
    c.execute("""CREATE TABLE posts(title TEXT, post TEXT)""")

    # insert multiple records using a tuple
    posts = [
        ("Good", "I'm good."),
        ("Well", "I'm well."),
        ("Excellent", "I'm excellent."),
        ("Okay", "I'm okay.")
    ]

    # insert data into table
    c.executemany('INSERT INTO posts VALUES(?, ?)', posts)
```

Save the file in the "blog" directory and run it from the terminal. Double-check to make sure the data was entered correctly with the SQLite Database Browser.

Main Page - outputting posts

Now that the database has been implemented, let's create the main listing page.

We'll be adding user authentication and security after the main functions have been put into place. It's a good strategy to break development into pieces like this, adding features step-by-step, to simplify the process.

Before adding the form, we'll start with simply outputting the data to the screen to make sure everything prints correctly.

Code:

```
# ex4.4_blog2.py - Query reviews tables, aggregate total of
# posts

# import the sqlite3 library
import sqlite3

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # total posts
    c.execute("SELECT COUNT(post) FROM posts")
    total = c.fetchone()[0]
    print "Total Posts: ", total

    # query posts
    c.execute("SELECT * FROM posts")

    # fetchall() retrieves all records from the query
    posts = c.fetchall()

    print "\nBlog Posts\n======""

    # output the rows to the screen, row by row
    for p in posts:
        print "Title: ", p[0]
        print "Post: ", p[1], "\n"
```

Once again, save to the "blog" directory and then run it from the terminal. You should see the data we just added to the database.

Next, let's look at how to create the HTML output via CGI:

```
# ex4.4_blog3.py - Query reviews tables, aggregate total of
posts

# import the libraries
import sqlite3
import cgi
import os

# set http header
print "Content-Type: text/html"
print

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # total posts
    c.execute("SELECT COUNT(post) FROM posts")
    total = c.fetchone()[0]

    # query posts
    c.execute("SELECT * FROM posts")

    # fetchall() retrieves all records from the query
    posts = c.fetchall()

    data_table = "<table
border=1><tr><th>Title</th><th>Post</th></tr>"
    for p in posts:
        data_table += "<tr>"
        data_table += "<td>%s</td>" % (p[0])
        data_table += "<td>%s</td>" % (p[1])
        data_table += "</tr>"
    data_table += "</table>"
    print """
<html>
```

```
<head><title>Blog</title></head>
<body>
    <h1><center>Bloggy</center></h1>
    <h2>Welcome!</h2>
    <p>Total Posts: %s</p>
    <p>%s</p>
</body>
</html>
""" % (total, data_table)
```

Save this file in your "cgi-bin", navigate to your blog directory in your terminal, and then run the command `python -m CGIHTTPServer`. Navigate to http://localhost:8000/cgi-bin/ex4.4_blog3.py in your browser and you should see the following output:

| Title | Post |
|-----------|----------------|
| Good | I'm good. |
| Well | I'm well. |
| Excellent | I'm excellent. |
| Okay | I'm okay. |

Main Page - adding posts

Next let's add a basic form to allow users the ability to create new posts.

Essentially, we just need to add a basic html form to the previous script:

```
# ex4.4_blog4.py - Main Page

# import the libraries
import sqlite3
import cgi
import os

# set http header
print "Content-Type: text/html"
print

# get post data
form = cgi.FieldStorage()
title = form.getFirst('title', '')
post = form.getFirst('post', '')

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # insert data to db if not empty
    if title != "" and post != "":
        # insert data into table
        c.execute('INSERT INTO posts VALUES(?, ?)', (title,
post))

    # total posts
    c.execute("SELECT COUNT(post) FROM posts")
    total = c.fetchone()[0]

    # query posts
    c.execute("SELECT * FROM posts")

    # fetchall() retrieves all records from the query
    posts = c.fetchall()
```

```
data_table = "<table border=1><tr><th>Title</th><th>Post</th></tr>"  
for p in posts:  
    data_table += "<tr>"  
    data_table += "<td>%s</td>" % (p[0])  
    data_table += "<td>%s</td>" % (p[1])  
    data_table += "</tr>"  
data_table += "</table>"  
print """\n<html>\n    <head><title>Blog</title></head>\n    <body>\n        <h1><center>Bloggy</center></h1>\n        <h2>Welcome!</h2>\n        <p>Total Posts: %s</p>\n        <p>%s</p>\n        <div>\n            <form method="POST" action="ex4.4_blog4.py">\n                <h2>Add New Post:</h2>\n                <p>Title: <input type="text" name="title" value=""></p>\n                <p>Post: <input type="text" name="post" value=""></p>\n                <input type="submit" value="Submit">\n            </form>\n        </div>\n    </body>\n</html>""" % (total, data_table)
```

Save to the "cgi-bin", load the server from the "blog" directory, navigate to http://localhost:8000/cgi-bin/ex4.4_blog3.py. Test out the form. Are you able to add a new post?

Login Page

We will now be adding a login page so that only authenticated users will be able to view the Main Page.

```
<html>
    <title>Login Required</title>
    <body>
        <form method="POST" action="/cgi-bin/ex4.4_login.py">
            <h2>Login Required!</h2>
            <p>Please enter your username and password to access
            the restricted site</p>
            <p>Username: <input type="text" name="user"
            value=""></p>
            <p>Password: <input type="password" name="password"
            value=""></p>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

Save this as *ex4.4_login.html* under your "blog" folder. Do not test it yet. Also, create the following script under "cgihttp/cgi-bin" and save it as

```
# ex4.4_login.py - login

#!/usr/bin/env python
import cgi

# get post data
form = cgi.FieldStorage()
post_user = form.getFirst('user', '')
post_password = form.getFirst('password', '')

# define username and password
username = "admin"
password = "admin"

if post_user == username and post_password == password:
    # if authentication is ok
    print "Refresh: 0; url=ex4.4_blog5.py\r\n"
else:
    # if authentication failed
    print "Refresh: 0; url=../ex4.4_login.html\r\n"
```

Essentially, this script checks if the username and password entered by the user in *login.html* are correct. If correct, the page is redirected to the Main Page. Otherwise, it is redirected back to *login.html*.

The `print "Refresh: 0; url=ex4.4_blog4.py\r\n"` statement is used for the redirection. Notice the relative path in the second print statement. Are you familiar with relative vs. absolute paths?

Cookies

Although we have created a login page, the user can still access our Main Page by navigating his or her browser to http://localhost:8000/cgi-bin/ex4.4_blog4.py. To address this issue, our main script must be able to check if the user has been authenticated or not. To allow the script to do this, we will be using cookies.

Cookies are pieces of information about the user that are stored in the user's browser.

Verifying user authentication through cookies alone is not secure. There are better solutions - namely using sessions. This example nevertheless will show you how to store cookies to the user's browser. You will learn about sessions in the next chapter.

The final login script should be:

```
# ex4.4_login.py - login

#!/usr/bin/env python
import cgi
import Cookie

# get post data
form = cgi.FieldStorage()
post_user = form.getFirst('user', '')
post_password = form.getFirst('password', '')

# define username and password
username = "admin"
password = "admin"

if post_user == username and post_password == password:
    # if authentication is ok
    thiscookie = Cookie.SimpleCookie()
    thiscookie['logged_in'] = True
    print thiscookie
    print "Refresh: 0; url=ex4.4_blog5.py\r\n"
else:
    # if authentication failed
    print "Refresh: 0; url=../ex4.4_login.html\r\n"
```

Save this in your "cgi-bin" directory, but do not run it just yet. If the username and password are correct, the script will create a cookie with value = True.

Now we also have to update the Main Page:

```
# ex4.4_blog5.py - Main Page

# import the libraries
import sqlite3
import cgi
import os
import Cookie

logged_in = False
# check if the user is authenticated through a Cookie named
'logged_in'
thiscookie = Cookie.SimpleCookie()
if os.environ.has_key('HTTP_COOKIE'):
    thiscookie.load(os.environ['HTTP_COOKIE'])
    if 'logged_in' in thiscookie:
        logged_in = bool(thiscookie['logged_in'].value)

# redirect to login page if not authenticated
if not logged_in:
    print "Refresh: 0; url=../ex4.4_login.html\r\n"

# set http header
print "Content-Type: text/html"
print

# get post data
form = cgi.FieldStorage()
title = form.getFirst('title', '')
post = form.getFirst('post', '')

# create a new database if the database doesn't already exist
with sqlite3.connect("blog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # insert data to db if not empty
    if title != "" and post != "":
        pass
```

```
# insert data into table
c.execute('INSERT INTO posts VALUES(?, ?)', (title,
post))

# total posts
c.execute("SELECT COUNT(post) FROM posts")
total = c.fetchone()[0]

# query posts
c.execute("SELECT * FROM posts")

# fetchall() retrieves all records from the query
posts = c.fetchall()

data_table = "<table
border=1><tr><th>Title</th><th>Post</th></tr>"
for p in posts:
    data_table += "<tr>"
    data_table += "<td>%s</td>" % (p[0])
    data_table += "<td>%s</td>" % (p[1])
    data_table += "</tr>"
data_table += "</table>"
print """
<html>
<head><title>Blog</title></head>
<body>
    <h1><center>Bloggy</center></h1>
    <h2>Welcome!</h2>
    <p>Total Posts: %s</p>
    <p>%s</p>
    <div>
        <form method="POST" action="ex4.4_blog5.py">
            <h2>Add New Post:</h2>
            <p>Title: <input type="text" name="title"
value=""></p>
            <p>Post: <input type="text" name="post"
value=""></p>
            <input type="submit" value="Submit">
        </form>
    </div>
</body>
</html>
```

```
<form method="POST" action="ex4.4_logout.py">
    <input type="submit" value="Logout">
</form>
</div>
</body>
</html>
""" % (total, data_table)
```

Once again, save this to the "cgi-bin" but don't run it yet.

Finally, we need a logout script. This script will simply unset the cookie named 'logged_in' and redirect the page to the login page:

```
# ex4.4_logout.py - login

#!/usr/bin/env python
import Cookie

# unset the Cookie
thiscookie = Cookie.SimpleCookie()
thiscookie['logged_in'] = ''
print thiscookie

# redirect to login page
print "Refresh: 0; url=../ex4.4_login.html\r\n"
```

Save to the "cgi-bin".s

Unix users: *Don't forget to make all files executable that reside in the cgi-bin, except for the database.*

Fire up the server again (from the "blog" directory), then navigate to http://localhost:8000/cgi-bin/ex4.4_logout.py. Test everything:

1. Can you login?
2. Are all the posts displaying?
3. Can you add a new post?

4. What happens when you logout?
 5. What happens when you use the wrong login credentials?
-

1. <http://docs.python.org/2/library/internet.html> ↪

Interlude: Modern Web Development

Overview

So far, we have addressed older technologies (e.g. Python, HTML, HTTP, SQL), used for web development since the advent of the Web. These technologies, however, underlie the web frameworks currently used today. Learning how each of these work is key to understanding what makes web frameworks tick, underneath the hood.

By learning web development from the ground up, you will develop a deeper understanding of web frameworks, allowing for quicker and more flexible web development.

In regards to web development, you've so far essentially learned how to walk, using these older technologies. Now, we are ready to quicken our pace. Before we start running though, let's have an overview of modern web development.

Front-end, Back-end, and Middleware

Modern web applications are made up of three parts or layers, each having a specific set of technologies.

1. **Front-end:** The presentation layer, it's what the end user sees when interacting with a web application. HTML makes up the structure, CSS provides a pretty facade, and JavaScript/jQuery create interaction. This layer is housed by the web browser. The Front-end is also reliant on the application logic and data source provided by the Middleware and Back-end in order to function.
2. **Middleware:** This layer relays information between the Front and Back-ends, in order to:

- Process HTTP requests and responses; Connect to the server;
 - Interact with APIs; and
 - Manage URL routing, authentication, sessions, and cookies.
3. **Back-end:** This is where data is stored, analyzed, and processed. Languages such as Python, PHP, and Ruby communicate back and forth with the database, web service, or other data source to produce the end-user's requested data.

Developers adept in web architecture and in programming languages like Python, have traditionally worked on the Back-end and Middleware layers, while designers have focused on the Front-end. These roles are becoming less and less defined, however - especially in start-ups. Developers are now also handling much of the Front-end work. This is due to both a lack of quality designers and the emergence of Front-end frameworks, like Bootstrap and backbone.js, which have significantly sped up the design process.

Model-View-Controller (MVC)

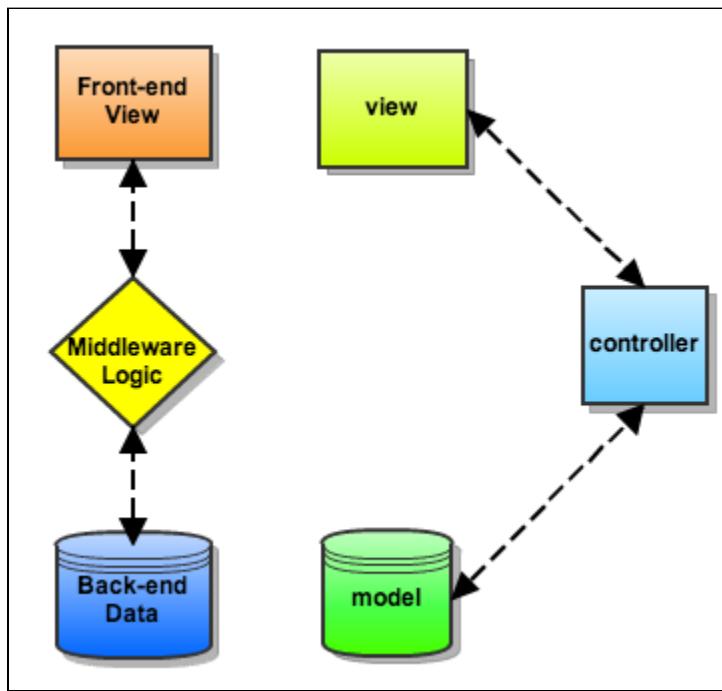
Web frameworks reside just above those three layers, abstracting away much of the processes that occur in each. There are pros and cons associated with this: It's great for experienced web developers, for example, who understand the automation (magic!) behind the scenes.

For example, think if you had to define the aspects that make up a tree, when referring to a tree, rather than just saying the word tree. You'd have to say, "The trunk with branches and leaves ... is getting big", instead of, "The tree is getting big." In other words, web frameworks simplify web development, by handling much of the superfluous, repetitious tasks.

For a beginner, however, it can be confusing, which reinforces the need for the Fundamentals Section of this course.

Frameworks also separate the presentation from the application logic and the underlying data in what's commonly referred to as the Model-View-Controller

architecture pattern. While the Front-end, Back-end, and Middleware layers operate linearly, MVC operates in a triangular pattern:



We'll be covering this pattern numerous times throughout the remainder of this course. Let's get to it!

Homework

- Please read more about the MVC pattern [here](#).
- Read about the differences between a website and a web application [here](#).
(Yes, there is a difference.)

5) Flask: QuickStart

5.1) Overview

Building on what you learned in the first section, this section introduces you to the Flask web framework.



Flask grew from an elaborate April fool's joke in 2010 into one of the most popular Python web frameworks in use today. [1](#) Small yet powerful, you can build your application from a single file, and, as it grows, organically develop components to add functionality and complexity [2](#). Let's take a look.

5.2) Installation

1. Within your terminal, navigate to your "realpython" directory, and then run the following commands to create a virtualenv.

```
virtualenv flask --no-site-packages
```

2. Navigate into the "flask" directory (root directory), then activate the virtual env:

Unix:

```
source bin/activate
```

Windows:

```
scripts\activate
```

3. Now install Flask:

```
pip install flask
```

From this point on, you will no longer be separating virtualenvs by chapter. Instead, each project will have a separate virtualenv.

5.3) Hello World

Let's start with a quick Hello World example. Create a new directory inside of the "flask" directory called "hello" and navigate into it.

1. Open gedit and add the following code:

```
# ex5.3a.py - Flask Hello World

from flask import Flask

# create the application object
app = Flask(__name__)

# use decorators to link the function to a url
@app.route("/")
@app.route("/hello")

# define the view using a function, which returns a string
def hello_world():
    return "Hello, World!"

# start the development server using the run() method
if __name__ == "__main__":
    app.run()
```

2. Save the file and run it:



A screenshot of a terminal window titled "hello — python — 80x24". The window shows the command `(flask)Michaels-MacBook-Pro:hello michaelherman\$ python ex5.3a.py` being run, followed by the output `* Running on http://127.0.0.1:5000/`.

1. This creates a test server (or development server), listening on port 5000. Open a web browser and navigate to <http://127.0.0.1:5000/>. You should see the "Hello, World!" greeting.
2. Test out the URL <http://127.0.0.1:5000/hello> as well.

Back in the terminal, press CTRL-C to stop the server.

What's going on here?

First, let's look at the code without the functions:

```
from flask import Flask

app = Flask(__name__)

if __name__ == "__main__":
    app.run()
```

1. Here, we imported the Flask class object.
2. Next, we created an instance of the application object from the Flask class. In other words, we defined a new Flask application and established the name of the module: "app".
3. Finally, we used the run() method to run the app locally. By setting the `__name__` variable equal to `"__main__"`, we indicated that we're running the statements in the current file rather than importing it.

I know that is confusing and probably makes little, if any, sense at this point, but you will eventually get it. Trust me.

Let's look at another example.

- Create two new files:

first.py

```
# first.py

if __name__ == '__main__':
    print 'This print statement is ran from the current
module.'
else:
    print 'This print statement is ran from a different
module'
```

second.py

```
#second.py
```

```
import first
```

- Now run the files:

```
$ python first.py
This print statement is ran from the current module.
```

```
$ python second.py
This print statement is ran from a different module
```

- *first.py* runs the code within the current module, while *second.py* imports the code from *first.py*.

Does that make better sense? If not, take the "Google-it-first" approach. Again, this is a confusing topic that many developers don't even really understand *how* it works. That said, modules are powerful tools. Learn the basics and use them, regardless of whether you *truly* understand how they work.

Now, let's look at the functions

```
@app.route("/")
@app.route("/hello")
def hello_world():
    return "Hello, World!"
```

1. The function should make sense. We're simply returning the string "`Hello, World!`".
2. Those funny looking statements above the function are called decorators. According to the official Python documentation, a decorator is defined as:

A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. ³

In other words, decorators are called before the function.

Let's look at a simple example:

```
def world(fn):
    def wrapped():
        return fn() + ", World!"
    return wrapped

def hello():
    return "Hello"

@world
def hello_world():
    return "Hello"

print hello()
print hello_world()
```

Run this from your terminal:

```
>>> def world(fn):
...     def wrapped():
...         return fn() + ", World!"
...     return wrapped
...
>>> def hello():
...     return "Hello"
...
>>> @world
... def hello_world():
...     return "Hello"
...
>>> print hello()
Hello
>>>
>>> print hello_world()
Hello, World!
>>> □
```

Make sense?

All right. Going back to the Flask example -

```
@app.route("/")
@app.route("/hello")
def hello_world():
    return "Hello, World!"
```

... the decorators define the URL routes. We assigned the `/` and `/hello` URLs to the `hello_world()` function. Thus, when you pull up either URL in your browser, the `hello_world()` function is called. Try adding another route such as `@app.route("/hw")` to your script. Fire up the server and navigate to the new URL: <http://127.0.0.1:5000/hw>.

See how easy it is to define URLs?

Please go over this lesson again so it sinks in. It's imperative that you understand everything that's going on, as all the Flask apps use this basic structure.

Homework

- Read about [modules](#) from the Python official documentation.
- Forgive me for not being able to better explain them myself. :)

5.4) Blog App

Let's quickly recreate the blog app we created in Chapter 5.

Requirements:

After a user logs in he or she is presented with all of the blog posts. Users can add new text-only blog entries from the same screen, read the entries themselves, or logout. That's it.

Project Structure

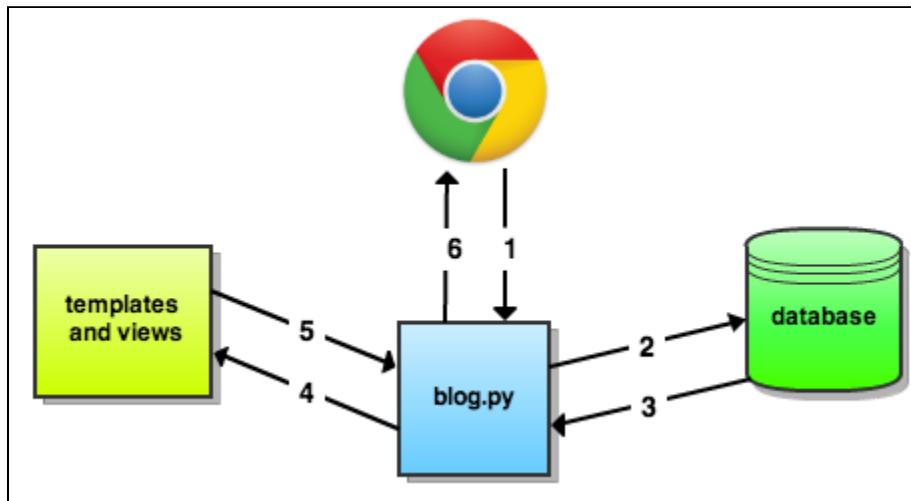
1. Back in your Flask root directory, fire up your virtualenv:

- Unix - `source bin/activate`

- Windows - `scripts\activate`
2. Create a new directory within the "Flask" root called "blog"
 3. Set up the following files and directories:

```
/blog  
/static  
/templates  
blog.py
```

This is a good project structure for almost any small application. The main HTML files and templates are held in the "templates" directory, while the styles (CSS, fonts, images, JavaScripts, etc.) are found in the "static" directory. The `blog.py` file, which is the application module (or controller), receives HTTP requests from the browser and then communicates with the database and templates to generate responses back to the user:



Model

As you recall, our database has one table called `posts` with two fields - `title` and `post`. We can use the exact same script to create and populate the database:

```
# fblog1.py - Create a SQLite3 table and populate it with data

import sqlite3

# create a new database if the database doesn't already exist
with sqlite3.connect("fblog.db") as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # create the table
    c.execute("""CREATE TABLE posts
                (title TEXT, post TEXT)
                """)

    # insert dummy data into the table
    c.execute('INSERT INTO posts VALUES("Good", "I\'m good.")')
    c.execute('INSERT INTO posts VALUES("Well", "I\'m well.")')
    c.execute('INSERT INTO posts VALUES("Excellent", "I\'m
excellent.")')
    c.execute('INSERT INTO posts VALUES("Okay", "I\'m okay.")')
```

Save the file and run it. Then, check the SQLite Browser to ensure the table was created correctly.

Application Module/Controller

Like the application module in the `hello_world` app, this script will define the imports, configurations, and each view.

```
# blog.py - application module

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for, g
import sqlite3

# configuration
DATABASE = 'fblog.db'

app = Flask(__name__)

# pulls in configurations by looking for UPPERCASE variables
app.config.from_object(__name__)

# function used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

if __name__ == '__main__':
    app.run(debug=True)
```

Save this file as *blog.py* in your main project directory. The configuration section is used for defining application-specific settings.⁴

Views

After a user logs in, he or she is redirected to the main blog homepage where all posts are displayed. Users can also add posts from this page. For now, let's get the page set up, and worry about the functionality later.

```
{% extends "template.html" %}  
{% block content %}  
<div class="jumbo">  
    <h2>Welcome to the Flask Blog!</h2>  
</div>  
{% endblock %}
```

We also need a login page:

```
{% extends "template.html" %}  
{% block content %}  
<div class="jumbo">  
    <h2>Welcome to the Flask Blog!</h2>  
    <h3>Please login to access your blog.</h3>  
    <p>Temp Login: <a href="/main">Login</a></p>  
</div>  
{% endblock %}
```

Save these files as *main.html* and *login.html* respectively in the "templates" directory. I know you have questions about the strange code in both these files. We'll get to that in just a second.

Now update *blog.py* by adding two new functions for the views:

```
@app.route('/')  
def login():  
    return render_template('login.html')  
  
@app.route('/main')  
def main():  
    return render_template('main.html')
```

Updated code:

```
# blog.py - application module

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for, g
import sqlite3

# configuration
DATABASE = 'fblog.db'

app = Flask(__name__)

# pulls in configurations by looking for UPPERCASE variables
app.config.from_object(__name__)

# method used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

@app.route('/')
def login():
    return render_template('login.html')

@app.route('/main')
def main():
    return render_template('main.html')

if __name__ == '__main__':
    app.run(debug=True)
```

We mapped the URL `'/'` to the `login()` function, which in turn sets the route to `login.html` in the templates directory. Now, how about the main page? Try to explain it to yourself.

Templates

Templates (or boilerplates) are HTML skeletons that serve as the base for either your entire website or pieces of your website. They eliminate the need to code the basic HTML structure more than once. Separating templates from the main business logic (*blog.py*) helps with the overall organization. As your app grows, for example, you could have a designer working on the front-end templates while you work on the backend application module.

Remember how ugly it was to use print statements to embed Python into HTML from Chapter 4:

```
#!/usr/bin/env python
print "Content-Type: text/html"
print
print """
<html>
<body>
<h2>Hello, World!</h2>
</body>
</html>
"""
```

Templates make it much easier to combine HTML and Python in a programmatic-like manner.

Let's start with a basic template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome, friends!</title>
  </head>
  <body>
    <div class="container">
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

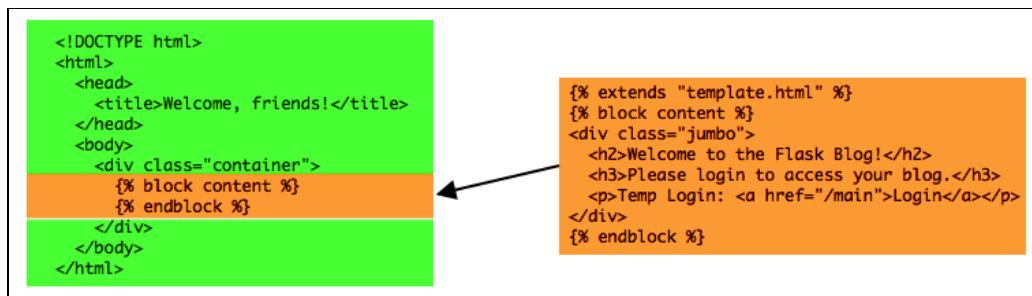
Save this as *template.html* within your templates directory.

There's a relationship between the basic template, *template.html*, and views, *login.html* and *main.html*. This is called template inheritance.

Look back at one of the views. Do you see the code `{% extends "template.html" %}`? This tag establishes the first relationship between the template and views. When Flask renders *main.html* it must first render *template.html*.

Anything surrounded by `{? ?}` is Python-like code, while objects surrounded by `{{ }}` are variables.⁵

You may have also noticed that both the views and template files have identical block tags: `{% block content %}` and `{% endblock %}`. These define where the child templates, *login.html* and *main.html*, fill in on the parent template. When Flask renders the parent template, *template.html*, the block tags are filled in with the code from the child templates:



Run the server!

All right! Fire up your server, navigate to <http://localhost:5000/>, and let's run a test to make sure everything is working up to this point.

You should see the login page, and then if you click the link, you should be directed to the main page. If not, kill the server and double-check your code against mine.

User Login

Now that we have the basic structure set up, let's have some fun and add the blog's main functionality. Starting with the login page, we need to set up a basic HTML form for users to login, so that they can access the main blog page.

Add the following username and password variables to the configuration section in `blog.py`:

```
USERNAME = 'admin'
PASSWORD = 'admin'
```

Also in the configuration section, add the `secret_key`, which is used for managing user sessions:

```
SECRET_KEY = 'hard to guess'
```

Make the value of your secret key really, really hard, if not impossible, to guess. Use a random key generator to do this. Never, ever use a value you pick on your own. [6](#)

Updated `blog.py` configuration:

```
# configuration
DATABASE = 'fblog.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'hard to guess'
```

Update the `login()` function in the `blog.py` file to match the following code:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']
or request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid Credentials. Please try again.'
        else:
            session['logged_in'] = True
            return redirect(url_for('main'))
    return render_template('login.html', error=error)
```

This function compares the username and password entered against those from the configuration section. If the correct username and password are entered, the user is redirected to the main page and the session key, `logged_in`, is set to `True`. If the wrong information is entered, an error message is flashed to the user.

Now, we need to update `login.html` to include the HTML form:

```
{% extends "template.html" %}  
{% block content %}  
    <h2>Welcome to the Flask Blog!</h2>  
    <h3>Please login to access your blog.</h3>  
    <form action="" method="post">  
        Username: <input type="text" name="username" value="{{  
            request.form.username }}">  
        Password: <input type="password" name="password" value="{{  
            request.form.password }}">  
        <p><input type="submit" value="Login"></p>  
    </form>  
{% endblock %}
```

Next, add a function for logging out to *blog.py*:

```
@app.route('/logout')  
def logout():  
    session.pop('logged_in', None)  
    flash('You were logged out')  
    return redirect(url_for('login'))
```

The `logout()` function uses the `pop()` method to reset the session key to the default value when the user logs out. It then redirects the user back to the login screen and flashes a message indicating that they were logged out.

Add the following code to the *template.html* file, just before the content tag (`{% block content %}`):

```
{% for message in get_flashed_messages() %}  
    <div class="flash">{{ message }}</div>  
{% endfor %}  
{% if error %}  
    <p class="error"><strong>Error:</strong> {{ error }}</p>  
{% endif %}
```

Finally, add a logout link to the *main.html* page:

```
{% extends "template.html" %}  
{% block content %}  
<div class="jumbotron">  
    <h2>Welcome to the Flask Blog!</h2>  
    <p><a href="/logout">Logout</a></p>  
</div>  
{% endblock %}
```

Fire up the server. Test everything out. Make sure you can login and logout and that the appropriate messages are displayed.

Sessions and Login_required Decorator

Now that users are able to login and logout, we need to protect *main.html* from unauthorized access. Currently, it can be accessed without logging in. Go ahead and see for yourself: Launch the server and navigate in your browser to <http://localhost:5000/main>. This is not good.

To prevent unauthorized access to *main.html*, we need to set up sessions, as well as a `login_required` decorator. Sessions store user information server-side. In this case, when the session key, `logged_in`, is set to `True`, the user has the rights to view the *main.html* page. We already set up the sessions. Go back and take a look. The `login_required` decorator, meanwhile, checks to make sure that a user is authorized (e.g., `logged_in`) before allowing them to view a certain page. To do this, we will set up a new function which will be used to restrict access to *main.html*.

Start by importing `functools` within your application module, *blog.py*:

```
from functools import wraps
```

`functools` is a module used for extending the capabilities of functions with other functions.⁷ First, setup the new function in *blog.py*:

```
def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to login first.')
            return redirect(url_for('login'))
    return wrap
```

This tests to see if `logged_in` is in the session. If it is, then we call the method, and if not, the user is redirected back to the login screen with a message stating that a login is required.

Add the decorator to the top of the `main()` function:

```
@app.route('/main')
@login_required
def main():
    return render_template('main.html')
```

Updated code:

```
# blog.py - application module

# imports
from flask import Flask, render_template, request, session, \
    flash, redirect, url_for
import sqlite3
from functools import wraps

# configuration
DATABASE = 'fblog.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'hard to guess'

app = Flask(__name__)

# pulls in configurations by looking for UPPERCASE variables
app.config.from_object(__name__)

# method used for connecting to the database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME'] \
            or request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid Credentials. Please try again.'
        else:
            session['logged_in'] = True
            return redirect(url_for('main'))
    return render_template('login.html', error=error)

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        ...
```

```
if 'logged_in' in session:
    return test(*args, **kwargs)
else:
    flash('You need to login first.')
    return redirect(url_for('login'))
return wrap

@app.route('/main')
@login_required
def main():
    return render_template('main.html')

@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('login'))

if __name__ == '__main__':
    app.run(debug=True)
```

When a request is sent to access *main.html*, it first hits the `@login_required` function and the entire function is momentarily replaced (or wrapped) by the `login_required()` function. Then when the user is logged in, the `main()` function is invoked, allowing the user to access *main.html*. If the user is not logged in, they are redirected back to the login screen.

Test this out. But first, did you notice in the terminal that you can see the client requests as well as the server responses? After you perform each test check the server responses.

1. Login successful:

```
127.0.0.1 - - [17/Feb/2013 07:30:55] "POST / HTTP/1.1" 302
-
127.0.0.1 - - [17/Feb/2013 07:30:55] "GET /main HTTP/1.1"
200 -
```

The login credentials were sent with a POST request, the server responded with a 302, redirecting the user to *main.html*. The GET request to access *main.html* was successful, as the server responded with a 200.

2. Logout:

```
127.0.0.1 - - [17/Feb/2013 07:33:37] "GET /logout HTTP/1.1" 302 -
127.0.0.1 - - [17/Feb/2013 07:33:37] "GET / HTTP/1.1" 200 -
```

When you logged out, you are actually issuing a GET request that redirects to *login.html*. This request was successful.

3. Login failed:

```
127.0.0.1 - - [17/Feb/2013 07:36:02] "POST / HTTP/1.1" 200 -
-
```

If you enter the wrong login credentials when trying to login you still get a 200 success code as the server responds with an error.

4. Attempt to access <http://localhost:5000/main> without first logging in:

```
127.0.0.1 - - [17/Feb/2013 07:39:42] "GET /main HTTP/1.1"
302 -
127.0.0.1 - - [17/Feb/2013 07:39:42] "GET / HTTP/1.1" 200 -
```

If you try to access *main.html* without logging in first, you will be redirected back to *login.html*.

The server log comes in handy when you need to debug your code. Let's say, for example, that you forgot to add the redirect to the `login()` function (`return redirect(url_for('main'))`). If you glance at your code and can't figure out what's going on, the server log may provide a hint:

```
127.0.0.1 - - [17/Feb/2013 07:43:31] "POST / HTTP/1.1" 200 -
```

You can see that the POST request was successful, but nothing happened after. This should give you enough of a hint to know what to do. This is a rather simple case, but you will find that when your codebase grows just how handy the server log can be with respect to debugging errors.

Show Posts

Now that security is set up, we need to display some information to the user. What's the point of the user logging in in the first place? Let's start by displaying the current posts. Update the `main()` function within `blog.py`:

```
@app.route('/main')
@login_required
def main():
    g.db = connect_db()
    cur = g.db.execute('select * from posts')
    posts = [dict(title=row[0], post=row[1]) for row in
    cur.fetchall()]
    g.db.close()
    return render_template('main.html', posts=posts)
```

Here we are connecting to the database and then fetching data from the "posts" table. The data is then passed to a dictionary, which is assigned to the variable `posts`. Finally, we pass that variable to the `main.html` file.

We next need to edit `main.html` to loop through the dictionary in order to display the titles and posts:

```
{% extends "template.html" %}  
{% block content %}  
    <h2>Welcome to the Flask Blog!</h2>  
    <p><a href="/logout">Logout</a></p>  
    <br/>  
    <br/>  
    <h3>Posts:</h3>  
    {% for p in posts %}  
        <strong>Title:</strong> {{ p.title }} <br/>  
        <strong>Post:</strong> {{ p.post }} <br/>  
        <br/>  
    {% endfor %}  
{% endblock %}
```

This is a relatively straightforward example: We passed in the `posts` variable from `blog.py` that contains the data fetched from the database. Then, we used a simple for loop to iterate through the variable to display the results.

Add Posts

Finally, users need the ability to add new posts. We can start by adding a new function to `blog.py` called `add()`:

```
@app.route('/add', methods=['POST'])
@login_required
def add():
    title = request.form['title']
    post = request.form['post']
    if not title or not post:
        flash("All fields are required. Please try again.")
        return redirect(url_for('main'))
    else:
        g.db = connect_db()
        g.db.execute('insert into posts (title, post) values
(?, ?)',
                     [request.form['title'],
                     request.form['post']])
        g.db.commit()
        g.db.close()
        flash('New entry was successfully posted')
        return redirect(url_for('main'))
```

First, we used an if statement to ensure that all fields are populated with data. Then, the data is added, as a new row, to the table.

Next, add the HTML form to the *main.html* page:

```
{% extends "template.html" %}

{% block content %}

<h2>Welcome to the Flask Blog!</h2>
<p><a href="/logout">Logout</a></p>
<div class="add">
    <h3>Add a new post:</h3>
    <form action="{{ url_for('add') }}" method="post"
class="add">
        <label><strong>Title:</strong></label>
        <input name="title" type="text">
        <p><label><strong>Post:</strong></label><br/>
        <textarea name="post" rows="5" cols="40"></textarea></p>
        <input class="button" type="submit" value="Save">
    </form>
    <br/>
    <br/>
    <h3>Posts:</h3>
    {% for p in posts %}
        <strong>Title:</strong> {{ p.title }} <br/>
        <strong>Post:</strong> {{ p.post }} <br/>
        <br/>
    {% endfor %}
    {% endblock %}
```

We issued an HTTP POST request to submit the form to the `add()` function, which then redirected us back to `main.html` with the new post:

```
127.0.0.1 - - [17/Feb/2013 09:29:18] "POST /add HTTP/1.1" 302 -
127.0.0.1 - - [17/Feb/2013 09:29:18] "GET /main HTTP/1.1" 200 -
```

Test out the application.

Style

All right. Now that the app is working properly, let's make it look a bit nicer. To do this, we need to edit the HTML, CSS, and JavaScript. I'll show you a more in-depth

example in the next chapters. For now, though, let's just create something very simple:

```
.container { background: #f4f4f4; margin: 2em auto; padding: 0.8em; width: 30em; border: 2px solid #000; }

.flash, .error { background: #000; color: #fff; padding: 0.5em; }
```

Save this as `styles.css` and place it in your "static" directory. Then add a link to the external stylesheet within the head (`<head> </head>`) of the `template.html` file:

```
<link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
```

This tag is fairly straightforward. Essentially, the `url_for()` function generates a URL to the `styles.css` file. In other words, this translates to: "Look in the static folder for the file `styles.css`".



Feel free to play around with the CSS more if you'd like. If you do, send me the CSS, so I can make mine look better. :)

Conclusion

Let's recap:

1. First, we used Flask to create a basic website structure to house static pages.
2. Then we added a login form.
3. We added sessions and the `login_required` decorator to prevent unauthorized access to the `main.html` page.
4. Next, we fetched data from SQLite to show all the blog posts, then added the ability for users to add new posts.
5. Finally, we added some basic CSS styles.

Simple, right? You can now take a look at the accompanying [video](#) to see how to deploy your app on PythonAnywhere.

1. <http://lucumr.pocoo.org/2010/4/3/april-1st-post-mortem> ↵
2. <http://flask.pocoo.org/> ↵
3. <http://docs.python.org/2/glossary.html#term-decorator> ↵
4. <http://flask.pocoo.org/docs/config/> ↵
5. <http://wiki.python.org/moin/Templating> ↵
6. http://en.wikipedia.org/wiki/Session_key ↵
7. <http://docs.python.org/2/library/functools.html> ↵

6) Flask: FlaskTaskr

6.1) Overview

In this section, we will develop a task manager called **FlaskTaskr**. We'll start by creating a simple skeleton app like in the blog exercise in the last chapter but we'll be adding plenty of bells and whistles in order to make this a full-featured application. Let's get to it.

For now, this application will do the following:

1. Users sign in and out from the landing page, which is managed by sessions.
Only one user is supported.
2. Once signed in, users can add new tasks. Each task consists of a name, due date, priority, status, and an auto-incremented ID.
3. Users can view all uncompleted tasks from the same screen.
4. Users can also delete tasks and mark tasks as completed. If a user deletes a task, it will also be deleted from the database.

Before beginning take a moment to review the steps taken to create the blog application from the previous Chapter. We'll be using the same process in Lesson 6.2, but it will go much faster.

Homework

- Read about the differences between sessions and cookies [here](#).

6.2) Initial Setup and Configuration

Initially, we're going to follow a similar workflow to the blog app. To keep things simple, though, I will not repeat anything learned from the previous chapter.

Let's get started.

1. Navigate to your "realpython" directory, and create a new directory called "flasktaskr". Navigate into the newly created directory.
2. Create a new virtualenv:

```
$ virtualenv --no-site-packages env
```

3. Fire up virtualenv:

Unix:

```
$ source env/bin/activate
```

Windows:

```
$ env\scripts\activate
```

4. Install Flask:

```
$ pip install flask==0.10.1
```

5. Setup the following directories within the root directory:

```
└── app
    ├── static
    └── templates
```

Configuration

Create a configuration file called *config.py* and save it in the main directory ("flasktaskr"):

```
# config.py

import os

# grabs the folder where the script runs
basedir = os.path.abspath(os.path.dirname(__file__))

DATABASE = 'flasktask.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'my precious'

# defines the full path for the database
DATABASE_PATH = os.path.join(basedir, DATABASE)
```

Remember the configuration section of our application module from the blog app? Go back and take a look. For medium and large-sized applications, it's good to create a separate script altogether to house this information so that the application module is easier to read.

Database

Based on the info above regarding the main functionality of the app, we need one database table, consisting of these fields - "task_id", "name", "due_date", "priority", and "status". The value of status will either be a 1 or 0: 1 if the task is open and 0 if closed.

```
# db_create.py

import sqlite3
from config import DATABASE_PATH

with sqlite3.connect(DATABASE_PATH) as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # create the table
    c.execute("""CREATE TABLE ftasks(task_id INTEGER PRIMARY
KEY AUTOINCREMENT,
        name TEXT NOT NULL, due_date TEXT NOT NULL, priority
INTEGER NOT NULL,
        status INTEGER NOT NULL)""")

    # insert dummy data into the table
    c.execute('INSERT INTO ftasks (name, due_date, priority,
status) VALUES("Finish this tutorial", "02/03/2013", 10, 1)')
    c.execute('INSERT INTO ftasks (name, due_date, priority,
status) VALUES("Finish my book", "02/03/2013", 10, 1)')
```

Two things of note:

1. Notice how we did not need to specify the "task_id" when entering data into the table as it's an auto-incremented value, which means that it's auto-generated with each new row of data. Also, we used a status of `1` to indicate that each of those tasks are considered "open" tasks.
2. We imported the `DATABASE_PATH` variable from the configuration file we created just a second ago.

Save the file as `db_create.py` under the root directory and run it. Was the table created? Did it populate with data? How do you check? SQL Database Browser.

Application Module/Controller

Create an application module (controller) called `views.py` and fill it with the following code:

```
# views.py

from flask import Flask, flash, redirect, render_template,
request, \
    session, url_for, g
from functools import wraps
import sqlite3

app = Flask(__name__)
app.config.from_object('config')

def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to login first.')
            return redirect(url_for('login'))
    return wrap

@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    flash('You are logged out. Bye. :(')
    return redirect(url_for('login'))

@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            or request.form['password'] != app.config['PASSWORD']:
                error = 'Invalid Credentials. Please try again.'
    else:
```

```
    session['logged_in'] = True
    return redirect(url_for('tasks'))
return render_template('login.html', error=error)
```

Note: we're saving this as views.py, since the controller holds all the views.

Save this file in the "app" directory. You've seen this all before. Right now, we have one view, *login.html*, which is mapped to the main URL, `'/'`. Sessions and the `login_required` decorator are set-up. You can see that after you login, you are redirected to `tasks`, which still needs to be specified. Please refer to the blog application from the previous chapter for further explanation on any details of this code that you do not understand.

Let's go ahead and setup the login template, base template, and stylesheet.

Templates and Styles

Login template:

```
{% extends "template.html" %}
{% block content %}
<h1>Welcome to FlaskTaskr.</h1>
<h3>Please login to access your task list.</h3>
<form action="" method="post">
    Username: <input type="text" name="username" value="{{ request.form.username }}">
    Password: <input type="password" name="password" value="{{ request.form.password }}">
    <input type="submit" value="Login">
</form>
<p><em>Use 'admin' for the username and password.</em></p>
{% endblock %}
```

Save this as *login.html* in the templates directory.

Base template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to FlaskTaskr!!</title>
    <link rel="stylesheet" href="{{ url_for('static',
filename='styles.css') }}">
  </head>
  <body>
    <div class="page">
      {%
        for message in get_flashed_messages() %
      <div class="flash">{{ message }}</div>
      <br/>
      {% endfor %}
      {% if error %}
        <div class="error"><strong>Error:</strong> {{ error
}}</div>
      {% endif %}
      {% block content %}
      {% endblock %}
    </div>
  </body>
</html>
```

Save this as *template.html* in the templates directory. Do you remember the relationship between the parent and child templates?

We'll temporarily "borrow" the majority of the stylesheet from the Flask [website](#). Please copy and paste this.

```
body {  
    font-family: sans-serif;  
    background: #eee;  
}  
  
a, h1, h2 { color: #377BA8; }  
  
h1, h2 {  
    font-family: 'Georgia', serif;  
    margin: 0;  
}  
  
h1 { border-bottom: 2px solid #eee; }  
  
h2 { font-size: 1.5em; }  
  
.page {  
    margin: 2em auto;  
    width: 50em;  
    border: 5px solid #ccc;  
    padding: 0.8em;  
    background: white;  
}  
  
.entries {  
    list-style: none;  
    margin: 0;  
    padding: 0;  
}  
  
.entries li { margin: 0.8em 1.2em; }  
  
.entries li h2 { margin-left: -1em; }  
  
.add-task {  
    font-size: 0.9em;  
    border-bottom: 1px solid #ccc;  
}  
.add-task dl { font-weight: bold; }
```

```
.metanav {  
    text-align: right;  
    font-size: 0.8em;  
    padding: 0.3em;  
    margin-bottom: 1em;  
    background: #fafafa;  
}  
  
.flash {  
    background: #CEE5F5;  
    padding: 0.5em;  
}  
  
.error {  
    background: #F0D6D6;  
    padding: 0.5em;  
}  
  
.datagrid table {  
    border-collapse: collapse;  
    text-align: left;  
    width: 100%;  
}  
  
.datagrid {  
    background: #fff;  
    overflow: hidden;  
    border: 1px solid #000000;  
    -webkit-border-radius: 3px;  
    -moz-border-radius: 3px;  
    border-radius: 3px;  
}  
.datagrid table td, .datagrid table th { padding: 3px 10px; }  
.datagrid table thead th {  
    background:-webkit-gradient(linear, left top, left bottom,  
    color-stop(0.05, #000000), color-stop(1, #000000));  
    background:-moz-linear-gradient(center top, #000000 5%,  
    #000000 100%);
```

```
filter:progid:DXImageTransform.Microsoft.gradient(startColorstr='#000000',
endColorstr='#000000');
background-color:#000000;
color:#FFFFFF; font-size:
15px; font-weight:
bold;
}
.datagrid table thead th:first-child { border: none; }
.datagrid table tbody td {
    color: #000000;
border-left: 1px solid #E1EEF4;
font-size: 12px;
font-weight: normal;
}
.datagrid table tbody .alt td {
    background: #E1EEF4;
    color: #000000;
}
.datagrid table tbody td:first-child { border-left: none; }
.datagrid table tbody tr:last-child td { border-bottom: none; }

.button {
background-color:#000;
display:inline-block;
color:#ffffff;
font-size:13px;
padding:3px 12px;
margin: 0;
text-decoration:none;
position:relative;
}
```

Save this as `styles.css` in the "static" directory.

Lastly, create a module that we will use to run the application:

```
# run.py

from app.views import app
app.run(debug=True)
```

Instead of running the application from the application module. We created this separate file to handle it. Save it as *run.py* in the "app" directory. Before running, though, create a blank *__init__.py* file, which establishes our app as a Python Project. Save this in the "app" directory as well.

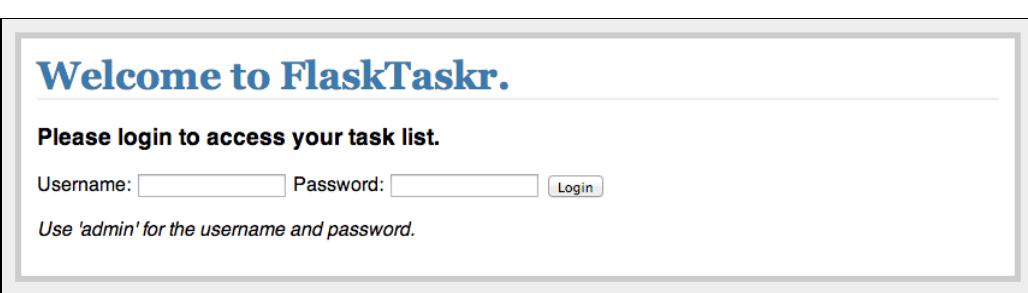
Your project structure should now look like this:

```
└── app
    ├── __init__.py
    ├── static
    │   └── styles.css
    ├── templates
    │   ├── login.html
    │   ├── tasks.html
    │   └── template.html
    └── views.py
├── config.py
├── db_create.py
└── flasktask.db
└── run.py
```

Fire up the server:

```
$ python run.py
```

Make sure everything works thus far. You'll only be able to view the login page (but not login) and the styles, as we have not setup the *tasks.html* page yet.



Tasks

The *tasks.html* page will be quite a bit different than the *main.html* page from our blog as the user will have the ability to delete tasks and mark tasks as complete rather than just being able to add new rows (tasks) to the table.

Let's start by adding the function to the *views.py* file:

```
@app.route('/tasks/')
@login_required
def tasks():
    g.db = connect_db()
    cur = g.db.execute('select name, due_date, priority,
task_id from ftasks where status=1')
    open_tasks = [dict(name=row[0], due_date=row[1],
priority=row[2], task_id=row[3]) for row in cur.fetchall()]
    cur = g.db.execute('select name, due_date, priority,
task_id from ftasks where status=0')
    closed_tasks = [dict(name=row[0], due_date=row[1],
priority=row[2], task_id=row[3]) for row in cur.fetchall()]
    g.db.close()
    return render_template('tasks.html', open_tasks=open_tasks,
closed_tasks=closed_tasks)
```

We queried the database for open and closed tasks, saving the results to two variables, and then passing those variables, `open_tasks` and `closed_tasks`, to the *tasks.html* page. These variables will then be used to populate the open and closed task lists.

Next, we need to add the ability to add new tasks, mark tasks as complete, and delete tasks. Add each of these three functions to the `views.py` file:

```
# Add new tasks:  
@app.route('/add/', methods=['POST'])  
@login_required  
def new_task():  
    g.db = connect_db()  
    name = request.form['name']  
    date = request.form['due_date']  
    priority = request.form['priority']  
    if not name or not date or not priority:  
        flash("All fields are required. Please try again.")  
        return redirect(url_for('tasks'))  
    else:  
        g.db.execute('insert into ftasks (name, due_date,  
priority, status) values (?, ?, ?, 1)',  
[request.form['name'], request.form['due_date'],  
request.form['priority']])  
        g.db.commit()  
        g.db.close()  
        flash('New entry was successfully posted. Thanks.')  
        return redirect(url_for('tasks'))  
  
# Mark tasks as complete:  
@app.route('/complete/<int:task_id>/',)  
@login_required  
def complete(task_id):  
    g.db = connect_db()  
    cur = g.db.execute('update ftasks set status = 0 where  
task_id='+str(task_id))  
    g.db.commit()  
    g.db.close()  
    flash('The task was marked as complete.')  
    return redirect(url_for('tasks'))  
  
# Delete Tasks:  
@app.route('/delete/<int:task_id>/',)  
@login_required  
def delete_entry(task_id):  
    g.db = connect_db()  
    cur = g.db.execute('delete from ftasks where
```

```
task_id='+str(task_id))
g.db.commit()
g.db.close()
flash('The task was deleted.')
return redirect(url_for('tasks'))
```

The last two functions pass in a variable, `task_id` from the `tasks.html` page (which we will create next). This variable is equal to the `task_id` field in the database. A query is then performed and the appropriate action takes place. In this case, an action means either marking a task as complete or deleting a task. Notice how we have to convert the `task_id` variable to a string, since we are using concatenation to combine the SQL query to the `task_id`, which is an integer.

Tasks Template:

```
{% extends "template.html" %}  
{% block content %}  
    <h1>Welcome to FlaskTaskR</h1>  
    <a href="/logout">Logout</a>  
    <div class="add-task">  
        <h3>Add a new task:</h3>  
        <table>  
            <tr>  
                <form action="{{ url_for('new_task') }}" method="post">  
                    <td>  
                        <label>Task Name:</label>  
                        <input name="name" type="text">  
                    </td>  
                    <td>  
                        <label>Due Date (mm/dd/yyyy):</label>  
                        <input name="due_date" type="text" width="120px">  
                    </td>  
                    <td>  
                        <label>Priority:</label>  
                        <select name="priority" width="100px">  
                            <option value="1">1</option>  
                            <option value="2">2</option>  
                            <option value="3">3</option>  
                            <option value="4">4</option>  
                            <option value="5">5</option>  
                            <option value="6">6</option>  
                            <option value="7">7</option>  
                            <option value="8">8</option>  
                            <option value="9">9</option>  
                            <option value="10">10</option>  
                        </select>  
                    </td>  
                    <td>  
                        &nbsp;  
                        &nbsp;  
                        <input class="button" type="submit" value="Save">  
                    </td>  
                </form>  
            </tr>
```

```
</table>
</div>
<div class="entries">
<br/>
<br/>
<h2>Open tasks:</h2>
<div class="datagrid">
<table>
    <thead>
        <tr>
            <th width="300px"><strong>Task Name</strong></th>
            <th width="100px"><strong>Due Date</strong></th>
            <th width="50px"><strong>Priority</strong></th>
            <th><strong>Actions</strong></th>
        </tr>
    </thead>
    {% for o in open_tasks %}
    <tr>
        <td width="300px">{{ o.name }}</td>
        <td width="100px">{{ o.due_date }}</td>
        <td width="50px">{{ o.priority }}</td>
        <td>
            <a href="{{ url_for('delete_entry', task_id = o.task_id) }}">Delete</a> -
            <a href="{{ url_for('complete', task_id = o.task_id) }}">Mark as Complete</a>
        </td>
    </tr>
    {% endfor %}
</table>
</div>
<br/>
<br/>
<div class="entries">
<h2>Closed tasks:</h2>
<div class="datagrid">
<table>
    <thead>
        <tr>
```

```
<th width="300px"><strong>Task Name</strong></th>
<th width="100px"><strong>Due Date</strong></th>
<th width="50px"><strong>Priority</strong></th>
<th><strong>Actions</strong></th>
</tr>
</thead>
{%
  for c in closed_tasks %}
  <tr>
    <td width="300px">{{ c.name }}</td>
    <td width="100px">{{ c.due_date }}</td>
    <td width="50px">{{ c.priority }}</td>
    <td>
      <a href="{{ url_for('delete_entry', task_id = c.task_id) }}">Delete</a>
    </td>
  </tr>
  {% endfor %}
</table>
</div>
{%
  endblock %}
```

Save this as *tasks.html* in the templates directory.

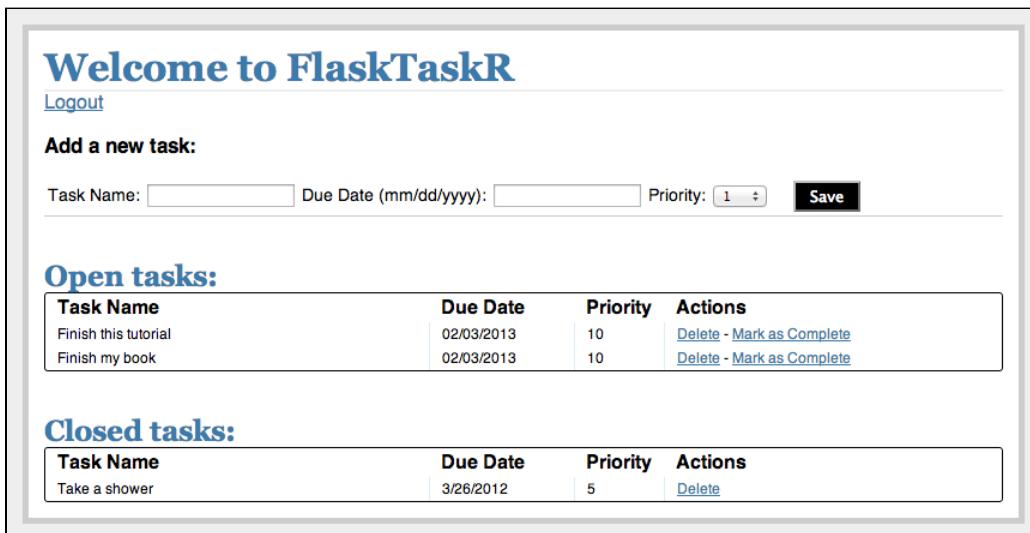
Although a lot is going on in here, the only new things are these statements:

```
<a href="{{ url_for('delete_entry', task_id = o.task_id) }}"
}>Delete</a>

<a href="{{ url_for('complete', task_id = o.task_id) }}">Mark
as Complete</a>
```

Essentially, we pulled the `task_id` from the database dynamically from each row in the database table as the for loop progresses. We then assigned it to a variable, also named `task_id`, which is then passed back to either the `delete()` function -
`@app.route('/delete/<int:task_id>/',)` - or the `complete()` function -
`@app.route('/complete/<int:task_id>/',)`.

Finally, go ahead and test out the functionality of the app. Fire up the server:
`python run.py`. If you get any errors, be sure to double check your code.



The screenshot shows a web application titled "Welcome to FlaskTaskR". At the top, there is a "Logout" link. Below it, a section titled "Add a new task:" contains input fields for "Task Name", "Due Date (mm/dd/yyyy)", "Priority" (set to 1), and a "Save" button. Below this, a section titled "Open tasks:" displays a table with two rows:

| Task Name | Due Date | Priority | Actions |
|----------------------|------------|----------|---|
| Finish this tutorial | 02/03/2013 | 10 | Delete - Mark as Complete |
| Finish my book | 02/03/2013 | 10 | Delete - Mark as Complete |

Below the open tasks, a section titled "Closed tasks:" displays a table with one row:

| Task Name | Due Date | Priority | Actions |
|---------------|-----------|----------|------------------------|
| Take a shower | 3/26/2012 | 5 | Delete |

6.3) Bells and Whistles

Now that we have a functional app, let's add some extensions so that the application can scale much easier. We will be looking at:

- Database Management via SQLAlchemy,
- User Registration,
- User Logins,
- Database Relationships,
- Managing Sessions,
- Error Handling,
- Unit Testing,
- Styling,
- Blueprints, and
- Deployment on Heroku.

Homework

- Please read over the [mainpage](#) of the Flask-SQLAlchemy extension. Compare the code samples to regular SQL. How do the classes compare to the SQL statements used for creating a new table?
- Take a look at all the Flask extensions [here](#). Read them over quickly.

6.4) Database Management via SQLAlchemy

As mentioned in Chapter 2, you can work with relational databases without learning SQL. Essentially, you need to use an Object Relational Mapper (ORM), which translates and maps SQL commands, and your entire database, into Python objects. It makes working with relational databases much easier as it eliminates having to write repetitive code. ORMs also make your application database agnostic, meaning you can switch SQL database engines without having to re-write the code that interacts with the database itself.

That said, no matter how much you use an ORM, you will eventually have to use SQL for troubleshooting or testing quick, one-off queries as well as advanced queries. It's also really, really helpful to know SQL, when trying to decide on the most efficient way to query the database, to know what calls the ORM will be making to the database. Learn SQL first, in other words. :)

We will be using the Flask-SQLAlchemy extension, which is a type of ORM, to manage our database.¹ Let's jump right in.

Create a Table

Forewarning: We'll be running a number of scripts in this subsection with little to no explanation. Don't worry. A detailed explanation will come at the end.

Start by installing Flask-SQLAlchemy root project directory:

- Fire up your virtualenv, then run the following command:

```
$ pip install Flask-SQLAlchemy==1.0
```

Delete `flasktask.db`, and then create a new file called `models.py` in the "app" directory. We're going to recreate the database using SQLAlchemy.

```
# models.py

from app import db

class FTasks(db.Model):

    __tablename__ = "ftasks"

    task_id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    due_date = db.Column(db.Date, nullable=False)
    priority = db.Column(db.Integer, nullable=False)
    status = db.Column(db.Integer)

    def __init__(self, name, due_date, priority, status):
        self.name = name
        self.due_date = due_date
        self.priority = priority
        self.status = status

    def __repr__():
        return '<name %r>' % (self.name)
```

We also have to transfer the creation of the Flask application object from `views.py` to `__init__.py`.

```
# __init__.py

from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

from app import views, models
```

The above file was created so that our modules (like the views and models modules) will be able to access the app and db objects.

Now *remove* the following lines from *views.py*:

```
import sqlite3

app = Flask(__name__)
app.config.from_object('config')

def connect_db():
    return sqlite3.connect(app.config['DATABASE'])
```

Then add the following:

```
from app import app, db
```

Also, update your *config.py* file:

```
# config.py

import os

# grabs the folder where the script runs
basedir = os.path.abspath(os.path.dirname(__file__))

DATABASE = 'flasktaskr.db'
USERNAME = 'admin'
PASSWORD = 'admin'
SECRET_KEY = 'my precious'

# defines the full path for the database
DATABASE_PATH = os.path.join(basedir, DATABASE)

# the database uri
SQLALCHEMY_DATABASE_URI = 'sqlite:///{} + DATABASE_PATH
```

And update *run.py*:

```
# run.py

from app import app
app.run(debug=True)
```

Lastly, update *db_create.py*.

```
# db_create.py

from app import db
from app.models import FTasks
from datetime import date

# create the database and the db table
db.create_all()

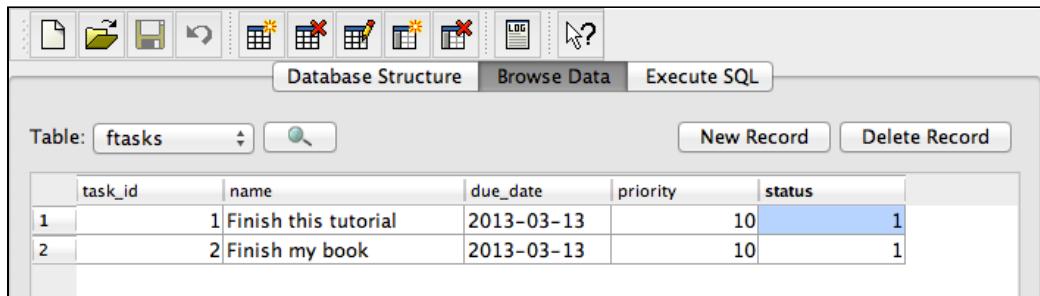
# insert data
db.session.add(FTasks("Finish this tutorial", date(2013, 3,
13), 10, 1))
db.session.add(FTasks("Finish my book", date(2013, 3, 13), 10,
1))

# commit the changes
db.session.commit()
```

Save all the files, and run the script:

```
$ python db_create.py
```

The *flasktask.db* should have been recreated. Open up SQLite Browser to ensure that the table and data above are present in the *ftasks* table.



The screenshot shows the SQLite Browser interface. At the top, there's a toolbar with various icons for database management. Below the toolbar, a menu bar has three tabs: "Database Structure", "Browse Data", and "Execute SQL". The "Browse Data" tab is currently selected. In the center, there's a search bar with the text "Table: ftasks" and a magnifying glass icon. To the right of the search bar are two buttons: "New Record" and "Delete Record". Below these controls is a table with five columns: "task_id", "name", "due_date", "priority", and "status". The table contains two rows of data:

| | task_id | name | due_date | priority | status |
|---|---------|----------------------|------------|----------|--------|
| 1 | 1 | Finish this tutorial | 2013-03-13 | 10 | 1 |
| 2 | 2 | Finish my book | 2013-03-13 | 10 | 1 |

So, what's going on in the scripts?

1. In *models.py* we have one class that defined the *ftasks* table:

```
task_id = db.Column(db.Integer, primary_key=True)
name = db.Column(db.String, nullable=False)
due_date = db.Column(db.Date, nullable=False)
priority = db.Column(db.Integer, nullable=False)
status = db.Column(db.Integer)
```

Note that variable names are used as the column names, any field that has a `primary_key` set to `True` will auto-increment.

2. In `config.py` we defined the `SQLALCHEMY_DATABASE_URI` to tell SQLAlchemy where to access the database.
3. In `db_create.py`, we initialized the database schema by calling `db.create_all()`.
4. We then populated it with some data. We used the `FTASKS` object to specify the data we wanted:

```
db.session.add(FTasks("Finish this tutorial", date(2013,
3, 13), 10, 1))
db.session.add(FTasks("Finish my book", date(2013, 3, 13),
10, 1))
```

To apply the above inserts to our database we needed to commit:

```
db.session.commit()
```

```
# views.py

from app import app, db
from flask import Flask, flash, redirect, render_template,
request, session, url_for, g
from functools import wraps
from app.forms import AddTask
from app.models import FTasks

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to login first.')
            return redirect(url_for('login'))
    return wrap

@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    flash('You are logged out. Bye. :(')
    return redirect (url_for('login'))

@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME'] or request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid Credentials. Please try again.'
        else:
            session['logged_in'] = True
            return redirect(url_for('tasks'))
    return render_template('login.html', error=error)
```

```
@app.route('/tasks/')
@login_required
def tasks():
    open_tasks =
        db.session.query(FTasks).filter_by(status='1').order_by(
            FTasks.due_date.asc())
    closed_tasks =
        db.session.query(FTasks).filter_by(status='0').order_by(
            FTasks.due_date.asc())
    return render_template('tasks.html', form =
        AddTask(request.form),
        open_tasks=open_tasks,
        closed_tasks=closed_tasks)

# Add new tasks:
@app.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    form = AddTask(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_task = FTasks(
            form.name.data,
            form.due_date.data,
            form.priority.data,
            '1'
        )
        db.session.add(new_task)
        db.session.commit()
        flash('New entry was successfully posted. Thanks.')
        return redirect(url_for('tasks'))

# Mark tasks as complete:
@app.route('/complete/<int:task_id>/',)
@login_required
def complete(task_id):
    new_id = task_id

    db.session.query(FTasks).filter_by(task_id=new_id).update({"status":"0"})
    db.session.commit()
```

```
        flash('The task was marked as complete. Nice.')
        return redirect(url_for('tasks'))\n\n# Delete Tasks:\n@app.route('/delete/<int:task_id>',)\n@login_required\ndef delete_entry(task_id):\n    new_id = task_id\n\n    db.session.query(FTasks).filter_by(task_id=new_id).delete()\n    db.session.commit()\n    flash('The task was deleted. Why not add a new one?')\n    return redirect(url_for('tasks'))
```

Since we are now using SQLAlchemy, we've modified the way we do database queries, INSERT and DELETE. The code is much cleaner. Take a look. Compare it with the actual SQL code from the beginning of the chapter.

6.5) User Registration

We're now going to use another powerful Flask extension called WTForms, which helps with form handling and data validation.² First, install the package. Make sure virtualenv is activated.

```
pip install Flask-WTF==0.8.4
```

Now let's create a new file called *forms.py* and add the following code:

```
# forms.py

from flask.ext.wtf import Form, TextField, DateField,
IntegerField, \
    SelectField, Required

class AddTask(Form):
    task_id = IntegerField('Priority')
    name = TextField('Task Name', validators=[Required()])
    due_date = DateField('Date Due (mm/dd/yyyy)', \
        validators=[Required()],
        format='%m/%d/%Y')
    priority = SelectField('Priority', validators=[Required()], \
        choices=[('1', '1'), ('2',
        '2'), ('3', '3'),
        ('4', '4'), ('5', '5')])
    status = IntegerField('Status')
```

Save it in "app" directory. As the name suggests, the validators, validate the data submitted by the user. For example, `Required` simply means that the field cannot be blank, while the format validator restricts the input to the MM/DD/YY date format.

Fire up your server: `python run.py`. Ensure that you can still view tasks, add new tasks, mark tasks as complete, and delete tasks.

Let's allow multiple users to access the task manager by setting up a user registration form.

Create a new table

First we need to create a new table in our database to house user data. To do so, just add a new class to `models.py`:

```
class User(db.Model):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)

    def __init__(self, name=None, email=None, password=None):
        self.name = name
        self.email = email
        self.password = password

    def __repr__(self):
        return '<User %r>' % (self.name)
```

Run `db_create.py` again. Open up SQLite Browser. Notice how it ignores the table already created and just creates the `users` table:

| Name | Object | Type |
|--------------------------|--------|---------------------|
| ▼ftasks | table | |
| task_id | field | INTEGER PRIMARY KEY |
| name | field | VARCHAR |
| due_date | field | DATE |
| priority | field | INTEGER |
| status | field | INTEGER |
| ▼users | table | |
| id | field | INTEGER PRIMARY KEY |
| name | field | VARCHAR |
| email | field | VARCHAR |
| password | field | VARCHAR |
| sqlite_autoindex_users_1 | index | |
| sqlite_autoindex_users_2 | index | |

Configuration

Update the configuration module, `config.py`:

Remove the following lines of code:

```
USERNAME = 'admin'  
PASSWORD = 'admin'
```

We no longer need this configuration since we will use the information from the *users* table in the database.

We also need to update *forms.py* to cater for registration and logging in:

```
# forms.py

from flask.ext.wtf import Form, TextField, PasswordField,
DateField, IntegerField, SelectField
from flask.ext.wtf import Required, Email, EqualTo, Length

class RegisterForm(Form):
    name = TextField('Username', validators=[Required(),
Length(min=6, max=25)])
    email = TextField('Email', validators=[Required(),
Length(min=6, max=40)])
    password = PasswordField('Password',
validators=[Required(), Length(min=6, max=40)])
    confirm = PasswordField('Repeat Password',[Required(),
EqualTo('password', message='Passwords must match'))]

class LoginForm(Form):
    name = TextField('Username', validators=[Required()])
    password = PasswordField('Password',
validators=[Required()])

class AddTask(Form):
    task_id = IntegerField('Priority')
    name = TextField('Task Name', validators=[Required()])
    due_date = DateField('Date Due (mm/dd/yyyy)', validators=[Required()],format='%m/%d/%Y')
    priority = SelectField('Priority',
validators=[Required()],choices=[('1', '1'), ('2', '2'), ('3', '3'),
('4', '4'), ('5', '5')])
    status = IntegerField('Status')
```

Next we need to update the Application Module (*views.py*):

1. Update the imports, and add the following code:

```
from app.forms import AddTask, RegisterForm, LoginForm
from app.models import FTasks, User
```

These allow us to use the `RegisterForm` and `LoginForm` classes from `forms.py` and the `User` class from `model.py`.

2. Add the new view function, `register()`:

```
@app.route('/register/', methods=['GET', 'POST'])
def register():
    error = None
    form = RegisterForm(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_user = User(
            form.name.data,
            form.email.data,
            form.password.data,
        )
        db.session.add(new_user)
        db.session.commit()
        flash('Thanks for registering. Please login.')
        return redirect(url_for('login'))
    return render_template('register.html', form=form,
                           error=error)
```

Here, the user information, which will be passed from the `register.html` template, is stored inside the variable `new_user`. That data is then stored in the database, and after successful registration, the user is redirected to `login.html` with a message thanking them for registering.

`validate_on_submit()` returns either True or False depending on whether the submitted data passes the form validators associated with each field in the form.

Templates

Registration:

```
{% extends "template.html" %}  
{% block content %}  
    <h1>Welcome to FlaskTaskr.</h1>  
    <h3>Please register to access the task list.</h3>  
    <form method="POST" action="">  
        <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;{{  
            form.email.label }}: {{ form.email }}</p>  
        <p>{{ form.password.label }}: {{ form.password }}&nbsp;&nbsp;{{  
            form.confirm.label }}: {{ form.confirm }}</p>  
        <p><input type="submit" value="Register"></p>  
    </form>  
    <p><em>Already registered?</em> Click <a href="/">here</a>  
to login.</p>  
    {% endblock %}
```

Save this as *register.html* under your templates directory.

Now let's add a registration link to the *login.html* page:

```
<p><em>Need an account? </em><a  
    href="/register">Signup! !</a></p>
```

Let's go ahead and test it out. Load the server, click the link to register, and register a new user. You should be able to register just fine, but we need to update the code so users can login. Everything turn out okay? Double check my code, if not.

6.6) User Login

The next step for allowing multiple logins is to change the `login()` function on the application module as well as the login template.

Application Module

1. Replace the current `login()` function with:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method=='POST':
        u = User.query.filter_by(name=request.form['name'],
                                 password=request.form['password']).first()
        if u is None:
            error = 'Invalid username or password.'
        else:
            session['logged_in'] = True
            flash('You are logged in. Go Crazy.')
            return redirect(url_for('tasks'))

    return render_template("login.html",
                           form = LoginForm(request.form),
                           error = error)
```

This code is not too much different than the old code. When a user submits their user credentials via a POST request, the database is queried for the submitted username and password. If the credentials are not found, an error populates; otherwise, the user is logged in and redirected to *tasks.html*.

Templates

Update *login.html* with the following code:

```
{% extends "template.html" %}  
{% block content %}  
    <h1>Welcome to FlaskTaskr.</h1>  
    <h3>Please login to access your task list.</h3>  
    <form method="post" action="">  
        <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;{{  
            form.password.label }}: {{ form.password }}&nbsp;&nbsp;<input  
            type="submit" value="Submit"></p>  
    </form>  
    <p><em>Need an account? </em><a  
        href="/register">Signup! !</a></p>  
    {% endblock %}
```

Test it out. Try logging in with the same user you registered. If done correctly, you should get logged in and redirected to *tasks.html*. Check out the server logs:

```
127.0.0.1 - - [18/Feb/2013 21:02:19] "POST / HTTP/1.1" 302 -  
127.0.0.1 - - [18/Feb/2013 21:02:19] "GET /tasks HTTP/1.1" 200  
-
```

Can you tell what happened? Can you predict what the server log will look like when you submit a bad username and/or password? Try it.

6.7) Database Relationships

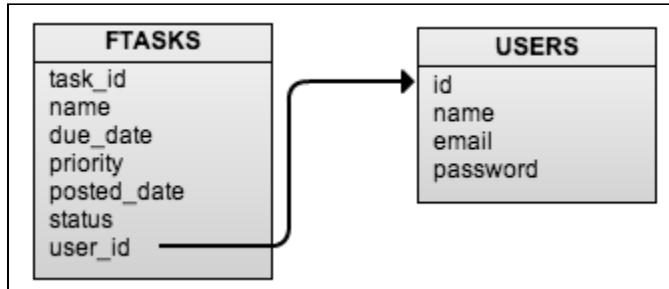
To complete the conversion to SQLAlchemy we need to update the tasks template and views.

First, let's update the database to add two new fields: "posted_date" and "user_id" to the FTASKS table. The "user_id" field needs to link back to the User table.

Database relationships

We briefly touched on the subject of relationally linking tables in Chapter 2, but essentially relational databases are designed to connect tables together using unique fields. By linking (or binding) the "id" field from the users table with the

"user_id" field from the FTASKS table, we can do basic SQL queries to find out who created a certain task as well as find out all the tasks created by a certain user:



Let's look at how to alter the tables to create such relationships within *model.py*.

Add the following field to the "ftasks" table:

```
user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

And this field to the "users" table:

```
tasks = db.relationship('FTasks', backref='poster')
```

The "user_id" field in the ftasks table is a foreign key, which binds the values from this field to the values found in the "id" field of the users table. Foreign keys are essential for creating relationships between tables in order to correlate information.

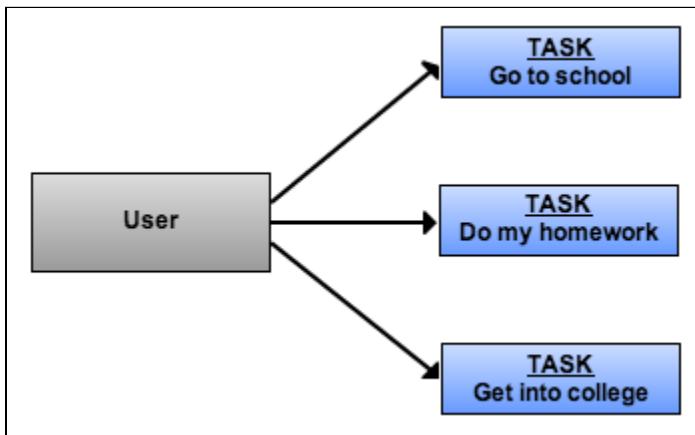
Need help with foreign keys? Take a look at the W3C [documentation](#).

Now, in a relational database there are three basic relationships.

For example:

1. One to One (1:1) - *one* employee is assigned *one* employee id
2. One to Many (1:M) - *one* department contains *many* employees
3. Many to Many (M:M) - *many* employees take *many* training courses

In our case, we have a one to many relationship: *one* user can post *many* tasks:



If we were to create a more advanced application we could also have a many to many relationship: *many* users could alter *many* tasks. However, we will keep this database simple: one user can create a task, one user can mark that same task as complete, and one user can delete the task.

Your ForeignKey() and relationship() functions are dependent on the type of relationship. In most One to Many relationships the ForeignKey is placed on the "many" side, while the relationship() is on the "one" side. The new field associated with the relationship() function is not an actual field in the database. Instead, it simply references back to the objects associated with the "many" side. I know this is confusing right now, but it should become clear after we go through an example.

We also need to update the imports as well as add another field, "posted_date", to the FTASKS class:

```
# models.py

from app import db

class FTasks(db.Model):

    __tablename__ = "ftasks"

    task_id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    due_date = db.Column(db.Date, nullable=False)
    priority = db.Column(db.Integer, nullable=False)
    posted_date = db.Column(db.Date, nullable=False)
    status = db.Column(db.Integer)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))

    def __init__(self, name, due_date, priority, posted_date,
                 status, user_id):
        self.name = name
        self.due_date = due_date
        self.priority = priority
        self.posted_date = posted_date
        self.status = status
        self.user_id = user_id

    def __repr__(self):
        return '<name %r>' % (self.body)

class User(db.Model):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)
    tasks = db.relationship('FTasks', backref='poster')
```

```
def __init__(self, name=None, email=None, password=None):
    self.name = name
    self.email = email
    self.password = password

def __repr__(self):
    return '<User %r>' % (self.name)
```

The above code will work if we will be using a fresh, empty database. But since our database already has the ftasks and users tables, SQLAlchemy will not try to redefine these database tables. We need a migration script that will update the schema and transfer any existing data:

```
# db_migrate.py

from app import db
from datetime import datetime
from config import DATABASE_PATH
import sqlite3

with sqlite3.connect(DATABASE_PATH) as connection:

    # get a cursor object used to execute SQL commands
    c = connection.cursor()

    # temporarily change the name of ftasks table
    c.execute("""ALTER TABLE ftasks RENAME TO old_ftasks""")

    # recreate a new ftasks table with updated schema
    db.create_all()

    # retrieve data from old_ftasks table
    c.execute("""SELECT name, due_date, priority,
                  status FROM old_ftasks ORDER BY task_id ASC""")

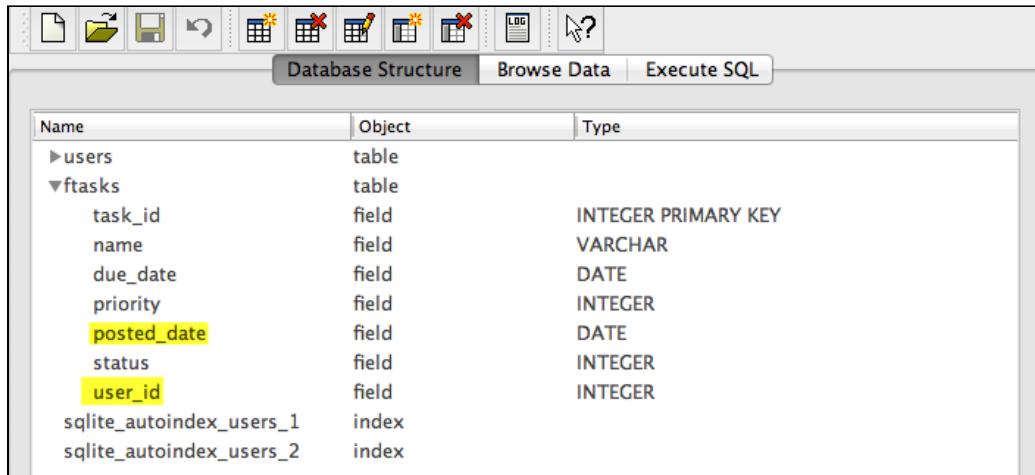
    # save all rows as a list of tuples; set posted_date to now
    # and user_id to 1
    data = [(row[0], row[1], row[2], row[3],
              datetime.now(), 1) for row in c.fetchall()]

    # insert data to ftasks table
    c.executemany("""INSERT INTO ftasks (name, due_date,
                                         priority, status,
                                         posted_date, user_id) VALUES (?, ?, ?, ?, ?,
                                         ?)""", data)

    # delete old_ftasks table
    c.execute("DROP TABLE old_ftasks")
```

Save as `db_migrate.py` under the root directory and run it.

Note that this script did not touch the "users" table; it is only the "ftasks" table that has underlying schema changes. Using SQLite Browser, verify that the *posted_date* and *user_id* columns have been added to the ftasks table.



The screenshot shows the SQLite Browser interface with the 'Database Structure' tab selected. A table named 'ftasks' is expanded, showing its columns: task_id, name, due_date, priority, posted_date, status, and user_id. The 'posted_date' and 'user_id' columns are highlighted with yellow boxes. The table also lists two indexes: sqlite_autoindex_users_1 and sqlite_autoindex_users_2.

| Name | Object | Type |
|--------------------------|--------|---------------------|
| ►users | table | |
| ▼ftasks | table | |
| task_id | field | INTEGER PRIMARY KEY |
| name | field | VARCHAR |
| due_date | field | DATE |
| priority | field | INTEGER |
| posted_date | field | DATE |
| status | field | INTEGER |
| user_id | field | INTEGER |
| sqlite_autoindex_users_1 | index | |
| sqlite_autoindex_users_2 | index | |

Forms

We now need to add the *posted_date* field to the AddTask form. Open *forms.py* and add the following line at the end of AddTask class:

```
posted_date = DateField('Posted Date (mm/dd/yyyy)',  
validators=[Required()], format='%m/%d/%Y')
```

Application Module

We also need to update our view for adding new tasks. Within the *new_task* function.

Change the following:

```
new_task = FTasks (
    form.name.data,
    form.due_date.data,
    form.priority.data,
    '1'
)
```

to:

```
new_task = FTasks (
    form.name.data,
    form.due_date.data,
    form.priority.data,
    form.posted_date.data,
    '1',
    '1'
)
```

We've added `form.posted_date.data` and `'1'`. The former simply captures the form data entered by the user on the `posted_date` field and passes it to the `FTasks` class, while the latter assigns `user_id` to 1. This means that the any task that we create is owned by the first user in the `users` database table. This is ok if we only have one user; later in subsequent section, we will change this to capture the `user_id` of the currently logged-in user.

Templates

Now, let's update the `tasks.html` template:

```
{% extends "template.html" %}

{% block content %}

    <h1>Welcome to FlaskTaskR</h1>
    <br/>
    <a href="/logout">Logout</a>
    <div class="add-task">
        <h3>Add a new task:</h3>
        <form method="POST" action="{{ url_for('new_task') }}">
            <p>{{ form.name.label }}: {{ form.name }}<br />{{ form.due_date.label }}:<br />{{ form.due_date }}&ampnbsp{{ form.posted_date.label }}:<br />{{ form.posted_date }}&ampnbsp{{ form.priority.label }}: {{ form.priority }}</p>
            <p><input type="submit" value="Submit"></p>
        </form>
    </div>
    <div class="entries">
        <br/>
        <br/>
        <h2>Open tasks:</h2>
        <div class="datagrid">
            <table>
                <thead>
                    <tr>
                        <th width="200px"><strong>Task Name</strong></th>
                        <th width="75px"><strong>Due Date</strong></th>
                        <th width="100px"><strong>Posted Date</strong></th>
                        <th width="50px"><strong>Priority</strong></th>
                        <th width="90px"><strong>Posted By</strong></th>
                        <th><strong>Actions</strong></th>
                    </tr>
                </thead>
                <tbody>
                    {% for o in open_tasks %}
                    <tr>
                        <td width="200px">{{ o.name }}</td>
                        <td width="75px">{{ o.due_date }}</td>
                        <td width="100px">{{ o.posted_date }}</td>
                        <td width="50px">{{ o.priority }}</td>
```

```
<td width="90px">{{ o.poster.name }}</td>
<td>
    <a href="{{ url_for('delete_entry', task_id =
o.task_id) }}>Delete</a> -
    <a href="{{ url_for('complete', task_id =
o.task_id) }}>Mark as Complete</a>
</td>
</tr>
{%
  endfor
%}
</table>
</div>
<br/>
<br/>
<div class="entries">
<h2>Closed tasks:</h2>
<div class="datagrid">
<table>
    <thead>
        <tr>
            <th width="200px"><strong>Task Name</strong></th>
            <th width="75px"><strong>Due Date</strong></th>
            <th width="100px"><strong>Posted Date</strong></th>
            <th width="50px"><strong>Priority</strong></th>
            <th width="90px"><strong>Posted By</strong></th>
            <th><strong>Actions</strong></th>
        </tr>
    </thead>
    {%
      for c in closed_tasks
    %}
    <tr>
        <td width="200px">{{ c.name }}</td>
        <td width="75px">{{ c.due_date }}</td>
        <td width="100px">{{ c.posted_date }}</td>
        <td width="50px">{{ c.priority }}</td>
        <td width="90px">{{ c.poster.name }}</td>
        <td>
            <a href="{{ url_for('delete_entry', task_id =
c.task_id) }}>Delete</a>
        </td>
    </tr>
```

```
{% endfor %}

</table>
</div>

{% endblock %}
```

The changes are fairly straightforward. Take a look at this file along with *forms.py* to see how the drop-down list is implemented. And now you are ready to test! Fire up your server and try adding a few tasks. Register a new user and add some more tasks. We can see that the first user is always showing up under *Posted by*.

Open tasks:

| Task Name | Due Date | Posted Date | Priority | Posted By | Actions |
|----------------------|------------|-------------|----------|-----------|---|
| Can I still add? | 2013-02-04 | 2013-03-26 | 1 | michael | Delete - Mark as Complete |
| different user | 2013-02-04 | 2013-02-02 | 1 | michael | Delete - Mark as Complete |
| Finish this tutorial | 2013-03-13 | 2013-03-26 | 10 | michael | Delete - Mark as Complete |
| Finish my book | 2013-03-13 | 2013-03-26 | 10 | michael | Delete - Mark as Complete |
| Finish this tutorial | 2013-03-13 | 2013-03-26 | 10 | michael | Delete - Mark as Complete |
| Finish my book | 2013-03-13 | 2013-03-26 | 10 | michael | Delete - Mark as Complete |

Let's correct that.

6.8 Managing Sessions

Do you remember the relationship we established between the two tables in the last lesson?

```
user_id = Column(Integer, ForeignKey('users.id'))
tasks = relationship('FTASKS', backref = 'poster')
```

Well, with that information, we can query for the actual name of the user for each task posted. First, we need to log the `user_id` in the session when a user successfully logs in. So make the update to the `login()` function in *views.py*:

```
@app.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method=='POST':
        u = User.query.filter_by(name=request.form['name'],
                                 password=request.form['password']).first()
        if u is None:
            error = 'Invalid username or password.'
        else:
            session['logged_in'] = True
            session['user_id'] = u.id
            flash('You are logged in. Go Crazy.')
            return redirect(url_for('tasks'))

    return render_template("login.html",
                           form = LoginForm(request.form),
                           error = error)
```

Next, when we post a new task, we need to grab that user id and add it to the SQLAlchemy ORM query:

```
@app.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    form = AddTask(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_task = FTasks(
            form.name.data,
            form.due_date.data,
            form.priority.data,
            form.posted_date.data,
            '1',
            session['user_id']
        )
        db.session.add(new_task)
        db.session.commit()
        flash('New entry was successfully posted. Thanks.')
        return redirect(url_for('tasks'))
```

We're grabbing the current user in session, pulling the user id and adding it to the query.

Another pop() method needs to be used for when a user logs out:

```
@app.route('/logout/')
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    flash('You are logged out. Bye. :(')
    return redirect(url_for('login'))
```

Now open up *tasks.html*. In each of the two for loops, note these statements:

```
<td width="90px">{{ o.poster.name }}</td>

<td width="90px">{{ c.poster.name }}</td>
```

Go back to your model real quick, and notice that because we used poster as the backref, we can use it like a regular query object.

Fire up your server. Register a new user and then login using that newly created user. Create new tasks and see how the "Posted By" field gets populated with the name of the user who created the task.

With that, we're done looking at database relationships and the conversion to SQLAlchemy. Again, we can now easily switch SQL database engines. The code now abstracts away much of the repetition from straight SQL so our code is cleaner and more readable.

Next, let's look at form validation.

6.9) Error Handling

Error handling is a means of dealing with errors should they occur at runtime - e.g., when the application is executing a task.

Let's look at an error.

Form Validation Errors

Try to register a new user without entering any information. Nothing should happen. Obviously this will be very confusing for end users. Thus, we need to add in an error message. Fortunately WTForms provides error messages for any form that has a validator attached to it.

Go ahead and check out *forms.py*, there are already some validators in place, it's pretty straightforward. For example, in the `RegisterForm` class, the `name` field should be between 6 and 25 characters:

```
class RegisterForm(Form):
    name = TextField('Username', validators=[Required(),
Length(min=6, max=25)])
    email = TextField('Email', validators=[Required(),
Length(min=6, max=40)])
    password = PasswordField('Password',
                           validators=[Required(),
Length(min=6, max=40)])
    confirm = PasswordField(
        'Repeat Password',
        [Required(), EqualTo('password',
message='Passwords must match')])
```

What we need to do is to display these error messages on our template. [Flashing](#) them is a good solution. To do so, simply add the following code to the *views.py* file:

```
def flash_errors(form):
    for field, errors in form.errors.items():
        for error in errors:
            flash(u"Error in the %s field - %s" % (
                getattr(form, field).label.text, error), 'error')
```

Then add an else statement to each view that has a form:

```
if form.validate_on_submit():
    ###Do Something###
else:
    flash_errors(form)
```

Again, try to register a new user without entering any information. You should now see the error messages. Now, log into the application. Try creating a task without entering any information. You should see error messages here as well.

Database Related Errors

Test to see what happens when you try to register someone with the same username and/or password. You should get an `IntegrityError`. We need to use the `try/except` pair to handle the error, as follows:

1. First add another import to your Application Module, `views.py`:

```
from sqlalchemy.exc import IntegrityError
```

2. Then update the `register()` function:

```
@app.route('/register/', methods=['GET', 'POST'])
def register():
    error = None
    form = RegisterForm(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_user = User(
            form.name.data,
            form.email.data,
            form.password.data,
        )
        try:
            db.session.add(new_user)
            db.session.commit()
            flash('Thanks for registering. Please login.')
            return redirect(url_for('login'))
        except IntegrityError:
            error = 'Oh no! That username and/or email already
exist. Please try again.'
        else:
            flash_errors(form)
            return render_template('register.html', form=form,
error=error)
```

Essentially, the `try` block code attempts to execute. If the program encounters the error specified in the `except` block, the code execution is stopped and the code

within the except block is ran. If the error does not occur then the program fully executes and the except block is skipped altogether.

Test again to see what happens when you try to register someone with the same username and/or email address.

You will *never* be able to anticipate every error, which is why you need to implement error handlers to catch common errors to handle them gracefully so that your application looks professional and to prevent any security vulnerabilities.

Error Debug Mode

The first thing you absolutely must do when preparing to make your app available to the public (deploying to production) is turn off debug mode. Debug mode simply provides a handy debugger for when errors occur, which is great during development, but you never want users to see this. It's also a security vulnerability, as it is possible to execute code through the debugger. You'll find the parameter in the *run.py* file:

```
app.run(debug=True)
```

Just change debug to `False` to disable it.

Custom Error Pages

Now go back and comment out - e.g., add comments to the code so the Python interpreter skips the code altogether - the try/except pairs within the `register()` function that you just set up, and then register a user with a duplicate name. You should see an Internal Server Error, which is also known as a 500 error. Another common error is the annoying 404 Page Not Found. Again, no matter how much you try to prevent such errors, they will still occur from time to time. Fortunately, Flask makes it easy to customize error handlers to handle these more gracefully. You set these up as functions like any other view:

```
@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500

@app.errorhandler(404)
def internal_error(error):
    return render_template('404.html'), 404
```

Then just set up a couple of templates:

Save the following as *404.html*:

```
{% extends "template.html" %}
{% block content %}
    <h1>Sorry ...</h1>
    <p>There's nothing here!</p>
    <p><a href="{{url_for('login')}}">Back</a></p>
{% endblock %}
```

Save the following as *500.html*:

```
{% extends "template.html" %}
{% block content %}
    <h1>Something's wrong!</h1>
    <p>Fortunately we are on the job, and you can just return
to the login page!</p>
    <p><a href="{{url_for('login')}}">Back</a></p>
{% endblock %}
```

Now try to add a duplicate entry. You should be redirected to the *500.html* template. Easy, right? Set them up for other common errors like 403 and 410 as well. Don't forget to uncomment the try/except pair from the register() function.

Logging Module

Finally, it's vital that you set up a means of capturing all errors so that you can spot trends, setup handling, and, of course, fix them.

It's very easy to setup logging at the server level or within the Flask application itself. Many of the third party libraries also log errors.

Here's a basic logger that uses the logging library [3](#):

```
if not app.debug:
    import os
    import logging

    from logging import Formatter, FileHandler
    from config import basedir

    file_handler =
        FileHandler(os.path.join(basedir, 'error.log'))
    file_handler.setFormatter(Formatter('%(asctime)s
%(levelname)s: %(message)s
'[in %(pathname)s:%(lineno)d']))
    app.logger.setLevel(logging.INFO)
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)
    app.logger.info('errors')
```

Here we set the filename that we want errors to be logged in, `error.log` and choose the message format (time, error level, error message) as well as the logging level. There are five levels shown in increasing orders of severity:

1. DEBUG
2. INFO
3. WARNING
4. ERROR

5. CRITICAL

By setting the level to `INFO`, the majority of errors will be logged. If you set up an email logger, you would probably want to set it at a higher severity so only the most important errors are sent to your email - otherwise, you may stop viewing them altogether.

In `__init__.py`, try adding the above logging code just before `from flasktaskr import views, models` and then set `debug=False` in `run.py`. Now try generating an error - i.e., navigating to a page that doesn't exist, and you'll see an `error.log` file in the main project directory.

Alright, in the next lesson, we'll attempt to prevent some errors from happening altogether as we look at unit testing.

6.10) Unit Testing

Conducting unit tests on your application is an absolute necessity, especially as your app grows in size. Such tests help ensure that your application does not break or regress when you make changes or additions to the main code base. Once you've defined the desired expectations of your application, you simply write tests to ensure that the main functions work as expected. The tests are performed regularly and before any changes are made to the code base. In large teams, each working on separate functions, it's good practice to run all unit tests before making changes to ensure that it works and after to ensure that it *still* works.

In many cases, defining the unit tests is done before the application is developed. This helps you fully hash out your application's requirements. It also prevents over-coding: You develop your application until the unit test passes, adding no more code than necessary. Keep in mind though that it's difficult, if not impossible, to establish all your test cases beforehand. You also do not want to limit your application in anyway. It's common practice to write a few unit tests beforehand just to help with building out the skeleton and to establish the main functionalities of the application.

Although there is an official Flask extension called Flask-Testing to perform tests, there is a serious lack of documentation (as of writing). So, I recommend using the pre-installed unit test package that comes with Python, aptly named unittest.⁴

Each test is written as a separate function within a larger class. You can break classes into several test suites. For example, one suite could test the managing of users and sessions, while another tests user registration, and so forth. Such test suites are meant to affirm whether the desired outcome varies from the actual outcome.

All tests begin with this basic framework:

```
import unittest

class TestCase(unittest.TestCase):

    # place your test functions here

    if __name__ == '__main__':
        unittest.main()
```

Let's create a unit test script:

```
# flasktaskr_test.py

import os
import unittest

from app import app, db
from app.models import User
from config import basedir

TEST_DB = 'test.db'

class AddUser(unittest.TestCase):

    # this is a special method that is executed prior to each
    test
    def setUp(self):
        app.config['TESTING'] = True
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////' + \
            os.path.join(basedir, TEST_DB)
        self.app = app.test_client()
        db.create_all()

    # this is a special method that is executed after each test
    def tearDown(self):
        db.drop_all()

    # each test should start with 'test'
    def test_user_setup(self):
        new_user =
User("mherman", "michael@mherman.org", "michaelherman")
        db.session.add(new_user)
        db.session.commit()

if __name__ == "__main__":
    unittest.main()
```

Save it as *flasktaskr_test.py* in the root directory and run it.

Output:

```
Ran 1 test in 0.546s

OK
Exit code: False
```

Let's try to see what happened as we ran this script:

1. The `setUp` method() was invoked which created a test database (if it doesn't exist yet) and initialized the database schema from the main database (e.g., creates the tables, relationships).
2. The `test_user_setup()` method was called, inserting data to the "users" table.
3. Lastly, the `tearDown()` method was invoked which dropped all the tables in the test database.

Try commenting out the `tearDown()` method and run the test script once again.

Check the database in the SQLite Browser. Is the data there?

Now, while the `tearDown()` method is still commented out, run the test script the second time.

Output:

```
IntegrityError: (IntegrityError) column email is not unique
u'INSERT INTO users (name, email, password) VALUES (?, ?, ?)'
('mherman', 'michael@mherman.org', 'michaelherman')

Ran 1 test in 0.047s

FAILED (errors=1)
Exit code: True
```

As you can see, we got an error this time because the name and email must be unique (as defined in our User class in *models.py*).

Assert

Each test should have an assert() method to either verify an expected result or a condition, or that an exception is raised.

Let's quickly look at an example of how assert works:

```
# flasktaskr_test2.py

import os
import unittest

from app import app, db
from app.models import User
from config import basedir

TEST_DB = 'test.db'

class AddUser(unittest.TestCase):

    # this is a special method that is executed prior to each
    test
    def setUp(self):
        app.config['TESTING'] = True
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////' + \
            os.path.join(basedir, TEST_DB)
        self.app = app.test_client()
        db.create_all()

    # this is a special method that is executed after to each
    test
    def tearDown(self):
        db.drop_all()

    # each test should start with 'test'
    def test_user_setup(self):
        new_user =
User("mherman", "michael@mherman.org", "michaelherman")
        db.session.add(new_user)
        db.session.commit()
        test = db.session.query(User).all()
        for t in test:
            t.name
        assert t.name == "mherman"
```

```
if __name__ == "__main__":
    unittest.main()
```

Save it as *flasktaskr_test2.py* in the main directory and run it.

In this example, we're testing the same thing: whether a new user is successfully added to the database. We then pull all data, extract just the name, and then test to make sure the name equals the expected result, which is "mherman".

Run this program in the current form, then change the assert statement to `assert t.name != "mherman"` so you can see what an assertion error looks like:

```
=====
FAIL: test_user_setup (__main__.AddUser)
-----
Traceback (most recent call last):
  File "flasktaskr_test.py", line 35, in test_user_setup
    assert t.name != "mherman"
AssertionError

-----
Ran 1 test in 0.641s

FAILED (failures=1)
Exit code: True
```

You can run multiple tests at once using the Python library [nose](#).

To install:

```
$ pip install nose==1.3.0
```

To run you can either explicitly state the names of the test files from the command line:

```
$ nosetests flasktaskr_test.py flasktaskr_test2.py
```

Or you can place all test files in a single directory and then call the directory name from the command line:

```
$ nosetests flasktaskr_tests
```

6.11) Styling

Now that we're done with creating the basic app, let's change up the styles a bit. Thus far, we've been using the CSS file from the main Flask tutorial. Let's make this app ours. We will be using Twitter Bootstrap to get the basics setup and then I'll briefly show you how to edit the main CSS file to make style changes.

First off, Twitter bootstrap is a front-end framework that makes your app look pretty good darn right out of the box.⁵ You can just use the generic styles; however, it's best to make some changes so that the layout doesn't look like a cookie-cutter template. The framework is great. You'll get the essential tools (e.g., CSS, HTML, and Javascript) needed to build a nice-looking website at your disposal. As long as you have a basic understanding of HTML and CSS, you can create a design quickly.

You can either [download](#) all the associated files and place them in your project directory:

```
└ static
  └ css
    └ bootstrap-responsive.css
    └ bootstrap-responsive.min.css
    └ bootstrap.css
    └ bootstrap.min.css
  └ img
    └ glyphicons-halflings-white.png
    └ glyphicons-halflings.png
  └ js
    └ bootstrap.js
    └ bootstrap.min.js
  styles.css
```

Or you can just link directly to the styles in your *template.html* file:

```
<link href="http://twitter.github.com/bootstrap/assets/css/  
bootstrap.css" rel="stylesheet">
```

It's perfectly fine to use the latter method during development just to get a sense of what your basic application will look like. It's also a means of just getting something up quickly. Keep in mind though that you will need to download the actual files and add them to your project structure to make any changes.

Go ahead and download the files and use this new code for your base template, *template.html*:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Welcome to FlaskTaskr!!!</title>
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <meta name="description" content="">
        <meta name="keywords" content="">
        <meta name="author" content="">
        <meta charset="utf-8">
        <link href="/static/css/bootstrap.css" rel="stylesheet">
        <style> body { padding-top: 65px; } </style>
    </head>
    <body>
        <div class="navbar navbar-fixed-top">
            <div class="navbar-inner">
                <div class="container">
                    <a class="btn btn-navbar" data-toggle="collapse"
data-target=".nav-collapse">
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </a>
                    <a class="brand" href="/">FlaskTaskr</a>
                    <div class="nav-collapse">
                        <ul class="nav">
                            </ul>
                        <ul class="nav pull-right">
                            {% if not session.logged_in %}
                                <li><a href="/register">Signup</a></li>
                            {% else %}
                                <li><a href="/logout">Logout</a></li>
                            {% endif %}
                        </ul>
                    </div>
                </div>
            </div>
        </div>
        <div class="new">
```

```
<div class="content">
    {% for message in get_flashed_messages() %}
        <div class=flash>{{ message }}</div>
        <br/>
    {% endfor %}
    {% if error %}
        <p class=error><strong>Error:</strong> {{ error }}</p>
    {% endif %}
    {% block content %}
    {% endblock %}
</div>
</div>
<footer class="footer">
    <hr>
    <p>&copy; FlaskTaskr</p>
</footer>
</body>
</html>
```

I won't go into too many details, but essentially we just pulled in the bootstrap stylesheets, added a navigation bar to the top, and used the bootstrap classes to style the app. Let's edit some of those styles now. Be sure to check out the bootstrap [documentation](#) for more information.

Before we start, take a look at your app. See the difference. Now, let's make some changes! We'll simplify this a bit and work with just the main bootstrap file, *bootstrap.css*.

First, let's update the color of the `h1`, `h2`, `a` tags, the font-size and font-weight of the `a` tag, and add the same border beneath the `h1` tag as before:

```
h1 {  
    font-size: 38.5px;  
    color: #8D1855;  
    border-bottom: 2px solid #eee;  
}  
  
h2 {  
    font-size: 31.5px;  
    color: #8D1855;  
}  
  
a {  
    color: #8D1855;  
    text-decoration: none;  
    font-size: 16px;  
    font-weight: bold;  
}
```

Next, let's update the `content` class. Add the following CSS to `bootstrap.css`:

```
.new {  
    margin: 2em auto;  
    width: 60em;  
    border: 2px solid #d4d4d4;  
    padding: 0.8em;  
    background: #fafafa;  
}
```

Update the flash and error message styles as well as the footer by adding the following css to `bootstrap.css`:

```
.flash, .error {  
    background: #DEEBF8;  
    padding: 0.5em;  
}  
  
footer {  
    margin: 2em auto;  
    width: 60em;  
    padding: 0.8em;  
}
```

Now, let's update the templates:

1. *login.html*:

```
{% extends "template.html" %}  
{% block content %}  
<h1>Welcome to FlaskTaskr.</h1>  
<h3>Please login to access your task list.</h3>  
<form method="post" action="">  
    {{ form.name.label }}{{ form.name }}  
    {{ form.password.label }}{{ form.password }}  
    <p><input type="submit" value="Submit"  
    class="btn-primary"></p>  
</form>  
<p><em>Need an account? </em><a  
    href="/register">Signup! !</a></p>  
{% endblock %}
```

2. *register.html*:

```
{% extends "template.html" %}

{% block content %}
    <h1>Welcome to FlaskTaskR</h1>
    <br/>
    <h3>Add a new task:</h3>
    <form method="post" action="{{ url_for('new_task') }}">
        <table>
            <tr>
                <td>{{ form.name.label }}{{ form.name
} }&nbsp;</td>
                <td>{{ form.due_date.label }}{{ form.due_
date }}</td>
            </tr>
            <tr>
                <td>{{ form.posted_date.label }}{{ form.posted_
date }}&nbsp;</td>
                <td>{{ form.priority.label }}{{ form.priority }}<td>
            <tr>
        </table>
        <p><input type="submit" value="Submit"
class="btn-primary"></p>
    </form>
    <hr>
    <br/>
    <h2>Open tasks:</h2>
    <div class="datagrid">
        <table>
            <thead>
                <tr>
                    <th width="200px"><strong>Task Name</strong></th>
                    <th width="80px"><strong>Due Date</strong></th>
                    <th width="90px"><strong>Posted Date</strong></th>
                    <th width="50px"><strong>Priority</strong></th>
                    <th width="90px"><strong>Posted By</strong></th>
                    <th width="50px"><strong>Delete</strong></th>
                    <th width="150px"><strong>Complete?</strong></th>
                </tr>
            <tbody>
```

```
</thread>
{%
  for o in open_tasks %
}
<tr>
<td width="200px">{{ o.name }}</td>
<td width="80px">{{ o.due_date }}</td>
<td width="90px">{{ o.posted_date }}</td>
<td width="50px">{{ o.priority }}</td>
<td width="90px">{{ o.poster.name }}</td>
<td width="50px"><a href="{{ url_for('delete_entry', task_id = o.task_id) }}>Delete</a></td>
<td width="150px"><a href="{{ url_for('complete', task_id = o.task_id) }}>Mark as Complete</a></td>
</tr>
{%
  endfor %
}
</table>
</div>
<br/>
<br/>
<h2>Closed tasks:</h2>
<div class="datagrid">
<table>
<thead>
<tr>
<th width="200px"><strong>Task Name</strong></th>
<th width="80px"><strong>Due Date</strong></th>
<th width="90px"><strong>Posted Date</strong></th>
<th width="50px"><strong>Priority</strong></th>
<th width="90px"><strong>Posted By</strong></th>
<th><strong>Actions</strong></th>
</tr>
</thead>
{%
  for c in closed_tasks %
}
<tr>
<td width="200px">{{ c.name }}</td>
<td width="80px">{{ c.due_date }}</td>
<td width="90px">{{ c.posted_date }}</td>
<td width="50px">{{ c.priority }}</td>
<td width="90px">{{ c.poster.name }}</td>
```

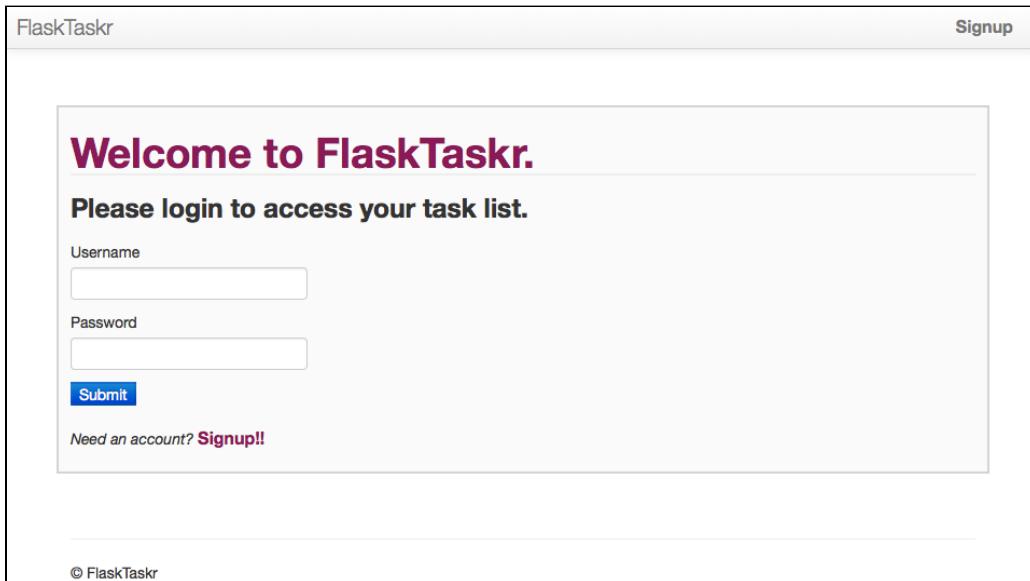
```
<td>
    <a href="{{ url_for('delete_entry', task_id =
c.task_id) }}>Delete</a>
</td>
</tr>
{%
endfor %}
</table>
</div>
{%
endblock %}
```

Add the data grid css to to *bootstrap.css*:

```
.datagrid table {  
    border-collapse: collapse;  
    text-align: left; width: 100%;  
}  
.datagrid {  
    background: #fff;  
    overflow: hidden;  
    border: 1px solid #000000;  
    -webkit-border-radius: 3px;  
    -moz-border-radius: 3px;  
    border-radius: 3px;  
}  
.datagrid table td, .datagrid table th {  
    padding: 3px 10px;  
}  
.datagrid table thead th {  
    background:-webkit-gradient( linear, left top, left bottom,  
color-stop(0.05, #000000), color-stop(1, #000000) );  
    background:-moz-linear-gradient( center top, #000000 5%,  
#000000 100% );  
  
filter:progid:DXImageTransform.Microsoft.gradient(startColorstr='#000000',  
endColorstr='#000000');  
    background-color:#000000;  
    color:#FFFFFF;  
    font-size: 15px;  
    font-weight: bold;  
}  
.datagrid table thead th:first-child {  
    border: none;  
}  
.datagrid table tbody td {  
    color: #000000;  
    border-left: 1px solid #E1EEF4;  
    font-size: 12px;  
    font-weight: normal;  
}  
.datagrid table tbody .alt td {  
    background: #E1EEF4;
```

```
    color: #000000;
}
.datagrid table tbody td:first-child {
    border-left: none;
}
.datagrid table tbody tr:last-child td {
    border-bottom: none;
}
```

Obviously, we can keep making little changes like this until we have something unique.



See what you can do on your own.

6.12) Blueprints

Flask has a built-in feature called Blueprints that allows us to break our application into components.⁶ This is nice feature especially for larger applications because it significantly increases code maintainability and reusability. It also allows us to apply custom templates to each components, among other cool things.

Before beginning, please note that this is not a complicated lesson but there are a number of layers to it so it can be confusing. That said, programming in general is nothing more than stacking various layers of knowledge on top of one another, which again is why it makes learning much easier to start at a low-level and build up from there rather than skipping layers. Sound familiar?

With regard to this lesson: Take it slow. Go through it once without changing any code. Just read and deduce what's happening. Take notes. Draw diagrams. Etc.

Let's add Blueprints to our FlaskTaskr application. We'll break it down to two components:

1. Users

- this will handle user login, logout, and registration
- we will use the twitter bootstrap to style this component

2. Tasks

- this will handle operations for viewing, deleting, and marking tasks as complete
- we will use the old styles.css to style this component

Are you ready? Let's start by identifying the existing directory structure and files for our application:

```
└── app
    ├── __init__.py
    ├── forms.py
    ├── models.py
    └── static
        ├── css
        │   ├── bootstrap-responsive.css
        │   ├── bootstrap-responsive.min.css
        │   ├── bootstrap.css
        │   └── bootstrap.min.css
        ├── img
        │   ├── glyphicons-halflings-white.png
        │   └── glyphicons-halflings.png
        ├── js
        │   ├── bootstrap.js
        │   └── bootstrap.min.js
        └── styles.css
    ├── templates
    │   ├── 404.html
    │   ├── 500.html
    │   ├── login.html
    │   ├── register.html
    │   ├── tasks.html
    │   └── template.html
    └── views.py
├── config.py
├── db_create.py
├── db_migrate.py
├── error.log
├── flasktask.db
├── flasktaskr.db
├── flasktaskr_test.py
├── flasktaskr_test2.py
└── run.py
└── test.db
```

After breaking our app up using Blueprints, our application should have the following structure:

```
└── app
    ├── __init__.py
    ├── static
    │   ├── css
    │   │   ├── bootstrap-responsive.css
    │   │   ├── bootstrap-responsive.min.css
    │   │   ├── bootstrap.css
    │   │   └── bootstrap.min.css
    │   ├── img
    │   │   ├── glyphicons-halflings-white.png
    │   │   └── glyphicons-halflings.png
    │   ├── js
    │   │   ├── bootstrap.js
    │   │   └── bootstrap.min.js
    │   └── styles.css
    ├── tasks
    │   ├── __init__.py
    │   ├── forms.py
    │   ├── static
    │   │   └── styles.css
    │   ├── templates
    │   │   └── tasks
    │   │       ├── tasks.html
    │   │       └── template.html
    │   └── views.py
    ├── templates
    │   ├── 404.html
    │   ├── 500.html
    │   └── template.html
    └── users
        ├── __init__.py
        ├── forms.py
        ├── static
        │   └── bootstrap.css
        ├── templates
        │   └── users
        │       ├── login.html
        │       ├── register.html
        │       └── template.html
```

```
|   |   └── views.py
|   └── views.py
└── config.py
└── db_create.py
└── db_migrate.py
└── error.log
└── flasktask.db
└── flasktaskr.db
└── flasktaskr_test.py
└── flasktaskr_test2.py
└── run.py
└── test.db
```

Study this new structure. You'll notice that both Users and Tasks have their own views, forms, templates, and static files. On the other hand, the *models.py* file remains at the main application directory. In our case, the users and tasks table are bounded by a database relationship so it makes more sense to define them on a single *models.py*.

Let's walk through each step to convert our application:

1. Create the new folders.
2. Create two empty `__init__.py` files, and place them in the "users" and "tasks" directories.

If you remember, `__init__.py` are indicators to python that a folder contains python modules (*.py files) that we will import.

3. Copy the bootstrap files to the "users" directory
4. Copy our old stylesheet, `styles.css`, to the "tasks" directory.

By using different css files, we will see later that the tasks page uses the old layout, while the login and registration pages uses the newer twitter bootstrap layout.

Users Component

1. Views

```
# /app/users/views.py

from app import db
from flask import Blueprint, flash, redirect,
render_template, request, session, url_for
from app.views import login_required, flash_errors
from forms import RegisterForm, LoginForm
from app.models import User
from sqlalchemy.exc import IntegrityError

mod = Blueprint('users', __name__, url_prefix='/users',
                 template_folder='templates',
                 static_folder='static')

@mod.route('/logout/')
def logout():
    session.pop('logged_in', None)
    session.pop('user_id', None)
    flash('You are logged out. Bye. :(')
    return redirect(url_for('.login'))

@mod.route('/', methods=['GET', 'POST'])
def login():
    error = None
    if request.method=='POST':
        u = User.query.filter_by(name=request.form['name'],
password=request.form['password']).first()
        if u is None:
            error = 'Invalid username or password.'
        else:
            session['logged_in'] = True
            session['user_id'] = u.id
            flash('You are logged in. Go Crazy.')
            return redirect(url_for('tasks.tasks'))

    return render_template("users/login.html",
                           form = LoginForm(request.form),
```

```
        error = error)

@mod.route('/register/', methods=['GET', 'POST'])
def register():
    error = None
    form = RegisterForm(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_user = User(
            form.name.data,
            form.email.data,
            form.password.data,
        )
        try:
            db.session.add(new_user)
            db.session.commit()
            flash('Thanks for registering. Please login.')
            return redirect(url_for('.login'))
        except IntegrityError:
            error = 'Oh no! That username and/or email
already exist. Please try again.'
        else:
            flash_errors(form)
    return render_template('users/register.html',
form=form, error=error)
```

What's going on here? First, we defined our Users Blueprint with custom templates and static folders:

```
mod = Blueprint('users', __name__, url_prefix='/users',
template_folder='templates', static_folder='static')
```

We also assigned it to `url_prefix='/users'`, which means all views for this component will start at `http://127.0.0.1:5000/users`

We then use the `mod` variable as if it's the Flask application itself. You can see that when we defined the routes for the views, e.g.

```
@mod.route('/logout/')
```

The rest of the code is the same, except for the parameters we passed to the `url_for` and `render_template` functions.

For the `url_for`, we used two forms:

- `url_for('tasks.tasks')` is used to construct the url for the tasks view of the tasks blueprint.
- `url_for('.login')` is used to construct the url for the login view of the *current* blueprint, which is the users blueprint.

For the `render_template`, we had to specify the path relative to the current blueprint's template folder: `render_template('users/register.html')` points to this template: `/app/users/templates/register.html`

2. Forms

```
# /app/users/forms.py

from flask.ext.wtf import Form, TextField, PasswordField,
Required, EqualTo, Length

class RegisterForm(Form):
    name = TextField('Username', validators=[Required(),
Length(min=6, max=25)])
    email = TextField('Email', validators=[Required(),
Length(min=6, max=40)])
    password = PasswordField('Password',
                           validators=[Required(),
Length(min=6, max=40)])
    confirm = PasswordField(
        'Repeat Password',
        [Required(), EqualTo('password',
message='Passwords must match')])

class LoginForm(Form):
    name = TextField('Username', validators=[Required()])
    password = PasswordField('Password',
                           validators=[Required()])
```

3. /app/users/templates/template.html:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Welcome to FlaskTaskr!!!</title>
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <meta name="description" content="">
        <meta name="keywords" content="">
        <meta name="author" content="">
        <meta charset="utf-8">
        <link href="{{ url_for('users.static',
filename='bootstrap.css') }}" rel="stylesheet">
        <style> body { padding-top: 65px; } </style>
    </head>
<body>
    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="btn btn-navbar"
data-toggle="collapse" data-target=".nav-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </a>
                <a class="brand" href="/">FlaskTaskr</a>
                <div class="nav-collapse">
                    <ul class="nav">
                    </ul>
                    <ul class="nav pull-right">
                        {% if not session.logged_in %}
                            <li><a href="{{
url_for('users.register') }}">Signup</a></li>
                        {% else %}
                            <li><a href="{{ url_for('users.logout') }}">Logout</a></li>
                        {% endif %}
                    </ul>
                </div>
            </div>
        </div>
    </div>
```

```
        </div>
    </div>
<div class="new">
    <div class="content">
        {%
            for message in get_flashed_messages() %
        %}
            <div class=flash>{{ message }}</div>
            <br/>
        {% endfor %}
        {% if error %}
            <p class=error><strong>Error:</strong> {{ error
} }
        {% endif %}
        {% block content %}
        {% endblock %}
    </div>
</div>
<footer class="footer">
    <hr>
    <p>&copy; FlaskTaskr</p>
</footer>
</body>
</html>
```

There is also nothing special here, except that we made sure that the `url_for()` function points to the correct css file.

4. /app/users/templates/login.html:

```
{% extends "users/template.html" %}  
{% block content %}  
    <h1>Welcome to FlaskTaskr.</h1>  
    <h3>Please login to access your task list.</h3>  
    <form method="post" action="">  
        {{ form.name.label }}{{ form.name }}  
        {{ form.password.label }}{{ form.password }}  
        <p><input type="submit" value="Submit"  
class="btn-primary"></p>  
    </form>  
    <p><em>Need an account? </em><a href="{{  
url_for('users.register') }}>Signup!!</a></p>  
{% endblock %}
```

Here, we need to make sure that the `extends()` function receives the correct template path, similar to the `render_template()` function. And don't forget the `url_for()` function.

5. `/app/users/templates/register.html:`

```
{% extends "users/template.html" %}  
{% block content %}  
    <h1>Welcome to FlaskTaskr.</h1>  
    <h3>Please register to access the task list.</h3>  
    <form method="post" action="">  
        <table>  
            <tr>  
                <td>{{ form.name.label }}{{ form.name  
} }&nbsp;&nbsp;</td>  
                <td>{{ form.email.label }}{{ form.email }}</td>  
            </tr>  
            <tr>  
                <td>{{ form.password.label }}{{ form.password  
} }&nbsp;&nbsp;</td>  
                <td>{{ form.confirm.label }}{{ form.confirm }}</td>  
            </tr>  
        </table>  
        <p><input type="submit" value="Register"  
class="btn-primary"></p>  
    </form>  
    <p><em>Already registered?</em> Click <a href="{{  
url_for('users.login') }}>here</a> to login.</p>  
    {% endblock %}
```

Can you identify the modifications in the file, *register.html*? Also, for the Tasks Components the process is the same. See if you can mimic what we did for the User Components before looking at the scripts below.

Tasks Component

1. Views

```
# /app/tasks/views.py

from app import db
from flask import Blueprint, flash, redirect,
render_template, request, session, url_for
from app.views import login_required, flash_errors
from forms import AddTask
from app.models import FTasks

mod = Blueprint('tasks', __name__, url_prefix='/tasks',
                 template_folder='templates',
                 static_folder='static')

@mod.route('/tasks/')
@login_required
def tasks():
    open_tasks =
        db.session.query(FTasks).filter_by(status='1').order_by(
            FTasks.due_date.asc())
    closed_tasks =
        db.session.query(FTasks).filter_by(status='0').order_by(
            FTasks.due_date.asc())
    return render_template('tasks/tasks.html', form =
AddTask(request.form),
                           open_tasks=open_tasks,
                           closed_tasks=closed_tasks)

@mod.route('/add/', methods=['GET', 'POST'])
@login_required
def new_task():
    form = AddTask(request.form, csrf_enabled=False)
    if form.validate_on_submit():
        new_task = FTasks(
            form.name.data,
            form.due_date.data,
            form.priority.data,
            form.posted_date.data,
            '1',
```

```
        session['user_id']
    )
    db.session.add(new_task)
    db.session.commit()
    flash('New entry was successfully posted. Thanks.')
else:
    flash_errors(form)
return redirect(url_for('.tasks'))


@mod.route('/complete/<int:task_id>',)
@login_required
def complete(task_id):
    new_id = task_id

    db.session.query(FTasks).filter_by(task_id=new_id).update({"status":"0"})
    db.session.commit()
    flash('The task was marked as complete. Nice.')
    return redirect(url_for('.tasks'))


@mod.route('/delete/<int:task_id>',)
@login_required
def delete_entry(task_id):
    new_id = task_id

    db.session.query(FTasks).filter_by(task_id=new_id).delete()
    db.session.commit()
    flash('The task was deleted. Why not add a new one?')
    return redirect(url_for('.tasks'))
```

2. Forms

```
# /app/tasks/forms.py

from flask.ext.wtf import Form, TextField, DateField,
IntegerField, \
    SelectField, Required

class AddTask(Form):
    task_id = IntegerField('Priority')
    name = TextField('Task Name', validators=[Required()])
    due_date = DateField('Date Due (mm/dd/yyyy)', validators=[Required()],
                         format='%m/%d/%Y')
    priority = SelectField('Priority', validators=[Required()],
                           choices=[('1', '1'), ('2',
                                     '2'), ('3', '3'),
                                     ('4', '4'), ('5',
                                     '5')])
    status = IntegerField('Status')
    posted_date = DateField('Posted Date (mm/dd/yyyy)', validators=[Required()],
                           format='%m/%d/%Y')
```

3. /app/tasks/templates/template.html:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to FlaskTaskr!!</title>
        <link rel="stylesheet" href="{{ url_for('tasks.static', filename='styles.css') }}">
    </head>
    <body>
        <div class="page">
            {% for message in get_flashed_messages() %}
                <div class="flash">{{ message }}</div>
                <br/>
            {% endfor %}
            {% if error %}
                <div class="error"><strong>Error:</strong> {{ error }}</div>
            {% endif %}
            {% block content %}
            {% endblock %}
        </div>
    </body>
</html>
```

4. /app/tasks/templates/tasks.html

```
{% extends "tasks/template.html" %}

{% block content %}
    <h1>Welcome to FlaskTaskR</h1>
    <br/>
    <a href="{{ url_for('users.logout') }}>Logout</a>
    <div class="add-task">
        <h3>Add a new task:</h3>
        <form method="POST" action="{{ url_for('tasks.new_task') }}>
            <p>{{ form.name.label }}: {{ form.name }}<br />
            {{ form.due_date.label }}:
                {{ form.due_date }}&ampnbsp{{ form.posted_date.label }}:
                {{ form.posted_date }}&ampnbsp{{ form.priority.label }}:
                {{ form.priority }}</p>
            <p><input type="submit" value="Submit"></p>
        </form>
    </div>
    <div class="entries">
        <br/>
        <br/>
        <h2>Open tasks:</h2>
        <div class="datagrid">
            <table>
                <thead>
                    <tr>
                        <th width="200px"><strong>Task Name</strong></th>
                        <th width="75px"><strong>Due Date</strong></th>
                        <th width="100px"><strong>Posted Date</strong></th>
                        <th width="50px"><strong>Priority</strong></th>
                        <th width="90px"><strong>Posted By</strong></th>
                        <th><strong>Actions</strong></th>
                    </tr>
                </thead>
                <tbody>
                    {% for o in open_tasks %}
                    <tr>
                        <td width="200px">{{ o.name }}</td>
                        <td width="75px">{{ o.due_date }}</td>
                        <td width="100px">{{ o.posted_date }}</td>
                        <td width="50px">{{ o.priority }}</td>
```

```
<td width="90px">{{ o.poster.name }}</td>
<td>
    <a href="{{ url_for('tasks.delete_entry',
task_id = o.task_id) }}>Delete</a> -
        <a href="{{ url_for('tasks.complete', task_id =
o.task_id) }}>Mark as Complete</a>
    </td>
</tr>
{% endfor %}
</table>
</div>
<br/>
<br/>
<div class="entries">
<h2>Closed tasks:</h2>
<div class="datagrid">
<table>
<thead>
<tr>
<th width="200px"><strong>Task Name</strong></th>
<th width="75px"><strong>Due Date</strong></th>
<th width="100px"><strong>Posted Date</strong></th>
<th width="50px"><strong>Priority</strong></th>
<th width="90px"><strong>Posted By</strong></th>
<th><strong>Actions</strong></th>
</tr>
</thead>
<tbody>
{% for c in closed_tasks %}
<tr>
<td width="200px">{{ c.name }}</td>
<td width="75px">{{ c.due_date }}</td>
<td width="100px">{{ c.posted_date }}</td>
<td width="50px">{{ c.priority }}</td>
<td width="90px">{{ c.poster.name }}</td>
<td>
        <a href="{{ url_for('tasks.delete_entry',
task_id = c.task_id) }}>Delete</a>
    </td>
</tr>

```

```
{% endfor %}  
</table>  
</div>  
{% endblock %}
```

Main App

1. /app/**init**.py:

```
# __init__.py

from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object('config')
db = SQLAlchemy(app)

if not app.debug:
    import os
    import logging

    from logging import Formatter, FileHandler
    from config import basedir

    file_handler =
        FileHandler(os.path.join(basedir, 'error.log'))
    file_handler.setFormatter(Formatter('%(asctime)s
%(levelname)s: %(message)s
'[in %(pathname)s:%(lineno)d']))
    app.logger.setLevel(logging.INFO)
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)
    app.logger.info('errors')

from app import views, models
from app.users.views import mod as usersMod
from app.tasks.views import mod as tasksMod

app.register_blueprint(usersMod)
app.register_blueprint(tasksMod)
```

This is how to setup a blueprint:

- Import the mod from the component: `from app.users.views import mod as usersMod`

- Register it to the application: `app.register_blueprint(usersMod)`

This needs to be done for each component that we have in our application. Fortunately, we only have two.

2. /app/views.py:

```
# views.py

from app import app, db
from flask import flash, redirect, render_template,
session, url_for
from functools import wraps

def flash_errors(form):
    for field, errors in form.errors.items():
        for error in errors:
            flash(u"Error in the %s field - %s" % (
                getattr(form, field).label.text, error),
            'error')

def login_required(test):
    @wraps(test)
    def wrap(*args, **kwargs):
        if 'logged_in' in session:
            return test(*args, **kwargs)
        else:
            flash('You need to login first.')
            return redirect(url_for('users.login'))
    return wrap

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500

@app.errorhandler(404)
def internal_error(error):
    return render_template('404.html'), 404

@app.route('/', defaults={'page': 'index'})
def index(page):
    return redirect(url_for('tasks.tasks'))
```

This code is essentially what's left after we moved most of our views to the Users and Tasks components. Take note of the last view:

```
@app.route('/', defaults={'page': 'index'})  
def index(page):  
    return(redirect(url_for('tasks.tasks')))
```

This view redirects the homepage to the tasks view of the Tasks blueprint.

3. /app/templates/404.html:

```
{% extends "template.html" %}  
{% block content %}  
    <h1>Sorry ...</h1>  
    <p>There's nothing here!</p>  
    <p><a href="{{url_for('users.login')}}">Back</a></p>  
{% endblock %}
```

4. /app/templates/500.html:

```
{% extends "template.html" %}  
{% block content %}  
    <h1>Something's wrong!</h1>  
    <p>Fortunately we are on the job, and you can just  
    return to the login page!</p>  
    <p><a href="{{url_for('users.login')}}">Back</a></p>  
{% endblock %}
```

Now, fire up your server. Everything should be the same as before, except for the layout of the tasks page. All done! Again, by breaking our application up logically by functional area using blueprints our code is cleaner and more readable. As your app grows, it will be much easier to develop the additional functionalities due to the separation of concerns.

6.13) Deploying on Heroku

As far as deployment options go, PythonAnywhere and Heroku are great options. We'll use PythonAnywhere throughout the web2py sections, so let me show you how to use Heroku with this app.⁷

Deploying an app to Heroku is ridiculously easy:

1. Signup for [Heroku](#)
2. Login to Heroku and download the [Heroku Toolbelt](#) applicable to your operating system
3. Once installed, open your command-line and run the following command:

```
$ heroku login
```

Then follow the prompts:

```
Enter your Heroku credentials.
Email: michael@mherman.org
Password (typing will be hidden):
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/michaelherman/.ssh/
id_rsa.pub
```

4. Activate your virtualenv
5. Heroku recognizes the dependencies needed through a *requirements.txt* file. Create one using the following command: `pip freeze > requirements.txt`. Now, this will only create the dependencies from the libraries you installed using Pip. If you used `easy_install`, you will need to add them directly to the file.
6. Create a [Procfile](#). Open up a text editor and save the following text in it:

```
web: python run.py
```

Then save the file in your application's root or main directory as *Procfile* (no extension). The word "web" indicates to Heroku that the application will be attached to HTTP once deployed (sent to the production server).

7. Update your *run.py* file. On your local machine, the application runs on port 5000 by default. On Heroku, the application **must** run on a random port number specified by Heroku. We will identify this port number by reading the environment variable 'PORT' and passing it to `app.run`:

```
# run.py

import os

from flasktaskr import app

port = int(os.environ.get('PORT', 5000))
app.run(host='0.0.0.0', port=port, debug=False)
```

8. Create a local Git repository:

```
$ git init
$ git add .
$ git commit -m "initial files"
```

9. Create your app on Heroku:

```
$ heroku create
```

10. Deploy your code to Heroku:

```
$ git push heroku master
$ heroku ps:scale web=1
```

11. Check to make sure your app is running:

```
$ heroku ps
```

12. View the app in your browser:

```
$ heroku open
```

13. If you see errors, open the Heroku log to view all errors and output:

```
$ heroku logs
```

That's it. You'll also want to push your local repository to Github. I'll show you how to do that in the next section.

You can see my app at <http://flasktaskr.herokuapp.com>

6.14) Boilerplate Template and Workflow

Say goodbye to FlaskTaskr. :(

You should now understand many of the underlying basics of creating an app in Flask. I want to leave you with a pre-configured Flask boilerplate [template](#) that has many of the bells and whistles we looked at already installed (batteries included) so that you can get started creating an app right away. I'll also detail a workflow for you to use throughout the development process to help you stay organized and ensure that your Flask instance will scale right along with you. In essence, I'll show you how to get an app up and running on Heroku as fast as possible, and then describe a simple workflow for you to adhere to as you build your app.

Setup your virtualenv

```
$ mkdir flask
$ cd flask
```

Unix:

```
$ virtualenv --no-site-packages env  
$ source env/bin/activate
```

Windows:

```
$ virtualenv --no-site-packages env  
$ env\scripts\activate
```

Download the boilerplate template from Github

This downloads the files needed for creating this generic application. Make sure you are in the "flask" directory before cloning.

```
$ git clone git://github.com/mjhea0/flask-boilerplate.git  
$ cd flask-boilerplate
```

Project structure:

```
├── Procfile
├── Procfile.dev
├── README.md
├── app.py
├── config.py
├── error.log
├── forms.py
├── models.py
├── requirements.txt
├── static
│   ├── css
│   │   ├── bootstrap-responsive.css
│   │   └── bootstrap.css
│   ├── img
│   │   ├── glyphicon-halflings-white.png
│   │   └── glyphicon-halflings.png
│   └── js
│       ├── libs
│       │   ├── bootstrap.min.js
│       │   ├── jquery.min.js
│       │   └── modernizr-2.0.6.min.js
│       └── plugins.js
└── templates
    ├── 404.html
    ├── 500.html
    ├── index.html
    ├── login.html
    ├── register.html
    └── template.html
```

Install the various libraries and dependencies:

```
$ pip install -r requirements.txt
```

You can also view the dependencies by running the command `pip freeze`:

```
Flask==0.10.1
Flask-SQLAlchemy==1.0
Flask-WTF==0.8.4
Jinja2==2.7
MarkupSafe==0.18
SQLAlchemy==0.8.2
WTForms==1.0.4
Werkzeug==0.9.3
itsdangerous==0.22
wsgiref==0.1.2
```

Deploy to Heroku

Follow the steps in the last lesson.

Now that you have your skeleton app up, it's time to start developing locally. Here's a great workflow to follow. Before we start, make sure to signup for a [Github](#) account.

Github is a web-based version control system and web hosting service. We will be using Github for version control, which is used to track changes in code, store a backup of your project's files, and allow for project collaboration.

It's important to PUSH to Github often during development in case you need to revert back to earlier versions of a code base, because of some change you made that caused the application to break. If you need more help, please refer to my [repository](#) for a basic workflow, designed for a single user.

Once you have a Github account, create a new repository. If you need help, follow along Github's [documentation](#).

Create a local repository and push initial files to Github:

```
$ git init
$ git add .
$ git commit -m 'first commit'
$ git remote add origin https://github.com/YOUR_GITHUB_USERNAME/
MY_APP.git
$ git push -u origin master
```

Now that everything is setup, let's look at the workflow you can follow as you create and modify your application:

1. Edit your application locally
2. Run and test locally
3. Push to Github
4. Push to Heroku
5. Test live application
6. Rinse and repeat

That's it. You now have a skeleton app to work with to build your own applications. Cheers!

If you prefer PythonAnywhere over Heroku, simply deploy your app there during the setup phase and then change step #4 in your workflow to deploy to PythonAnywhere instead of Heroku. Simple. You can also look at the documentation [here](#).

And with that, we are down with Flask. Let's move on to a high-level web framework: **web2py**.

-
1. <http://pythonhosted.org/Flask-SQLAlchemy/>
 2. <http://pythonhosted.org/Flask-WTF/>
 3. <http://docs.python.org/2/howto/logging.html>

4. <http://docs.python.org/2/library/unittest.html>
5. <http://twitter.github.com/bootstrap/index.html>
6. <http://flask.pocoo.org/docs/blueprints/>
7. <https://devcenter.heroku.com/articles/python>

Interlude: Web Frameworks, Compared

Overview

As previously mentioned, web frameworks alleviate the overhead incurred from common, repetitive tasks associated with web development. By using a web framework, web developers delegate responsibility of low-level tasks to the framework itself, allowing the developer to focus on the application logic.

These low-level tasks include handling of:

- Client requests and subsequent server responses,
- URL routing,
- Separation of concerns (application logic vs. HTML output), and
- Database communication.

Take note of the concept "Don't Repeat Yourself" (or DRY). Always avoid reinventing the wheel. This is exactly what web frameworks are great at. The majority of the above low-level tasks are part of every web application. Since frameworks automate much of these tasks, you can get up and running quickly, so you can focus your development time on what really matters: making your application stand out from the crowd.

Most frameworks also include a development web server, which is a great tool used not only for rapid development but testing as well.

The majority of web frameworks can be classified as either full (high-level), or micro, depending on the amount and level of automation it can perform, and its number of pre-installed components (batteries). Full frameworks come with many pre-installed batteries and a lot of low-level task automation, while micro frameworks come with few batteries and less automation. All web frameworks do

offer some automation, however, to help speed up web development. In the end, it's up to the developer to decide how much control he or she wants. Beginning developers should first focus on demystifying much of the *magic*, (what developers commonly refer to automation as), to help understand the differences between the various web frameworks and avoid later confusion.

Popular Frameworks

1. **web2py**, **Django**, and **Turbogears** are all full frameworks, which offer a number of pre-installed utilities and automate many tasks in the backend. They all have excellent documentation and community support. The high-level of automation, though, can make the learning curve for these quite steep.
2. **web.py**, **CherryPy**, and **bottle.py**, are micro-frameworks with few batteries included, and automate only a few underlying tasks. Each has excellent community support and is easy to install and work with. **web.py**'s documentation is a bit unorganized, but still relatively well-documented like the other frameworks.
3. Both **Pyramid** and **Flask**, which are still considered micro-frameworks, have quite a few pre-installed batteries and a number of additional pre-configured batteries available as well, which are very easy to install. Again, documentation and community support are excellent, and both are very easy to use.

Components

Before starting development with a new framework, learn what pre-installed and available batteries it offers. At the core, most of the components work in a similar manner; however, there are subtle differences. Take Flask and web2py, for example; Flask uses an ORM for database communication and a type of template engine called Jinja2. web2py, on the other hand, uses a DAL for communicating with a database and has its own brand of templates.

Don't just jump right in, in other words, thinking that once you learn to develop in one, you can use the same techniques to develop in another. Take the time to learn the differences between frameworks to avoid later confusion.

What does this all mean?

As stated in the Introduction of this course, the framework(s) you decide to use should depend more on which you are comfortable with and your end-product goals. If you try various frameworks and learn to recognize their similarities while also respecting their differences, you will find a framework that suits your tastes as well as your application.

Don't let other developers dictate what you do or try. Find out for yourself!

7) web2py: QuickStart

7.1) Overview

web2py is a high-level, open source web framework designed for rapid development. With web2py, you can accomplish everything from installation, to project setup, to actual development, quickly and easily. In no time flat, you'll be up and running and ready to build beautiful, dynamic websites.



In this section, we'll be exploring the fundamentals of web2py - from installation to the basic development process. You'll see how web2py automates much of the low-level, routine tasks of development, resulting in a smoother process, and one which allows you to focus on high-level issues. web2py also comes pre-installed with many of the components we had to manually install for Flask.

web2py, developed in 2007 by Massimo Di Pierro (an associate professor of Computer Science at DePaul University [1](#)), takes a different approach to web development than the other Python frameworks. Because Di Pierro **aimed** to lower the barrier for entry into web development, so more automation (often called *magic*) happens under the hood, which can make development simpler, but it can also give you less control over how your app is developed. All frameworks share a number of basic common traits. If you learn these as well as the web development fundamentals from Section One, you will better understand what's happening behind the scenes, and thus know how to make changes to the automated processes for better customization

Homework

- Watch [this](#) excellent speech by Di Pierro from PyCon US 2012. Don't worry if all the concepts don't make sense right now. They will soon enough. Pause the video at times and look up any concepts that you don't understand. Take notes. Google-it-first.

7.2) Installation

Quick Install

If you want to get started quickly, you can [download](#) the binary archive, unzip, and run either web2py.exe (Windows) or web2py.app (Unix). You must set an administrator password to access the administrative interface. The Python Interpreter is included in the archive, as well as many third party libraries and packages. You will then have a development environment set up, and be ready to start building your application - all in less than a minute, and without even having to touch the terminal.² This is the quick and dirty method of installation, however. If you intend on using a virtualenv, which is the common practice, you will need to download the source code instead.

Full Install

1. Navigate to the "realpython" directory from your terminal. Then run the following command to create the virtualenv:

```
$ virtualenv web2py --no-site-packages
```

2. Enter the new "web2py" directory, and activate the virtualenv:

Unix

```
$ source bin/activate
```

Windows

```
$ scripts\activate
```

3. Create a directory called "start". Download the source code from the [web2py website](#) and place it into this new directory. Unzip the file.
4. Back on your command line, launch web2py:

```
$ python web2py.py
```

5. After web2py loads, set an admin password, and you're good to go. Once you've finished your current session with web2py, exit the virtualenv:

```
$ deactivate
```

6. To run web2py again, activate virtualenv and launch web2py.

web2py by default separates projects (a collection of related applications); however, it's still important to use a virtualenv to keep your third-party libraries isolated from one another. That said, we'll be using the same web2py instance and directory, "start", for the apps in this chapter.

Regardless of how you install web2py (Quick vs Full), a number of libraries are pre-imported. These provide a functional base to work with so you can start programming dynamic websites as soon as web2py is installed.

7.3) Hello World

1. Navigate to the "web2py" directory. Activate your virtualenv. Enter the "start" directory. Fire up the server.

```
$ cd desktop/realpython/web2py  
$ source bin/activate  
$ cd start  
$ python web2py.py
```

2. Input your admin password. Once logged in, click the button for the "Administrative Interface" on the right side of the page. Enter your password again. You're now on the **Sites** page. This is the main administration page where you create and modify your applications.

The screenshot shows the web2py administrative interface. On the left, under 'Installed applications', there are three entries: 'admin (currently running)' (with 'Manage' and 'Disable' buttons), 'examples' (with 'Manage' and 'Disable' buttons), and 'welcome' (with 'Manage' and 'Disable' buttons). On the right, there's a sidebar with 'Version' information (2.4.5 - stable+timestamp.2013.03.18.22.46.22, Running on Rocket 1.2.6) and a message saying 'web2py is up to date'. Below that is a section for 'New simple application' with a text input field for 'Application name:' and a 'Create' button. At the bottom of the sidebar is a link 'Upload and install packed application'. A small orange triangle at the bottom left contains the text 'Questions?' and 'join our google group'.

3. To create a new application, type the name of the app, "hello_world", in the text field below "New simple application". Click create to be taken to the **Edit** page. All new applications are just copies of the "welcome" app:

The screenshot shows the 'Hello World' application within the web2py framework. The main page displays 'Welcome' and 'customize me!'. Below it is a 'Hello World' heading and a 'How did you get here?' section with a numbered list of steps. To the right is a sidebar with a 'Administrative Interface' button, a 'Don't know what to do?' section with links to 'Online examples', 'web2py.com', and 'Documentation', and a 'Share' button at the bottom.

You'll soon find out that web2py has a default for pretty much everything (but which can be modified). In this case, if you don't make any changes to the

views, for example, your app will have the basic styles and layout taken from the default, "welcome" app.

Think about some of the pros and cons to having a default for everything. You obviously can get an application set up quickly. Perhaps if you aren't adept at a particular part of development, you could just rely on the defaults. However, if you do go that route you probably won't learn anything new - and you could have a difficult transition to another framework that doesn't rely on defaults. Thus, I urge you to practice. Break things. Learn the default behaviors. Make them better. Make them your own.

4. The **Edit** page is logically organized around the Model-View-Controller (MVC) design workflow:

- **Models** represent the data.
- **Views** visually represent the data model.
- **Controllers** route user requests and subsequent server responses

We'll go over the MVC architecture with regard to web2py in more detail in later chapters. For now, just know that it's a means of splitting the back-end business logic from the front-end views, and it's used to simplify the development process by allowing you to develop your application in phases or chunks.

5. Next we need to modify the default controller, `default.py`, so click the "edit" button next to the name of the file.
6. Now we're in the **web2py IDE** (the Shell). Replace the `index()` function with the following code:

```
def index():
    return dict(message="Hello from web2py!")
```

7. Save the file, then hit the Back button to return to the **Edit** page.

- Now let's update the view. Edit the `default/index.html` file, replacing the existing code with:

```
<html>
  <head>
    <title>Hello App!</title>
  </head>
  <body>
    <br/>
    <h1>{{=message}}</h1>
  </body>
</html>
```

- Save the file, return to the **Edit** page again. Click the "index" link next to the `default.py` file which loads the main page. You should see the greeting, "Hello from web2py!" staring back at you.

Easy right? So what happened?

The controller returned a dictionary with the key/value pair `message="Hello from web2py"`. Then, in the view, we defined how we wanted the greeting to be displayed by the browser. In other words, the functions in the controller return dictionaries, which are then converted into views surrounded by `{{ }}` tags. The values from the dictionary are used as variables. In the beginning, you won't need to worry about the views since web2py has so many pre-made views already built in (again: defaults). So, you can focus solely on back-end development.

When a dictionary is returned, web2py looks to associate the dictionary with a view that matches the following format: `[controller_name]/[function_name].[extension]`. If no extension is specified, it defaults to `.html`, and if web2py cannot find the view, it defaults to using the `generic.html` view:

The screenshot shows a list of files under the 'Views' section. The files listed are:

- __init__.py
- appadmin.html extends layout.html
- default/index.html
- default/user.html extends layout.html
- generic.html extends layout.html
- generic.ics
- generic.json
- generic.jsonp
- generic.load

A red arrow points to the 'generic.ics' file entry.

For example, if we created this structure:

- Controller = run.py
- Function = hello()
- Extension .html

Then the url would be: http://your_site/run/hello.html

In the hello_world app, since we used the `default.py` controller with the `index()` function, web2py looked to associate this view with `default/index.html`.

You can also easily render the view in different formats, like JSON or XML, by simply updating the extension:

- Go to http://localhost:8000/hello_world/default/index.html
- Change the extension to render the data in a different format:
 - XML: http://localhost:8000/hello_world/default/index.xml
 - JSON: http://localhost:8000/hello_world/default/index.json
 - RSS: http://localhost:8000/hello_world/default/index.rss

Try this out. When done, hit CRTL-C from your terminal to stop the server.

7.4) Deploying on PythonAnywhere

Of all the various cloud-based hosting solutions, [PythonAnywhere](#) provides the simplest means of deploying a web2py application. As its name suggests, PythonAnywhere is a Python-hosting platform that allows you to develop within your browser from anywhere in the world where there's an Internet connection.³



Simply invite a friend or colleague to join your session, and you have the perfect collaborative environment for pair programming projects - or for getting help with a certain script you can't get working. It has a lot of other useful [features](#) as well, such as Drop-box integration and Python Shell access, among others.

1. Go ahead and create an account and log in. Once logged in, click "Web" => "Add a new web app", choose `your_username.pythonanywhere.com`, and click the button for web2py. Set an admin password and then click "Next" one last time to set up the web2py project.
2. Navigate to https://your_username.pythonanywhere.com. (Note the https in the url.) Look familiar? It better. Open the Admin Interface just like before and you can now start building your application.

If you wanted, you could develop your entire app on PythonAnywhere. Cool, right?

3. Back on your local version of web2py return to the admin page <http://127.0.0.1:8000/admin/default/site>, click the "Manage" drop down button, the select "Pack All" to save the *w2p-package* to your computer.
4. Once downloaded return to the Admin Interface on PythonAnywhere. To create your app, go to the "Upload and install packed application" section on

the right side of the page, give your app a name ("hello_World"), and finally upload the *w2p-file* you saved to your computer earlier. Click install.

5. Navigate to your app's homepage:

https://your_username.pythonanywhere.com/hello_world/default/index

Congrats. You just deployed your first app!

Homework

- Python Anywhere is considered a Platform as a Service (PaaS). Find out exactly what that means. What's the difference between a PaaS hosting solution vs. shares hosting?
- Learn more about other Python PaaS options: <http://www.slideshare.net/appsembler/pycon-talk-deploy-python-apps-in-5-min-with-a-paas>

7.5) seconds2minutes App

Let's create a new app that converts seconds to minutes. Activate your virtualenv, start the server, set a password, enter the Admin Interface, and create a new app called "seconds2minutes".

We'll be developing this locally. However, feel free to try developing it directly on PythonAnywhere. Better yet: Do both.

In this example, the controller will define two functions. The first function, `index()`, will return a form to `index.html`, which is then displayed for users to enter the number of seconds they want converted over to minutes. Meanwhile, the second function, `convert()`, takes the number of seconds and converts them to the number of minutes. Both the variables are then passed to the `convert.html` view.

1. Replace the code in `default.py` with:

```
def index():
    form=FORM('# of seconds: ',
              INPUT(_name='seconds', requires=IS_NOT_EMPTY()),
              INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('convert', args=form.vars.seconds))
    return dict(form=form)

def convert():
    seconds = request.args(0, cast=int)
    return dict(seconds=seconds, minutes=seconds/60,
                new_seconds=seconds%60)
```

2. Edit the *default/index.html* view, replacing the default code with:

```
<center>
<h1>seconds2minutes</h1>
<h3>Please enter the number of seconds you would like
converted to minutes.</h3>
<p>{{=form}}</p>
</center>
```

3. Create a new view called *default/convert.html*, replacing the default code with:

```
<center>
<h1>seconds2minutes</h1>
<p>{{=seconds}} seconds is {{=minutes}} minutes and
{{=new_seconds}} seconds.</p>
<br/>
<p><a href="/seconds2minutes/default/index">Try
again?</a></p>
</center>
```

Check out the live app. Test it out.

When we created the form, as long as a value is entered, we will be redirected to *convert.html*. Try entering no value, as well as a string or float. Currently, the only

validation we have is that the form doesn't show up blank. Let's alter the code to add additional validators.

Change the form validator to `requires=IS_INT_IN_RANGE(0,1000000)`. The new `index()` function looks like this:

```
def index():
    form=FORM('# of seconds:',
              INPUT(_type='integer', _name='seconds',
                    requires=IS_INT_IN_RANGE(0,1000000)),
              INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('convert',args=form.vars.seconds))
    return dict(form=form)
```

Test it out. You should get an error, unless you enter an integer between 0 and 999,999:

The screenshot shows a web page with the title "seconds2minutes". Below the title, there is a bold instruction: "Please enter the number of seconds you would like converted to minutes.". Underneath this, there is a text input field with the placeholder "# of seconds:" and the value "120.2". To the right of the input field is an error message in a yellow box: "enter an integer between 0 and 999999". On the left side of the input field, there is a button with the text "oops!" enclosed in a black rounded rectangle. A thick red arrow points from this button towards the error message. At the bottom right of the input field is a "Submit Query" button.

Again, the `convert()` function takes the seconds and then runs the basic expressions to convert the seconds to minutes. These variables are then passed to the dictionary and are used in the `convert.html` view.

Homework

- Deploy this app on PythonAnywhere.

7.6) Sentiment Analysis

What is Sentiment Analysis?

Essentially, [sentiment analysis](#) measures the sentiment of something - a feeling rather than a fact, in other words. The aim is to break down natural language data, analyze each word, and then determine if the data as a whole is positive, negative, or neutral.

Twitter is a great resource for sourcing data for sentiment analysis. You could use the Twitter API to pull hundreds of thousands of tweets on topics such as Obama, abortion, gun control, etc. to get sense of how the nation feels about a particular topic. Companies use sentiment analysis to gain a deeper understanding about marketing campaigns, certain product lines, and the company itself.

You'll want to pick a topic that people have strong opinions about.

In this example, we'll be using a [natural language classifier](#) to power a web application that allows you to enter data for analysis via an html form. The focus is not on the classifier but on the development of the application. For more information on how to develop your own classifier using Python, please read this amazing [article](#).

1. Start by reading the API [documentation](#) for the natural language classifier we'll be using for our app. Are the docs clear? What questions do you have? Write them down. If you can't answer them by the end of this lesson, try the "Google-it-first" method, then, if you still have questions, post them to the Real Python [message forum](#).
2. First, what the heck is cURL? For simplicity, cURL is a utility used for transferring data across numerous protocols.⁴ We will be using it to test HTTP requests.

Traditionally, you would access cURL from the terminal in Unix systems. Unfortunately for Windows users, command prompt does not come with

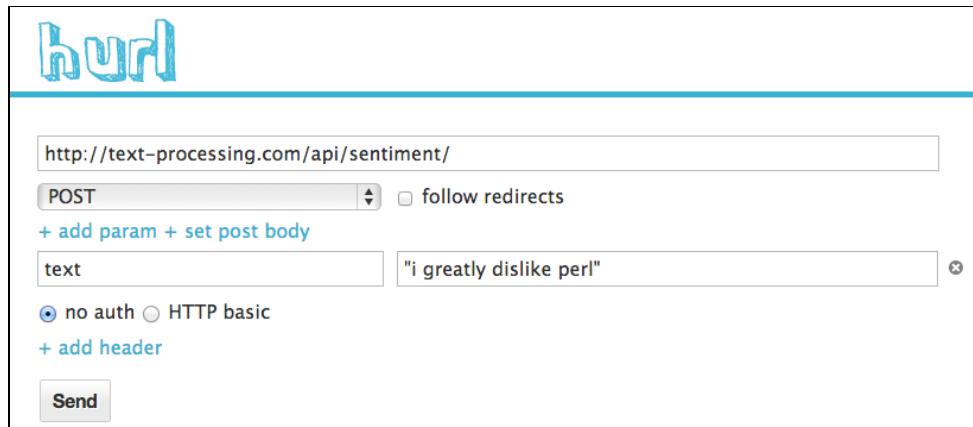
the utility. Fortunately, there is an advanced command line tool called [Cygwin](#) available that provides a Unix-like terminal for Windows. See the Appendix for more information.

Unix users, and Windows users with Cygwin installed, test out the API in the terminal:

```
$ curl -d "text=great" http://text-processing.com/api/  
sentiment/  
{"probability": {"neg": 0.35968353095023886, "neutral":  
0.29896828324578045, "pos": 0.64031646904976114}, "label":  
"pos"}  
  
$ curl -d "text=i hate apples" http://text-processing.com/  
api/sentiment/  
{"probability": {"neg": 0.65605365432549356, "neutral":  
0.3611947857779943, "pos": 0.34394634567450649}, "label":  
"neg"}  
  
$ curl -d "text=i usually like ice cream but this place is  
terrible" http://text-processing.com/api/sentiment/  
{"probability": {"neg": 0.90030914036608489, "neutral":  
0.010418429982506104, "pos": 0.099690859633915108},  
"label": "neg"}  
  
$ curl -d "text=i really really like you, but today you  
just smell." http://text-processing.com/api/sentiment/  
{"probability": {"neg": 0.638029187699517, "neutral":  
0.001701536649255885, "pos": 0.36197081230048306}, "label":  
"neg"}
```

So, you can see the natural language, the probability of the sentiment being positive, negative, or neutral, and then the final sentiment. Did you notice the last two text statements are more neutral than negative but were classified as negative? Why do you think that is? How can a computer analyze sarcasm?

You can also test the API on [Hurl](#):



Steps:

- Enter the URL
- Change the HTTP method to POST
- Add the parameter `text` and `"i greatly dislike perl"`
- Press send

Surprised at the results?

All right. Let's build the app. Before we begin, though, we will be using the `requests` library for initiating the POST request. The cURL command is equivalent to the following code:

```
import requests

url = 'http://text-processing.com/api/sentiment/'
data = {'text': 'great'}
r = requests.post(url, data=data)
print r.content
```

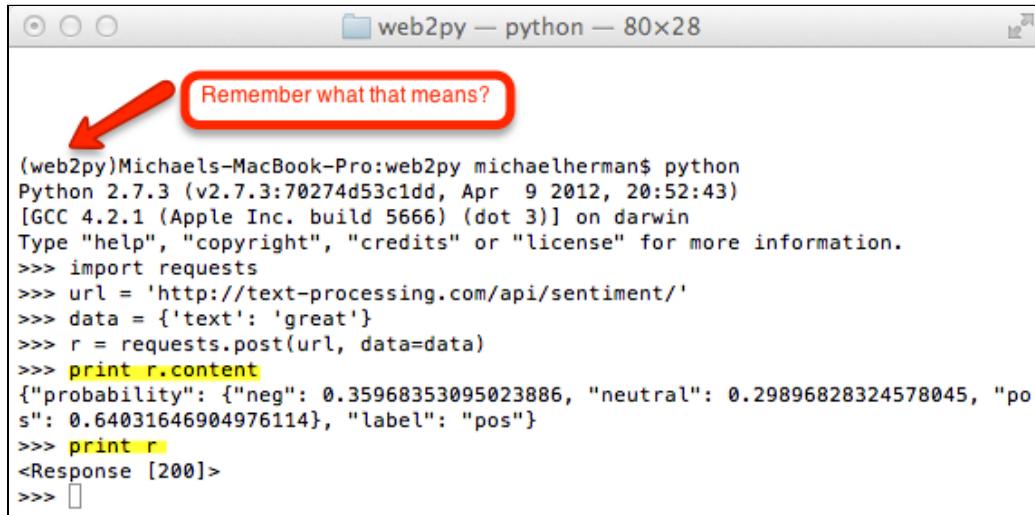
Go ahead and install the `requests` library.

Wait. Didn't we already do that? Remember => Since we're in a different virtualenv, we need to install it again.

Remember the command?

```
pip install requests
```

Now, test out the above code in your Shell:



```
(web2py)Michaels-MacBook-Pro:web2py michaelherman$ python
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> url = 'http://text-processing.com/api/sentiment/'
>>> data = {'text': 'great'}
>>> r = requests.post(url, data=data)
>>> print r.content
{"probability": {"neg": 0.35968353095023886, "neutral": 0.29896828324578045, "pos": 0.64031646904976114}, "label": "pos"}
>>> print r
<Response [200]>
>>> 
```

Now, let's build the app for easily testing sentiment analysis.

Pulse App

1. You know the drill: Activate virtualenv, fire up the server, enter the Admin Interface, and create a new app called "pulse".
2. Like the last app, the controller will define two functions, index() and pulser(). index(), will return a form to *index.html*, so users can enter the text for analysis. pulser(), meanwhile, handles the POST request to the API and outputs the results of the analysis to *pulser.html*.
3. Replace the code in the default controller with:

```
import requests

def index():
    form=FORM(
        TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('pulser',args=form.vars.pulse))
    return dict(form=form)

def pulser():
    text = request.args(0)
    text = text.split('_')
    text = ' '.join(text)
    url = 'http://text-processing.com/api/sentiment/'
    data = {'text': text}
    r = requests.post(url, data=data)
    return dict(text=text, r=r.content)
```

4. Now let's create some basic views.

default/index.html:

```
{{extend 'layout.html'}}
<center>
<br/>
<br/>
<h1>check a pulse</h1>
<h4>Just another Sentiment Analysis tool.</h4>
<br/>
<p>{{=form}}</p>
</center>
```

default/pulser.html:

```
{ {extend 'layout.html'}}  
<center>  
  <p>{{=text}}</p>  
  <br/>  
  <p>{{=r}}</p>  
  <br/>  
  <p><a href="/pulse/default/index">Another Pulse?</a></p>  
</center>
```

5. All right. Test this out. Compare the results to the results using either cURL or Hurl to make sure all is set up correctly.
6. Now, let's finish cleaning up *pulse.html*. We need to parse/decode the JSON file. What do you think the end user wants to see? Do you think they care about the probabilities? Or just the end results? What about a graph? That would be cool. It all depends on your (intended) audience. Let's just parse out the end result.

Update *default.py*:

```
import requests
import json

def index():
    form=FORM(
        TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit')).process()
    if form.accepted:
        redirect(URL('pulser',args=form.vars.pulse))
    return dict(form=form)

def pulser():
    text = request.args(0)
    text = text.split('_')
    text = ' '.join(text)

    url = 'http://text-processing.com/api/sentiment/'
    data = {'text': text}

    r = requests.post(url, data=data)

    binary = r.content
    output = json.loads(binary)
    label = output["label"]

    return dict(text=text, label=label)
```

Update *default/pulser.html*:

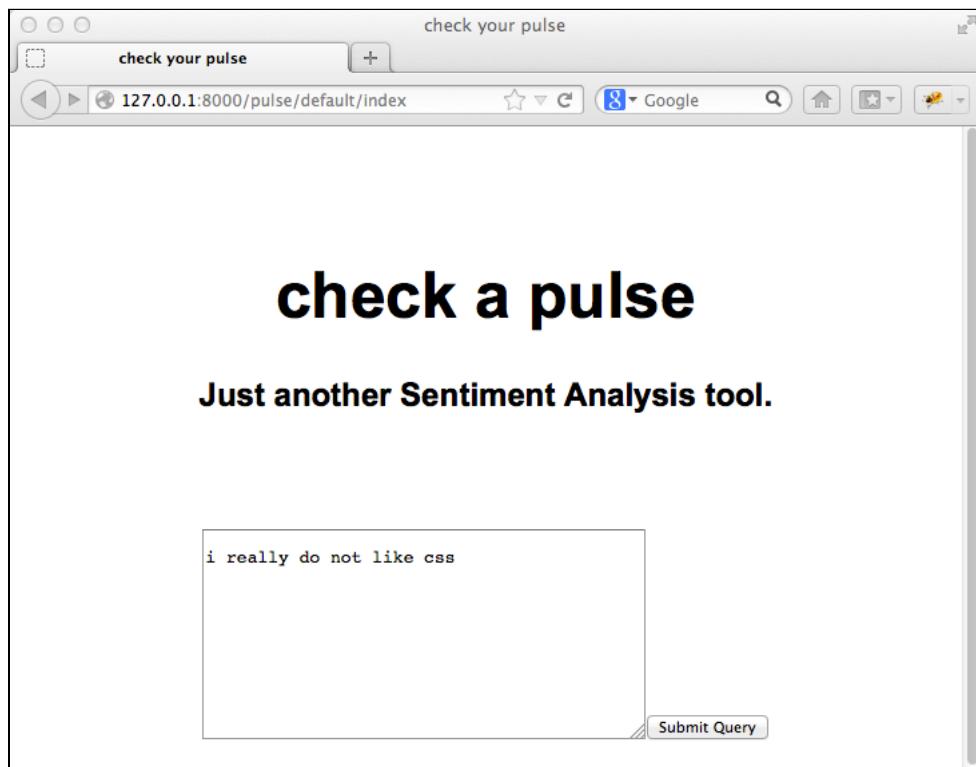
```
{ {extend 'layout.html'}}  
<center>  
<br/>  
<br/>  
<h1>your pulse</h1>  
<h4>{{=text}}</h4>  
<p>is</p>  
<h4>{{=label}}</h4>  
<br/>  
<p><a href="/pulse/default/index">Another Pulse?</a></p>  
</center>
```

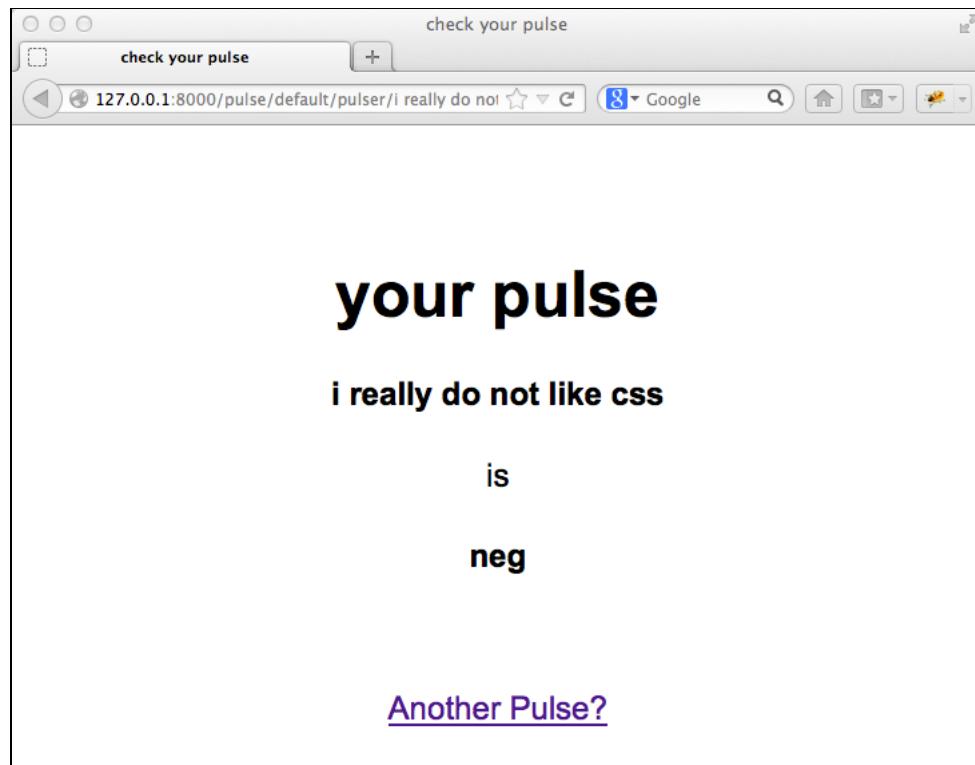
7. Make it pretty.

Update *layout.html*

```
<!DOCTYPE html>  
<html>  
<head>  
<title>check your pulse</title>  
<meta charset="utf-8" />  
<style type="text/css">  
    body {font-family: Arial, Helvetica, sans-serif;  
font-size:x-large;}  
</style>  
<{  
    middle_columns = {0:'span12',1:'span9',2:'span6'}  
}>  
<{{block head}}><{{end}}>  
</head>  
<body>  
<div class="{{=middle_columns}}>  
<{{block center}}>  
<{{include}}>  
<{{end}}>  
</div>  
</body>  
</html>
```

8. This is not pretty; it's functional:





9. Make your's pretty.

What else could you do with this? Well, you could easily tie a database into the application to save the inputted text as well as the results. With sentiment analysis, you want your algorithm to get smarter ([machine learning](#)). Right now, the algorithm is static. Try entering the term "i like milk". It's negative, right?

```
{"probability": {"neg": 0.50114184747628709, "neutral": 0.34259733533730058, "pos": 0.49885815252371291}, "label": "neg"}
```

Why is that?

Test out each word:

"i":

```
{"probability": {"neg": 0.5488526802724282, "neutral": 0.37816113425135217, "pos": 0.45114731972757172}, "label": "neg"}
```

"like":

```
{"probability": {"neg": 0.52484460041100567, "neutral": 0.45831376351784164, "pos": 0.47515539958899439}, "label": "neg"}
```

"milk":

```
{"probability": {"neg": 0.54015839746206784, "neutral": 0.47078672070829519, "pos": 0.45984160253793216}, "label": "neg"}
```

All negative. Doesn't seem right. The algorithm needs to be updated.

Unfortunately, that's beyond the scope of this course. By saving each result in a database, you can begin analyzing the results to at least find errors and spot trends. From there, you can begin to update the algorithm. Good luck.

7.7) Blog App

Let's recreate the blog app we created in the previous section. Pay attention, as this will go quickly. Remember the requirements?

- The user is presented with the basic user login screen
- After logging in, the user can add new posts, read existing posts, or logout
- Sessions need to be used to protect against unauthorized users accessing the main page

Once again, activate your virtualenv, start the server, set a password, and enter the Admin Interface. Finally create a new app called "web2blog". (Please Feel free to come up with something a bit more creative.)

Model

The database has one table, *blog_posts*, with two fields - *title* and *post*. Remember how we used an ORM to interact with the database with the Flask framework?

Well, web2py uses a similar system called a Database Abstraction Layer (DAL).⁵

To keep things simple, an ORM is a subset of a DAL. Both are used to map database functions to Python objects.

Open up *db.py* and append the following code:

```
db.define_table('blog_posts',
    Field('title', notnull=True),
    Field('post', 'text', notnull=True))
```

We'll go over the main differences between an ORM and a DAL in the next chapter. For now, go ahead and save the file and return to the **Edit** page. As long as there are no errors, web2py created an admin interface used to manage the database, located directly below the "Models" header. Click the button to access the admin interface. From here you can add data to the tables within the database. Go ahead and add a few rows of dummy data, then click on the actual table (*db.posts*) to view the records you just added.

As soon as the admin interface is accessed, *db.py* is executed and the tables are created. Return to the **Edit** page. A new button should have populated next to the admin interface button called "sql.log". Click the link to view the actual SQL statements used to create the table. Scroll to the bottom. You should see the following:

```
timestamp: 2013-03-04T19:51:44.464307
CREATE TABLE blog_posts(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title CHAR(512) NOT NULL,
    post TEXT NOT NULL
);
success!
```

But what are all those other tables? We'll get to that. One step at a time.

Controller

Next, replace the `index()` function in `default.py` with:

```
def index():
    form=SQLFORM(db.blog_posts)
    if form.process().accepted:
        response.flash = "Post accepted - cheers!"
    elif form.errors:
        response.flash = "Post not accepted - fix the error(s)."
    else:
        response.flash = "Please fill out the form - thank you!"
    posts = db().select(db.blog_posts.ALL)
    return dict(posts=posts, form=form)
```

This adds an HTML form so that users can add posts. It also queries the database, pulling all rows of data and returning the results as a dictionary. Again, the values of the dictionary are turned into variables.

View

Edit the view `default/index.html` with the following code:

```
{%extend 'layout.html'%}
<h1>Add a Post</h1>
{{=form}}
<h1>Current Posts</h1>
<br/>
<table>
    <tr><td><h3>Title</h3></td><td><h3>Post</h3></td></tr>
    {{for p in posts:}}
    <tr><td>{{=(A(p.title))}}</td><td>{{=(A(p.post))}}</td></tr>
    {{pass}}
</table>
```

Now, view your app at: <http://127.0.0.1:8000/web2blog/default/>

Wrap Up

All right. So, what are we missing?

1. User login/registration
2. Login_required Decorator to protect *index.html*
3. Session Management

Here's where the real power of web2py comes into play. All of those are auto-implemented in web2py. That's right. Are you starting to like defaults? Remember how long this took to implement into the Flask app?

Simply add the login_required decorator to the index() function in the controller, *default.py*:

```
@auth.requires_login()
def index():
    form=SQLFORM(db.blog_posts)
    if form.process().accepted:
        response.flash = "Post accepted - cheers!"
    elif form.errors:
        response.flash = "Post not accepted - fix the error(s)."
    else:
        response.flash = "Please fill out the form - thank you!"
    posts = db().select(db.blog_posts.ALL)
    return dict(posts=posts, form=form)
```

Try to access the site again. You should now have to register and login. That's all there is to it! Session Management is already set up as well. I'll explain how this works in the next chapter.

-
1. <http://www.cdm.depaul.edu/People/Pages/facultyinfo.aspx?fid=343> ↵

2. <http://web2py.com/books/default/chapter/29/03#Startup>
3. https://www.pythonanywhere.com/details/develop_anywhere
4. <http://en.wikipedia.org/wiki/CURL>
5. <http://web2py.com/books/default/chapter/29/06>

8) web2py: py2manager

8.1) Introduction

In the last chapter we built several small applications to illustrate the power of web2py. Those applications were meant more for learning. In this chapter we will develop a much larger application: a task manager, similar to FlaskTaskr, called **py2manager**.

This application will be developed from the ground up to not only show you all that web2py has to offer - but to also dig deeper into modern web development and the Model View Controller development style.

This application will do the following:

1. Users must sign in (and register, if necessary) before hitting the landing page, *index.html*.
2. Once signed in, users can add new companies, projects, and notes associated with projects and view other employees' profiles.
3. Each company consists of a company name, email, phone number, and URL.
4. Each project consists of a name, employee name (person who logged the project), description, start date, due date, and completed field that indicates whether the project has been completed.
5. Finally, notes reference a project and include a text field for the actual note, created date, and a created by field.

Up to this point, you have developed a number of different applications using the Model View Controller (MVC) architecture pattern:

1. Model: data warehouse (database)

2. View: data output (templates, JavaScript, CSS, HTML, etc.)
3. Controller: link between the user and the application

Again, a user sends a request to a web server. The server, in turn, passes that request to the controller. Using the established workflow, the controller then performs an action, such as querying or modifying the database (model). Once the data is found or updated, the controller then passes the results back to the views, which is seen by the user (response).

Most modern web frameworks utilize MVC-style architecture, offering similar components. But each framework implements the various components slightly different, due to the choices made by the developers of the framework. Learning about such differences is vital for sound development.

We'll look at MVC in terms of web2py as we develop py2manager.

8.2) Setup

Before we begin, let's setup a virtualenv and a new app:

1. Navigate to your "realpython/web2py" directory
2. Install a new virtualenv:

```
virtualenv --no-site-packages py2manager
```

3. CD into the new directory, and then activate the virtual environment:

Unix:

```
source bin/activate
```

Windows:

```
scripts\activate
```

4. Download the source code from the web2py [website](#) and place it into the "py2manager" directory. Unzip the file, placing all files and folders into the "py2manager" directory.
5. Create a new app from your terminal:

```
python web2py.py -S py2manager
```

6. Exit the Shell.
7. Navigate to the "Applications" directory, then into the "py2manager" directory. This directory holds all of your application's files.

Sublime Text

Before moving on, we're going to start using a more advanced text editor called Sublime Text to help keep our project organized and speed up development. The internal web2py IDE is great for small applications, but you'll want to use either an advanced text editor or an IDE for larger applications.

You can download Sublime Text 2 [here](#). Once installed, you need to load your development project:

1. "Project" => "Add folder to project"
2. Navigate to your "realpython/py2manager/applications/py2manager" directory (where your application data is held).
3. Select the directory and then click Open.

You should now have the entire application structure (files and folders) in Sublime. Take a look around. Open the "Models", "Views", and "Controllers" folders. Simply double-click to open a particular file to load it in the editor window. Files appear as tabs on the top of the editor window, allowing you to move between files quickly.

```

default.py      db.py
1  # -*- coding: utf-8 -*-
2
3 #####
4  ## This scaffolding model makes your app work on Google App Engine too
5  ## File is released under public domain and you can use without limitations
6 #####
7
8  ## if SSL/HTTPS is properly configured and you want all HTTP requests to
9  ## be redirected to HTTPS, uncomment the line below:
10 # request.requires_https()
11
12 if not request.env.web2py_runtime_gae:
13     ## if NOT running on Google App Engine use SQLite or other DB
14     db = DAL('sqlite://storage.sqlite', pool_size=1, check_reserved=['all'])
15 else:
16     ## connect to Google BigTable (optional 'google:datastore://namespace')
17     db = DAL('google:datastore')
18     ## store sessions and tickets there
19     session.connect(request, response, db=db)
20     ## or store session in Memcache, Redis, etc.
21     ## from gluon.contrib.memdb import MEMDB
22     ## from google.appengine.api.memcache import Client
23     ## session.connect(request, response, db = MEMDB(Client()))
24
25     ## by default give a view/generic.extension to all actions from localhost
26     ## none otherwise. a pattern can be 'controller/function.extension'
27     response.generic_patterns = ['*'] if request.is_local else []
28
29     ## (optional) optimize handling of static files
30     # response.optimize_css = 'concat,minify,inline'
31     # response.optimize_js = 'concat,minify,inline'
32 #####

```

Line 1, Column 1 Spaces: 4 Python

8.3) Version Control

It's common practice to put your application under version control before development. This is an optional step, which we skipped before, but now that you're versed in the development process, and hopefully ready to start building more complex apps, it's vital to use a version control system.

Such systems allow you to easily track changes when code is updated, revert (or rollback) to earlier versions of your codebase in case of an error (like inadvertently deleting a file or a large chunk of code, for example), and collaborate on the project. Take the time to learn how to use a version control system. This could save you much time in the future - when an error occurs, for example, and you need to rollback your code - and is a required skill for web developers to have.

1. Start by downloading [Git](#), if you don't already have it.
2. If you've never installed Git before you need to set your global first name, last name, and email address. Open the terminal in Unix or the Git Bash Shell in

Windows (Start > All Programs > Git > Git Bash), then enter the following commands:

```
git config --global user.name "FIRST_NAME LAST_NAME"  
git config --global user.email "MY_NAME@example.com"
```

3. Sign up for [Github](#) to host your Git repository (which is just a secure location to store your files), and setup a [new](#) repository.
4. Back in your terminal, navigate to your project directory, then run the following commands:

```
git init  
touch README  
git add *  
git commit -m "My initial commit message"  
git remote add origin git@github.com:Your-Username/  
awesomeProject.git  
git push origin master
```

5. This creates the necessary files and pushes them to the remote repository on Github.
6. You'll want to PUSH after each lesson:

```
git add *  
git commit -m 'Text goes here'  
git push origin master
```

The string `text goes here` should be replaced each time with a brief description of the changes made since the last PUSH.

7. If you need to PULL the master copy from Github:

```
git pull origin master
```

If you are collaborating on a project, you'll want to grab the latest master version before making changes to the codebase to the local copy on your machine.

That's it. With regard to Git, it's essential that you know how to:

- Add and commit,
- Pull the master from Github, and
- Push your local copy back to master on Github.

If you want to take it a step further, it's good to learn how to make branches, which are just copies of the master repository where you can make changes that do not affect that master branch, and then merging that branch with the main branch. You can think of them as a sandbox where you can make any change you want without affecting the main codebase. Then if you do decide to keep those changes, you can merge them with the master branch.

Let's begin developing our application.

8.4) Database

As you saw in the previous chapter, web2py uses an API called a Database Abstraction Layer (DAL) to map Python objects to database objects. Like an ORM, a DAL hides the complexity of the underlying SQL code. The major difference between an ORM and a DAL, is that a DAL operates at a lower level.¹ In other words, its syntax is somewhat closer to SQL. If you have experience with SQL, you may find DAL easier to work with than an ORM. If not, learning the syntax is no more difficult than an ORM:

ORM:

```
class User(db.Model):
    __tablename__ = 'users'
    name = db.Column(db.String, unique=True, nullable=False)
    email = db.Column(db.String, unique=True, nullable=False)
    password = db.Column(db.String, nullable=False)
```

DAL:

```
db.define_table('users',
    Field('name', 'string', unique=True, notnull=True),
    Field('email', 'string', unique=True, notnull=True),
    Field('password', 'string', 'password', readable=False,
label='Password'))
```

The above examples create the exact same "users" table. ORMs generally use classes to declare tables, while the web2py DAL uses functions. Both are portable among many different relational database engines (database agnostic). Meaning you can switch your database engine without having to re-write the code within your Model. web2py is integrated with a number of popular databases, including SQLite, PostgreSQL, MySQL, SQL Server, FireBird, Oracle, MongoDB, among others.

Shell

If you prefer the command line, you can work directly from the web2py Shell. The following is a quick, unrelated example:

- In your terminal navigate to the project root directory (realpython/web2py/py2manager), then run the following command:

```
python web2py.py --shell=py2manager
```

- Run the following DAL commands: [2](#)

```
>>> db =
DAL('sqlite://storage.sqlite', pool_size=1, check_reserved=['all'])

>>> db.define_table('special_users', Field('name'),
Field('email'))
<Table special_users (id,name,email)>

>>> db.special_users.insert(id=1, name="Alex",
email="hey@alex.com")
1L

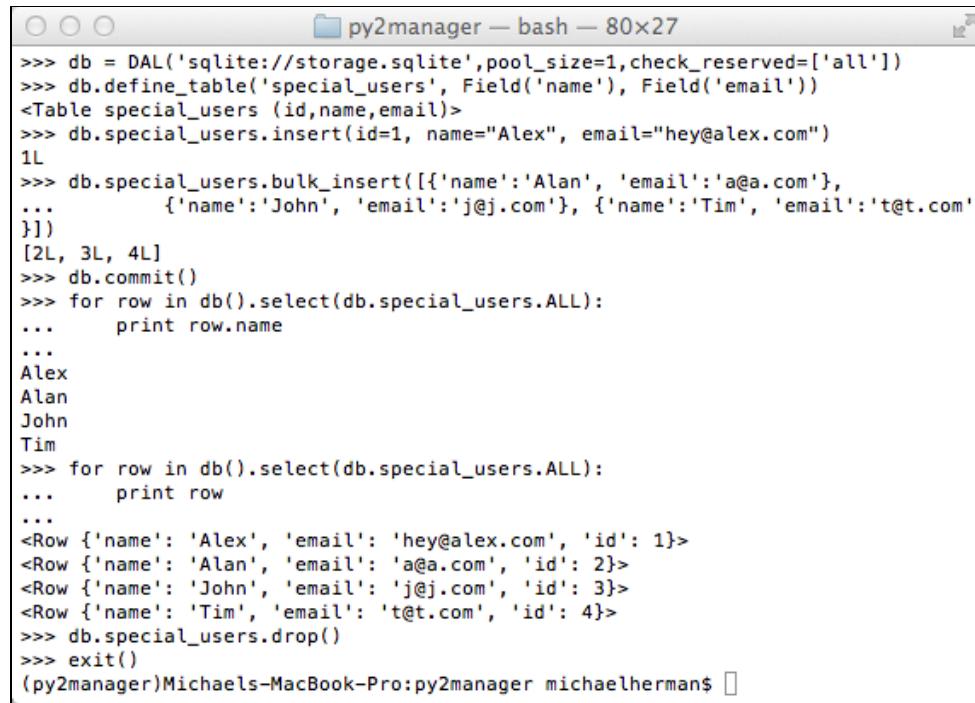
>>> db.special_users.bulk_insert([{'name':'Alan',
'email':'a@a.com'},
 {'name':'John', 'email':'j@j.com'}, {'name':'Tim',
'email':'t@t.com'}])
[2L, 3L, 4L]

>>> db.commit()

>>> for row in db().select(db.special_users.ALL):
...     print row.name
...
Alex
Alan
John
Tim

>>> for row in db().select(db.special_users.ALL):
...     print row
...
<Row {'name': 'Alex', 'email': 'hey@alex.com', 'id': 1}>
<Row {'name': 'Alan', 'email': 'a@a.com', 'id': 2}>
<Row {'name': 'John', 'email': 'j@j.com', 'id': 3}>
<Row {'name': 'Tim', 'email': 't@t.com', 'id': 4}>

>>> db.special_users.drop()
>>> exit()
```



```

>>> db = DAL('sqlite://storage.sqlite', pool_size=1, check_reserved=['all'])
>>> db.define_table('special_users', Field('name'), Field('email'))
<Table special_users (id,name,email)>
>>> db.special_users.insert(id=1, name="Alex", email="hey@alex.com")
1L
>>> db.special_users.bulk_insert([{'name':'Alan', 'email':'a@a.com'},
...     {'name':'John', 'email':'j@j.com'}, {'name':'Tim', 'email':'t@t.com'}
])
[2L, 3L, 4L]
>>> db.commit()
>>> for row in db().select(db.special_users.ALL):
...     print row.name
...
Alex
Alan
John
Tim
>>> for row in db().select(db.special_users.ALL):
...     print row
...
<Row {'name': 'Alex', 'email': 'hey@alex.com', 'id': 1}>
<Row {'name': 'Alan', 'email': 'a@a.com', 'id': 2}>
<Row {'name': 'John', 'email': 'j@j.com', 'id': 3}>
<Row {'name': 'Tim', 'email': 't@t.com', 'id': 4}>
>>> db.special_users.drop()
>>> exit()
(py2manager)Michaels-MacBook-Pro:py2manager michaelherman$ 

```

Here we created a new table called "special_users" with the fields "name" and "email". We then inserted a single row of data then multiple rows. Finally, we printed the data to the screen before dropping the table and exiting the Shell.

web2py Admin

Now, as I mentioned in the last chapter, everything has a default. These are the default values for each table field:

```

Field(name, 'string', length=None, default=None,
      required=False, requires='<default>',
      ondelete='CASCADE', notnull=False, unique=False,
      uploadfield=True, widget=None, label=None, comment=None,
      writable=True, readable=True, update=None, authorize=None,
      autodelete=False, represent=None, compute=None,
      uploadfolder=os.path.join(request.folder, 'uploads'),
      uploadseparate=None, uploadfs=None)

```

So, the field, `Field('company_name')`, would by default be a string value, not required (`notnull=False`), and does not have to be unique (`unique=False`). Keep this in mind when you are creating your database tables.

Let's create the model for our application.

1. Create a new file to define your database schema, in Sublime within Models, called `db_tasks.py`. Then add the following code:

```
db.define_table('company',
    Field('company_name', notnull=True, unique=True),
    Field('email'),
    Field('phone', notnull=True),
    Field('url'),
    format = '%(company_name)s')

db.company.email.requires=IS_EMAIL()
db.company.url.requires=IS_EMPTY_OR(IS_URL())

db.define_table('project',
    Field('name', notnull=True),
    Field('employee_name', db.auth_user,
default=auth.user_id),
    Field('company_name', 'reference company',
notnull=True),
    Field('description', 'text', notnull=True),
    Field('start_date', 'date', notnull=True),
    Field('due_date', 'date', notnull=True),
    Field('completed', 'boolean', notnull=True),
    format = '%(company_name)s')

db.project.employee_name.readable =
db.project.employee_name.writable = False
```

We defined a two tables tables: "company" and "project". You can see the foreign key in the project "table", `reference client`. The "auth_user" table is an auto-generated table, among others. Also, the "employee_name" field in the "project"

table references the logged in user. So when a user posts a new project, their user information will automatically be added to the database. Save the file.

Fire up web2py in your terminal: `python web2py.py -a 'your password' -i
127.0.0.1 -p 8000`

Navigate to the **Edit** page and click the "database administration" button to execute the DAL commands.

Take a look at the *sql.log* file within the databases directory in Sublime to verify exactly which tables and fields were created. You can also read the documentation on all the auto-generated tables in the web2py official [documentation](#).

Notice the `format` attribute. All references are linked to the Primary Key of the associated table, which is the auto-generated ID (check the *sql.log*). By using the `format` attribute references will not show up by the id - but by the preferred field.

Register yourself as a new user, then setup a new a company, a project (associated with the newly created company), and an employee:

- Navigate to the following url: <http://localhost:8000/py2manager/default/user/login>
- Click register, then enter your information.
- Navigate to: <http://localhost:8000/py2manager/appadmin/index>
- Setup a dummy company and project.

Remember => web2py addresses a number of potential security flaws automatically. One of them is session management: web2py provides a built-in mechanism for administrator authentication, and it manages sessions independently for each application. The administrative interface also forces the use of secure session cookies when the client is not "localhost".

One less thing you have to worry about. For more information, please check out the web2py [documentation](#).

PUSH the changes to Github:

```
git add *
git commit -m 'updated database'
git push origin master
```

Homework

- Download the [web2py cheatsheet](#). Paste it on your wall. Frame it if you want.
Read it too.

8.5) URL Routing

Controllers describe the application logic and workflow in order to link the user with the application. More precisely, the controller controls the requests made by the users, obtains and organizes the desired information, and then responds back to the user via views and templates. For example, go to the Real Python [website](#) and log in. When you clicked the "Log In" button after you entered your credentials, a POST request was sent to the controller. The controller then took that information and compared it with the users in the MySQL database. Once your user credentials were found, this information was sent back to the controller. Then the controller redirected you to the appropriate view.

Web frameworks simplify this process significantly.

URL Routing

web2py provides a simple means of matching URLs with views.³ In other words, when the controller provides you with the appropriate view, there is an URL associated with that view, which can be customized.

Let's look at an example:

```
def index():
    return dict(message="Hello!")
```

This is just a simple function used to output the string "Hello!" to the screen. You can't tell from the above info, but the application name is "hello" and the controller used for this function is *default.py*. The function name is "index".

In this case the generated URL will be:

```
http://www.yoursite.com/hello/default/index.html
```

This is also the default URL routing method:

```
http://www.yousite.com/application_name/controller_name/
function_name.html
```

You can customize the URL routing methods in the *routes.example.py* file, which is found in the "web2py" root directory ("realpython/web2py/py2manager"). Just rename it to *routes.py*. For example, if you wanted to remove the controller_name from the url, add the following code to the *routes.py* file:

```
routers = dict(
    BASE  = dict(default_application='py2manager'),
)
```

Go ahead and make those changes.

Test this out. Restart the server. Navigate to the login page again:

<http://localhost:8000/py2manager/user/login>

For more information on URL routing, please see the official web2py documentation.

Let's setup the logic and URL routing in the py2manager app. Add the following code to *default.py*:

```
@auth.requires_login()
def index():
    project_form = SQLFORM(db.project).process()
    projects = db(db.project).select()
    users = db(db.auth_user).select()
    companies = db(db.company).select()
    return locals()
```

Here we displayed the data found in the "project", "auth_user", and "company" tables, as well as added a form for adding projects.

Most of the functionality is now in place for a basic application. We just need to update the views, organize the *index.html* page, and update the layout and styles. Before moving on though, PUSH the changes to Github.

Homework

- Please read more about the basic Git commands [here](#).

8.6) Initial Views

Views describe how the response should be translated to the user using mostly a combination of HTML, JavaScript, and CSS.

Some major components of the views include:

- Template Engine: Template engines are used for embedding Python into standard HTML. web2py uses a slightly modified Python syntax to make the code more readable. You can also define control statements such as for and while loops and if statements.

For example:

```
<html>
  <body>
    {{numbers = [1, 2, 3]}}
    <ul>
      {{for n in numbers:}}<li>{{=n}}</li>{{pass}}
    </ul>
  </body>
</html>
```

If you want to test this, first create a function in the controller:

```
def tester():
    return locals()
```

Then create a view, `tester.html`, with the above HTML.

- Template Composition: web2py can extend and include a set of sub templates. For example, you could have the main page, `index.html`, that extends `default.html`. Meanwhile, `default.html` could include two sub templates, `header.html` and `footer.html`:
 - Main `index.html` page:

```
{{extend 'default.html'}}
<h1>This is the index.html page</h1>
```

- The extended layout file, `default.html` must include an `{{include}}` directive, which embeds the HTML from `index.html`:

```
<html>
  <head>
    {{extend 'header.html'}}
  </head>
  <body>
    {{include}}
  </body>
    {{extend 'footer.html'}}
</html>
```

- jQuery libraries: web2py includes a number of jQuery and JavaScript libraries - a number of which are pre-configured. Refer to the web2py [documentation](#) for more information on jQuery, JavaScript, and other components of the views.

Let's build the templates and views for py2manager.

1. *index.html:*

```
 {{extend 'layout.html'}}
<h2>Welcome to py2manager</h2>
<br/>
{{=(project_form)}}
<br/>
<h3> All Open Projects </h3>
<ul>{{for project in projects:}}
  <li>
    {{=(project.name)}}
  </li>
  {{pass}}
</ul>
```

This file has a form at the top to add new projects to the database. It also lists out all open projects using a for loop. You can view the results here:

<http://localhost:8000/py2manager/index>.

2. *user.html:*

Open up this file. This view was created automatically to make the development process easier and quicker. If you go back to the `default.py` file, you can see a description of the main functionalities of the user function:

```
"""
exposes:
    http://..../[app]/default/user/login
    http://..../[app]/default/user/logout
    http://..../[app]/default/user/register
    http://..../[app]/default/user/profile
    http://..../[app]/default/user/retrieve_password
    http://..../[app]/default/user/change_password
use @auth.requires_login()
    @auth.requires_membership('group name')
    @auth.requires_permission('read','table
name',record_id)
    to decorate functions that need access control
"""

```

3. Layout:

Let's edit the main layout to replace the generic template. Start with `models/menu.py`. Update the following code:

```
response.logo =
A(B('py',SPAN(2,'manager')),XML('&trade;&ampnbsp'),
    _class="brand")
response.title = "py2manager"
response.subtitle = T('just another project manager')
```

Then update the application menu:

```
response.menu = [(T('Home'), False, URL('default', 'index'), []),
                  (T('Add Project'), False, URL('default', 'add'), []),
                  (T('Add Company'), False, URL('default', 'company'), []),
                  (T('Employees'), False, URL('default', 'employee'), [])]

DEVELOPMENT_MENU = False
```

Take a look at your changes. Remember to PUSH to Github:

```
git add *
git commit -m 'updated the templates and views'
git push origin master
```

Now that we've gone over the Model View Controller architecture in detail, let's now focus on the main functions of the application.

8.7) Profile Page

Remember the auto-generated "auth_user" table? Take a look at the *sql.log* for a quick reminder. Again, the *auth_user* table is part of a larger set of auto-generated tables aptly called the Auth tables.

It's easy to add fields to any of the Auth tables. Open up *db.py* in Sublime and place the following code after `auth = Auth(db)` and before

```
auth.define_tables() :
```

```
auth.settings.extra_fields['auth_user']= [
    Field('address'),
    Field('city'),
    Field('zip'),
    Field('image', 'upload')]
```

Save the file. Navigate to <http://localhost:8000/py2manager/index> and login if needed. Once logged in, you can see your name in the upper right-hand corner. Click the drop down arrow, then select "Profile". You should see the new fields. Go

ahead and update the new fields and upload an image. Then click Save Profile. Nice, right?

Profile

First name: Michael

Last name: Herman

E-mail: michael@mherman.org

Address: 328 Lexington Ave

City: San Francisco

Zip: 9412

Image:

No file chosen

[Choose File](#) [file] delete



Save profile

8.8) Add Projects

To clean up the homepage, let's move the form to add new projects to a separate page. Open your `default.py` file, and add a new view:

```
@auth.requires_login()
def add():
    project_form = SQLFORM(db.project).process()
    return dict(project_form=project_form)
```

Then update the index() view:

```
@auth.requires_login()
def index():
    projects = db(db.project).select()
    users = db(db.auth_user).select()
    companies = db(db.company).select()
    return locals()
```

Add a new template in the default directory called *add.html*:

```
{%extend 'layout.html'%}
<h2>Add a new project:</h2>
<br/>
{{=project_form.custom.begin}}
<strong>Project
name</strong><br/>{{=project_form.custom.widget.name}}<br/>
<strong>Company
name</strong><br/>{{=project_form.custom.widget.company_name}}<br/>
<strong>Description</strong><br/>{{=project_form.custom.widget.description}}<br/>
<strong>Start
Date</strong><br/>{{=project_form.custom.widget.start_date}}<br/>
<strong>Due
Date</strong><br/>{{=project_form.custom.widget.due_date}}<br/>
{{=project_form.custom.submit}}
{{=project_form.custom.end}}
```

In the controller, we used web2py's SQLFORM to generate a form automatically from the database. We then customized the look of the form using the following syntax: `form.custom.widget[fieldname]`.

Push to Github.

8.9) Add Companies

First, we need to add a form for adding new companies, which follows almost a nearly identical pattern as adding a form for projects. Try working on it on your own before looking at the code.

Default.py:

```
@auth.requires_login()
def company():
    company_form = SQLFORM(db.company).process()
    return dict(company_form=company_form)
```

Add a new template in the default directory called *company.html*:

```
{%extend 'layout.html'%}
<h2>Add a new company:</h2>
<br/>
{==company_form.custom.begin}
<strong>Company
Name</strong><br/>{==company_form.custom.widget.company_name}<br/>
<strong>Email</strong><br/>{==company_form.custom.widget.email}<br/>
<strong>Phone</strong><br/>{==company_form.custom.widget.phone}<br/>
<strong>URL</strong><br/>{==company_form.custom.widget.url}<br/>
{==company_form.custom.submit}
{==company_form.custom.end}
```

Push to Github.

8.10) Homepage

Now, let's finish organizing the homepage to display all projects. We'll be using the [SQLFORM.grid](#) to display all projects. Essentially, the SQLFORM.grid is a high-level table that creates complex CRUD controls. It provides pagination, the ability to browse, search, sort, create, update and delete records from a single table.

Let's look at a quick example.

1. Update the `index()` function in `default.py`:

```
@auth.requires_login()
def index():
    response.flash = T('Welcome!')
    grid = SQLFORM.grid(db.project)
    return locals()
```

`return locals()` is used to return a dictionary to the view, containing all the variables. It's equivalent to `return dict(grid=grid)`, in the above example. We also added a flash greeting.

2. Update the `index.html` view:

```
{{extend 'layout.html'}}
<h2>All projects:</h2>
<br/>
{{=grid}}
```

3. Navigate to <http://127.0.0.1:8000/py2manager> to view the new layout. Play around with it. Add some more projects. Download them in CSV. Notice how you can sort specific fields in ascending or descending order by clicking on the header links. This is the generic grid. Let's customize it to fit our needs.
4. Append the following code to the bottom of `db_tasks.py`:

```
db.project.start_date.requires =
IS_DATE(format=T('%m-%d-%Y'),
        error_message='Must be MM-DD-YYYY!')

db.project.due_date.requires =
IS_DATE(format=T('%m-%d-%Y'),
        error_message='Must be MM-DD-YYYY!')
```

This changes the date format from YYYY-MM-DD to MM-DD-YYYY. What happens if you use a lowercase `y` instead? Try it and see.

Test this out by adding a new project: <http://127.0.0.1:8000/add>. Use the built-in AJAX calendar. Oops. That's still inputting dates the old way. Let's fix that.

5. Within the views folder, open *web2py_ajax.html* and make the following changes:

Change:

```
var w2p_ajax_date_format = "{ {=T('%Y-%m-%d') } }";
var w2p_ajax_datetime_format = "{ {=T('%Y-%m-%d
%H:%M:%S') } }";
```

To:

```
var w2p_ajax_date_format = "{ {=T('%m-%d-%Y') } }";
var w2p_ajax_datetime_format = "{ {=T('%m-%d-%Y
%H:%M:%S') } }";
```

6. Now let's update the grid in the index() function:

```
grid = SQLFORM.grid(db.project, create=False,
fields=[db.project.name, db.project.employee_name,
db.project.company_name, db.project.start_date,
db.project.due_date, db.project.completed],
deletable=False, maxtextlength=50)
```

What does this do? Take a look at the documentation [here](#). It's all self-explanatory. Compare the before and after for additional help.

8.11) More Grids

First, let's add a grid to the company view.

1. *default.py*:

```
@auth.requires_login()
def company():
    company_form = SQLFORM(db.company).process()
    grid = SQLFORM.grid(db.company, create=False,
    deletable=False, editable=False, maxtextlength=50,
    orderby=db.company.company_name)
    return locals()
```

2. *company.html*:

```
{{extend 'layout.html'}}
<h2>Add a new company:</h2>
<br/>
{{=company_form.custom.begin}}
<strong>Company
Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>

<strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>

<strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
{{=company_form.custom.submit}}
{{=company_form.custom.end}}
<br/>
<br/>
<h2>All companies:</h2>
<br/>
{{=grid}}
```

Next, let's create the employee view:

1. *default.py*:

```
@auth.requires_login()
def employee():
    employee_form = SQLFORM(db.auth_user).process()
    grid = SQLFORM.grid(db.auth_user, create=False,
    fields=[db.auth_user.first_name, db.auth_user.last_name,
    db.auth_user.email], deletable=False, editable=False,
    maxtextlength=50)
    return locals()
```

2. *employee.html*:

```
{{extend 'layout.html'}}
<h2>All employees:</h2>
<br/>
{{=grid}}
```

Test both of the new views out, and then PUSH to Github.

8.12 Notes

Next, let's add the ability to add notes to each project.

1. Add a new table to the database to *db_task.py*:

```
db.define_table('note',
    Field('post_id', 'reference project', writable=False),
    Field('post', 'text', notnull=True),
    Field('created_on', 'datetime', default=request.now,
writable=False),
    Field('created_by', db.auth_user,
default=auth.user_id))

db.note.post_id.readable = db.note.post_id.writable = False
db.note.created_on.readable = db.note.created_on.writable
= False
db.note.created_on.requires = IS_DATE(format=T('%m-%d-%Y'),
error_message='Must be MM-DD-YYYY!')
db.note.created_by.readable = db.note.created_by.writable
= False
```

2. Update the index() function and add a note() function in the controller:

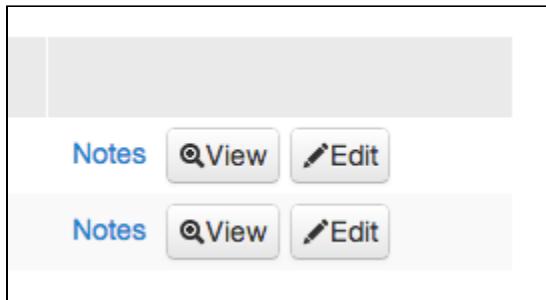
```
@auth.requires_login()
def index():
    response.flash = T('Welcome!')
    notes = [lambda project:
A('Notes',_href=URL("default","note",args=[project.id]))
        grid = SQLFORM.grid(db.project, create=False,
links=notes, fields=[db.project.name,
db.project.employee_name, db.project.company_name,
db.project.start_date, db.project.due_date,
db.project.completed], deletable=False, maxtextlength=50)
        return locals()

@auth.requires_login()
def note():
    project = db.project(request.args(0))
    db.note.post_id.default = project.id
    form = crud.create(db.note) if auth.user else "Login
to Post to the Project"
    allnotes = db(db.note.post_id==project.id).select()
    return locals()
```

3. Take a look. Add some notes. Now let's add a new view called *default/note.html*:

```
{ {extend 'layout.html'} }  
<h2>Project Notes</h2>  
<br/>  
<h4>Current Notes</h4>  
{ {for n in allnotes:}}  
    <ul>  
        <li>{ {=db.auth_user[n.created_by].first_name} } on  
{ {=n.created_on.strftime("%m/%d/%Y") }}  
            - { {=n.post}}</li>  
    </ul>  
    { {pass}}  
<h4>Add a note</h4>  
{ {=form}}<br>
```

4. Finally, let's update the index() function to add a button for the Notes link:



```
@auth.requires_login()  
def index():  
    response.flash = T('Welcome!')  
    notes = [lambda project: A('Notes', _class="btn",  
    _href=URL("default","note",args=[project.id]))]  
    grid = SQLFORM.grid(db.project, create=False,  
    links=notes, fields=[db.project.name,  
    db.project.employee_name, db.project.company_name,  
    db.project.start_date, db.project.due_date,  
    db.project.completed], deletable=False, maxtextlength=50)  
    return locals()
```

We just added the `.btn` class to the `notes` variable. How do we know where this class comes from? Open up the default template, `layout.html`, and find the area in the head, `<head></head>` where the CSS files are included:

```
<!-- include stylesheets -->
{ {
    response.files.append(URL('static','css/web2py.css'))
    response.files.append(URL('static','css/
bootstrap.min.css'))
    response.files.append(URL('static','css/
bootstrap-responsive.min.css'))
    response.files.append(URL('static','css/
web2py_bootstrap.css'))
} }
```

With CSS, the last file included (`web2py_bootstrap.css`) will overwrite any classes in the CSS files before it. So, start with the last included file to see if the `.btn` class is there, and if not, work your way up through the files.

PUSH the code to GitHub

8.13) Error Handling

web2py handles errors much differently than other frameworks. Tickets are automatically logged, and web2py does not differentiate between the development and production environments.

Have you seen an error yet? Remove the closing parenthesis from the statement in the `index()` function: `response.flash = T('Welcome!')`. Now navigate to the homepage. You should see that a ticket number was logged. When you click on the ticket number, you get the specific details regarding the error.

You can also view all tickets here: <http://localhost:8000/admin/errors/py2manager>

You do not want users seeing errors, so add the following code to the *routes.py* file:

```
routes_onerror = [
    ('*/*', '/py2manager/static/error.html')
]
```

Then add the *edit.html* file:

```
<h2>This is an error. We are working on fixing it.</h2>
```

Refresh the homepage to see the new error message. Now errors are still logged, but end users won't see them. Correct the error. PUSH the new code to GitHub.

Homework

- Please read the web2py [documentation](#) regarding error handling

8.14) Final Word

What's this app missing? Would you like to see any additional features? If so, please post your feedback in the Real Python [forum](#). Thanks!

-
1. <http://stackoverflow.com/questions/580383/where-is-the-line-between-dal-and-orm>
 2. <http://web2py.com/books/default/chapter/29/06>
 3. <http://web2py.com/books/default/chapter/29/04#URL-rewrite>

9) web2py: REST Redux

9.1) Introduction

Remember the data we scraped from Socrata back in Chapter 3? Go back and quickly review the lesson. Then, locate the *project.db* file on your local computer we used to store the data. In short, we're going to build our own RESTful web service to expose the data that we scraped. Why would we want to do this when the data is already available?

1. The data could be in high demand but the Socrata website is unreliable. By scraping the data and providing it via REST, you can ensure the data is always available to you or your clients.
2. Again, the data could be in high demand but it's poorly organized on the website. You can cleanse the data after scraping and offer it in a more human and machine readable format.
3. You want to create a mashup. Perhaps you are scraping other websites (legally) that also have data sources and you're creating an aggregator service.

Whatever the reason, let's look at how to quickly setup a REST web service via web2py by to expose the data we pulled.

Remember =>

- *Each resource or endpoint should be identified by a separate URL.*
- *There are four HTTP methods used for interacting with Databases (CRUD):*
 - *read (GET)*
 - *create (POST)*
 - *update (PUT)*
 - *delete (DELETE).*

Let's start with a basic example before using the scraped data from Socrata. We'll be using the web2py IDE for this chapter for simplicity's sake.

9.2) Basic REST

Navigate to your web2py directory. Fire up virtualenv. Load web2py. Then create a new app called rest.

To set up REST follow these steps:

1. Create a new database table in in *db.py*:

```
db.define_table('fam',Field('role'),Field('name'))
```

2. Enter some dummy data into the database.
3. Add the RESTful functions to the controller: [1](#)

```
@request.restful()
def api():
    response.view = 'generic.'+request.extension
    def GET(*args,**vars):
        patterns = 'auto'
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)
    def POST(table_name,**vars):
        return db[table_name].validate_and_insert(**vars)
    def PUT(table_name,record_id,**vars):
        return
    db(db[table_name]._id==record_id).update(**vars)
    def DELETE(table_name,record_id):
        return db(db[table_name]._id==record_id).delete()
    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

These functions expose any field in our database. If you want to limit the resources exposed, you'll need to define various patterns.

For example:

```
def GET(*args, **vars):
    patterns = [
        "/test[fam]",
        "/test/{fam.name.startswith}",
        "/test/{fam.name}/:field",
        ]
    parser = db.parse_as_rest(patterns, args, vars)
    if parser.status == 200:
        return dict(content=parser.response)
    else:
        raise HTTP(parser.status, parser.error)
```

So:

- `http://127.0.0.1:8000/rest/default/api/test` => GET all data
- `http://127.0.0.1:8000/rest/default/api/test/t` => GET a single data point where the "name" starts with "t"
- `http://127.0.0.1:8000/rest/default/api/test/1` => Can you guess what this does? Go back and look at the database schema for help.

For simplicity, let's expose everything.

4. Test out the following GET requests in your browser:

- URI:

```
http://127.0.0.1:8000/rest/default/api/fam.json
```

Result:

```
{"content": [{"role": "Father", "id": 1, "name": "Tom"}, {"role": "Mother", "id": 2, "name": "Jane"}, {"role": "Brother", "id": 3, "name": "Jeff"}, {"role": "Sister", "id": 4, "name": "Becky"}]}
```

Explanation: GET all data

- URI:

```
http://127.0.0.1:8000/rest/default/api/fam/id/1.json
```

Results:

```
{"content": [{"role": "Father", "id": 1, "name": "Tom"}]}
```

Explanation: GET data where "id" == 1

5. Test out the following requests in the Shell and look at the results after each requests in the database:

```
>>> import requests
>>> payload = {"name" : "john", "role" : "brother"}
>>> r = requests.post("http://127.0.0.1:8000/rest/default/
api/fam.json", data=payload)
>>> print r
<Response [200]>

>>> r = requests.delete("http://127.0.0.1:8000/rest/
default/api/fam/2.json")
>>> print r
<Response [200]>

>>> payload = {"name" : "Jeffrey"}
>>> r = requests.put("http://127.0.0.1:8000/rest/default/
api/fam/3.json", data=payload)
>>> print r
<Response [200]>
```

6. Now in most cases, you do not want just anybody having access to your model like this. Besides, limiting the datapoints as described above, you also want to have user authentication in place.
7. Register a new user (<http://127.0.0.1:8000/rest/default/user/register>), and then update the function in the controller, adding a login required decorator:

```
auth.settings.allow_basic_login = True
@auth.requires_login()
@request.restful()
def api():
    response.view = 'generic.'+request.extension
    def GET(*args,**vars):
        patterns = 'auto'
        parser = db.parse_as_rest(patterns,args,vars)
        if parser.status == 200:
            return dict(content=parser.response)
        else:
            raise HTTP(parser.status,parser.error)
    def POST(table_name,**vars):
        return db[table_name].validate_and_insert(**vars)
    def PUT(table_name,record_id,**vars):
        return
    db(db[table_name]._id==record_id).update(**vars)
    def DELETE(table_name,record_id):
        return db(db[table_name]._id==record_id).delete()
    return dict(GET=GET, POST=POST, PUT=PUT,
               DELETE=DELETE)
```

8. Now you need to be authenticated to make any requests:

Unauthorized:

```
>>> import requests
>>> from requests.auth import HTTPBasicAuth
>>> payload = {"name" : "Katie", "role" :
"Cousin"}
>>> auth = HTTPBasicAuth("Michael", "reallyWRONG")
>>> r = requests.post("http://127.0.0.1:8000/rest/default/
api/fam.json", data=payload, auth=auth)
>>> print r
<Response [403]>

>>> auth = HTTPBasicAuth("michael@mherman.org",
"reallyREAL")
>>> r = requests.post("http://127.0.0.1:8000/rest/default/
api/fam.json", data=payload, auth=auth)
>>> print r
<Response [200]>
```

9. Test this out some more.

Homework

- Please watch [this](#) short video on REST.

9.3) Advanced REST

All right. Now that you've seen the basics of creating a RESTful web service. Let's build a more advanced example using the Socrata data.

1. Create a new app called "socrata."
2. Register a new user: <http://127.0.0.1:8000/socrata/default/user/register>
3. Create a new tables with the following schema:

```
db.define_table('socrata', Field('name'), Field('url'), Field('views',
integer))
```

4. Now we need to extract the data from the *projects.db* data and import into the new database table you just created. There are a number of different ways to handle this. [2](#) We'll export the data from the old database in CSV format and then import it directly into the new web2py table.

- Open *projects.db* in your SQLite Browser. Then click File => Export => Table as CSV file. Save the file in the following directory as *socrata.csv*:

```
..../realpython/web2py/socrata/applications/socrata
```

- You need to rename the "text" field since it's technically a restricted name. Also, the field names "text" and "url" are labeled under the wrong columns respectively. Open up the CSV file in gedit and make the following changes to the header:

| | "url" | "name" | "views" |
|----|---|---|---------|
| 1 | "/Government/2010-Report-to-Congress-on-White-House-Staff/vedg-c5ob" | "2010 Report to Congress on White House Staff", "498280" | |
| 3 | "/Government/The-White-House-Nominations-Appointments-5qsd-mtsn" | "The White House - Nominations & Appointments", "418075" | |
| 4 | "/Government/2011-Report-to-Congress-on-White-House-Staff/73b8-rwdg" | "2011 Report to Congress on White House Staff", "330053" | |
| 5 | "/Government/Milk-RadNet-Laboratory-Analysis/pkjf-5jst" | "Milk RadNet Laboratory Analysis", "209690" | |
| 6 | "/Government/Precipitation-RadNet-Laboratory-Analysis/e2xy-undq" | "Precipitation RadNet Laboratory Analysis", "192020" | |
| 7 | "/Government/Le-Sarkom-tre/mm2p-747g" | "Le Sarkomètre", "182317" | |
| 8 | "/Government/2012-Annual-Report-to-Congress-on-White-House-Staff/jv7a-cjdv" | "2012 Annual Report to Congress on White House Staff", "166695" | |
| 9 | "/Government/Drinking-Water-RadNet-Laboratory-Analysis/4lg7-9eqd" | "Drinking Water RadNet Laboratory Analysis", "146778" | |
| 10 | "/Government/Sorted-RadNet-Laboratory-Analysis/w9fb-tgv6" | "Sorted RadNet Laboratory Analysis", "144936" | |
| 11 | "/Government/Air-Filter-Cartridge-RadNet-Laboratory-Analysis/3d4g-9muv" | "Air Filter & Cartridge RadNet Laboratory Analysis", "111070" | |
| 12 | "/dataset/Outpatient-Drug-and-Alcohol-Treatments/73p4-7cmf" | "Outpatient Drug and Alcohol Treatments", "2" | |

- Also, open the file in Excel. Use the Text-to-Columns feature on the "views" column to extract out the word "views" from each data point. The word "views" makes it impossible to filter and sort the data. When we scrapped the data, we should have only grabbed the view count, not the word "views". Mistakes are how you learn, though. If you don't know how to do this in Excel, I've included the *socrata.csv* file formatted correctly.
- To upload the CSV file, return to the **Edit** page on web2py, click the button for "database administration", then click the "db.socrata" link. Scroll to the bottom of the page and click "choose file" select *socrata.csv*. Now click import.
- There should now be 20,000+ rows of data in the "socrata" table:

| 20194 selected | | | | |
|-------------------------------|------------|------------------|------------------|---------------|
| next 100 rows | | | | |
| | socrata.id | socrata.name | socrata.url | socrata.views |
| 1 | | 2010 Report t... | /Government/2... | 498280 |
| 2 | | The White Hou... | /Government/T... | 418075 |
| 3 | | 2011 Report t... | /Government/2... | 330053 |
| 4 | | Milk RadNet ... | /Government/M... | 209690 |
| 5 | | Precipitation... | /Government/P... | 192020 |
| 6 | | Le Sarkomètre | /Government/L... | 182317 |
| 7 | | 2012 Annual R... | /Government/2... | 166695 |
| 8 | | Drinking Wate... | /Government/D... | 146778 |
| 9 | | Sorted RadNet... | /Government/S... | 144936 |
| 10 | | Air Filter & ... | /Government/A... | 111070 |
| 11 | | Outpatient Dr... | /dataset/Outp... | 2 |
| 12 | | Alternative D... | /dataset/Alte... | 2 |
| 13 | | Outpatient Dr... | /dataset/Outp... | 2 |
| 14 | | Outpatient Dr... | /dataset/Outp... | 2 |

In the future, when you set up your Scrapy Items Pipeline, you need to dump the data right to the web2py database. The process is the same as outlined in Chapter 3. Also, make sure to only grab the view count, not the word "views".

Now let's design the actual API.

1. First, When designing your RESTful API, you should follow these best practices: [3](#)
 - Keep it simple and intuitive,
 - Use HTTP methods,
 - Provide HTTP status codes,
 - Use simple URLs for accessing endpoints/resources,

- JSON should be the format of choice, and
- Use only lowercase characters.

2. Add the following code to *default.py*:

```
@request.restful()  
def api():  
    response.view = 'generic.'+request.extension  
    def GET(*args,**vars):  
        patterns = 'auto'  
        parser = db.parse_as_rest(patterns,args,vars)  
        if parser.status == 200:  
            return dict(content=parser.response)  
        else:  
            raise HTTP(parser.status,parser.error)  
    def POST(table_name,**vars):  
        return db[table_name].validate_and_insert(**vars)  
    def PUT(table_name,record_id,**vars):  
        return  
        db(db[table_name]._id==record_id).update(**vars)  
    def DELETE(table_name,record_id):  
        return db(db[table_name]._id==record_id).delete()  
    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

GET

1. Navigate to the following URL to see the resources/end points that are available via GET:

```
http://127.0.0.1:8000/socrata/default/api/patterns.json
```

Output:

```
{"content": ["/socrata[socrata]", "/socrata/  
id/{socrata.id}", "/socrata/id/{socrata.id}[:field",  
"/socrata/views/{socrata.views.ge}/{socrata.views.lt}",  
"/socrata/  
views/{socrata.views.ge}/{socrata.views.lt}[:field"]]}
```

Endpoints:

- `http://127.0.0.1:8000/socrata/default/api/socrata.json`
- `http://127.0.0.1:8000/socrata/default/api/socrata/id/[id].json`
- `http://127.0.0.1:8000/socrata/default/api/socrata/id/[id]/[field_name].json`
- `http://127.0.0.1:8000/socrata/default/api/socrata/
views/[start_range]/[end_range].json`
- `http://127.0.0.1:8000/socrata/default/api/
socrata/[start_range]/[end_range]/[field_name].json`

Let's look at each one in detail with the Python Shell.

Import the requests library to start:

```
>>> import requests
```

2. `http://127.0.0.1:8000/socrata/default/api/socrata.json`

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/  
default/api/socrata.json")  
>>> print r  
<Response [400]>  
>>> print r.content  
too many records
```

3. `http://127.0.0.1:8000/socrata/default/api/socrata/id/[id].json`

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/
default/api/socrata/id/100.json")
>>> print r
<Response [200]>
>>> print r.content
{"content": [{"name": "Ohio arsons per 1,000 by county",
"views": 1, "url": "/dataset/
Ohio-arsons-per-1-000-by-county/8axx-frtc", "id": 100}]}}
```

4. [http://127.0.0.1:8000/socrata/default/api/socrata/id/\[id\]/\[field_name\].json](http://127.0.0.1:8000/socrata/default/api/socrata/id/[id]/[field_name].json)

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/
default/api/socrata/id/100/views.json")
>>> print r
<Response [200]>
>>> print
r.content
{"content": [{"views": 1}]}

>>> r = requests.get("http://127.0.0.1:8000/socrata/
default/api/socrata/id/100/name.json")
>>> print r
<Response [200]>
>>> print
r.content
{"content": [{"name": "Ohio arsons per 1,000 by county"}]}
```

5. [http://127.0.0.1:8000/socrata/default/api/socrata/
views/\[start_range\]/\[end_range\].json](http://127.0.0.1:8000/socrata/default/api/socrata/
views/[start_range]/[end_range].json)

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/views/300000/1000000.json")
>>> print
r
<Response [200]>
>>> print
r.content
{"content": [{"name": "The White House - Nominations & Appointments", "views": 418075, "url": "/Government/The-White-House-Nominations-Appointments/n5m4-mism", "id": 2}, {"name": "2011 Report to Congress on White House Staff", "views": 330053, "url": "/Government/2011-Report-to-Congress-on-White-House-Staff/73t8-rw4g", "id": 3}]}}
```

6. [http://127.0.0.1:8000/socrata/default/api/socrata/\[start_range\]/\[end_range\]/\[field_name\].json](http://127.0.0.1:8000/socrata/default/api/socrata/[start_range]/[end_range]/[field_name].json)

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/views/300000/1000000/name.json")
>>> print r
<Response [200]>
>>> print r.content
{"content": [{"name": "The White House - Nominations & Appointments"}, {"name": "2011 Report to Congress on White House Staff"}]}
```

POST

1. <http://127.0.0.1:8000/socrata/default/api/socrata.json>

```
>>> payload = {'name': 'new database', 'url': 'http://new.com', 'views': '22'}
>>> r = requests.post("http://127.0.0.1:8000/socrata/default/api/socrata.json")
>>> print r
<Response [200]>
```

PUT

1. `http://127.0.0.1:8000/socrata/default/api/socrata/[id].json`

```
>>> payload = {'name':'new  
database'}  
>>> r = requests.put("http://127.0.0.1:8000/socrata/  
default/api/socrata/3.json", data=payload)  
>>> print r  
<Response [200]>
```

DELETE

1. `http://127.0.0.1:8000/socrata/default/api/socrata/[id].json`

```
>>> r = requests.delete("http://127.0.0.1:8000/socrata/  
default/api/socrata/3.json")  
>>> print r  
<Response [200]>
```

Patterns

Again, you can customize the patterns to define the specific available endpoints.

For example:

```
patterns = [  
    "/socrata/{socrata.name.startswith}",  
    "/socrata/{socrata.name}/:field",  
    "/socrata/{socrata.id}/:field",  
    "/socrata/{socrata.id}"  
]
```

1. `http://127.0.0.1:8000/socrata/default/api/socrata/{socrata.name.startswith}`

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/government.json")
>>> print r
<Response [200]>
>>> print r.content
{"content": [{"name": "Government Employees with salaryum greater than 100K", "views": 5, "url": "/dataset/Government-Employees-with-salaryum-greater-than-10/ctwy-exdw", "id": 1168}, {"name": "Government", "views": 878, "url": "/dataset/Government/g5vf-f4hx", "id": 1269}, {"name": "Government Services", "views": 758, "url": "/Business/Government-Services/st9p-sb8m", "id": 1619}, {"name": "Government data is the property of taxpayers and should be free to all citizens", "views": 658, "url": "/Government/Government-data-is-the-property-of-taxpayers-and-s/u83v-7srs", "id": 1855}, {"name": "Government Website Satisfaction", "views": 18, "url": "/dataset/Government-Website-Satisfaction/j3zd-539b", "id": 10144}, {"name": "government relations", "views": 33, "url": "/dataset/government-relations/peg6-sm86", "id": 19010}]} }
```

2. <http://127.0.0.1:8000/socrata/default/api/id/{socrata.id}>

```
>>> r = requests.get("http://127.0.0.1:8000/socrata/default/api/id/1168.json")
>>> print r
<Response [200]>
>>> print r.content
{"content": [{"name": "Government Employees with salaryum greater than 100K", "views": 5, "url": "/dataset/Government-Employees-with-salaryum-greater-than-10/ctwy-exdw", "id": 1168}]} }
```

3. <http://127.0.0.1:8000/socrata/default/api/socrata/{socrata.id}>

```
>>> r = requests.delete("http://127.0.0.1:8000/socrata/
default/api/socrata/1168.json")
>>> print
r
<Response [200]>
```

Try adding additional [patterns](#)), and test out the API using various POST methods.

Authentication

Finally, make sure to add the login() required decorator to the api() function, so that users have to be registered to make API calls.

```
auth.settings.allow_basic_login = True
@auth.requires_login()
```

Unauthorized:

```
>>> r = requests.delete("http://127.0.0.1:8000/socrata/
default/api/socrata/1168.json")
>>> print
r
<Response [403]>
```

-
1. http://web2py.com/books/default/chapter/29/10#parse_as_rest-experimental
 2. <http://web2py.com/books/default/chapter/29/6#Legacy-databases-and-keyed-tables>
 3. <http://devo.ps/blog/2013/03/22/designing-a-restful-api-that-doesn-t-suck.html>

10) Django: Quickstart

10.1) Overview

Like web2py, Django is a high-level web framework, which supports elegant, beautiful design and rapid web development. With a strong community of supporters and some of the largest and most popular sites using it such as Reddit, Instagram, Mozilla, Pinterest, Disqus, and Rdio, to name a few [1](#), it's the most well-known and used Python web framework. In spite of that, Django has a high learning curve due to much of the implicit automation that happens in the backend. It's much more important to understand the basics - e.g., the Python syntax and language, web client and server fundamentals, etc. - and then move on to lighter-weight/minimalist frameworks (like Flask or bottle.py) so that when you do start developing with Django, it will be much easier to obtain a deeper understanding of the automation and its integrated functionality. Even web2py, which is slightly more automated, is easier to learn because it was specifically designed as a learning tool.



Django projects are logically organized around the Model-View-Controller (MVC) architecture. However, Django's architecture is slightly different in that the views act as the controllers. So, projects are actually organized in a Model-Template-Views architecture (MTV):

- **Models** represent the backend data, traditionally in the form of a relational database. Django uses the Django-ORM to organize and manage databases, which functions in relatively the same manner, despite a much different syntax, as SQLAlchemy and web2py's DAL.
- **Templates** visually represent the data model. This is the presentation layer and defines how information is displayed to the end user.
- **Views** define the business logic (much like the controllers in MVC architecture), which logically link the templates and models

I know this is a bit confusing, but just remember that the MTV and MVC architectures work the same: [2](#)

If you're familiar with other MVC Web-development frameworks ... you may consider Django views to be the controllers and Django templates to be the views.

This is an unfortunate confusion brought about by differing interpretations of MVC.

In Django's interpretation of MVC, the view describes the data that gets presented to the user; it's not necessarily just how the data looks, but which data is presented.

In contrast ... similar frameworks suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly how the data looks, not which data is presented.

In this chapter, you'll see how easy it is to get a project up due to the automation of common web development tasks and included integrated functions (batteries included). As long as you are aware of the inherent structure and organization (scaffolding) that Django uses, you can focus less on monotonous tasks, inherent in web development, and more on developing the higher-level portions of your application.

Brief History

Django grew organically from the web developers at the Lawrence Journal-World newspaper in Lawrence, Kansas (home of the University of Kansas) in 2003. The

developers realized that web development up to that point followed a similar formula/pattern resulting in much redundancy: [3](#)

1. Write a Web application from scratch.
2. Write another Web application from scratch.
3. Realize the application from step 1 shares much in common with the application from step 2.
4. Refactor the code so that application 1 shares code with application 2.
5. Repeat steps 2-4 several times.
6. Realize you've invented a framework.

The developers found that the commonalities shared between most applications could (and should) be automated. Django came to fruition from this rather simple realization, changing the state of web development as a whole.

Homework

- Please read Django's design [Philosophies](#)
- Optional: To gain a deeper understanding of the history of Django and the current state of web development, please read the first [chapter](#) of the Django book.

10.2) Installation

1. Within your terminal, navigate to your "realpython" directory, and then run the following command to create a virtualenv for your Django projects from this chapter:

```
$ virtualenv django --no-site-packages
```

2. Navigate into the "django" directory (the Django root directory), then activate the virtualenv:

Unix:

```
$ source bin/activate
```

Windows:

```
$ scripts\activate
```

3. Now install Django 1.5:

```
$ pip install Django==1.5
```

Please make sure to install Django version 1.5, which as of writing, is the latest stable release.

If you need to check the Django version currently installed on your machine, open the Python shell and run the following commands:

```
>>> import django
>>> django.VERSION
(1, 5, 0, 'final', 0)
```

Again, when you are done working with your Django project simply type `deactivate` to exit the virtualenv. Then to reactivate the virtualenv, navigate to the "django" directory and type one of the following:

Unix:

```
$ source bin/activate
```

Windows:

```
$ scripts\activate
```

10.3) Hello, World!

Basic Setup

1. Navigate to the "django" directory. Activate your virtualenv. Start a new Django project:

Unix:

```
$ django-admin.py startproject sites
```

Windows:

```
$ django-admin.py startproject sites
```

2. This will create the basic project layout (commonly referred to as the scaffolding), containing one directories and five files:

```
└── manage.py
    └── sites
        ├── __init__.py
        ├── settings.py
        ├── urls.py
        └── wsgi.py
```

If you just type `django-admin.py` you'll be taken to the help section, which displays all the available commands that can be performed with `django-admin.py`.

For now you just need to worry about these files `manage.py`, `settings.py`, and `urls.py`:

- **manage.py**: This file is a command-line utility, used to manage and interact with your project. You probably won't ever have to edit the file itself; however, it is used with almost every process as you develop your Project.

- **settings.py:** This is your project settings file for your project, where you configure your project's resources, such as database connections, external applications, and template files. There are numerous defaults setup in this file, which often get changed as you develop your Project.
 - **urls.py:** This file contains the URL mappings, connecting URLs to Views.
3. Before we start creating our **Hello, World!** application, let's make sure everything is setup correctly by running the development server. Navigate into the first *sites* directory and run the following command:

```
$ python manage.py runserver
```

You can specify a different port with the following command (if necessary):

```
$ python manage.py runserver 8080
```

You should see something similar to this:

```
Validating models...

0 errors found
June 03, 2013 - 13:07:43
Django version 1.5, using settings 'sites.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Navigate to <http://localhost:8000/> to view the development server:

It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Here's what to do next:

- If you plan to use a database, edit the `DATABASES` setting in `helloworld/settings.py`.
- Start your first app by running `python manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

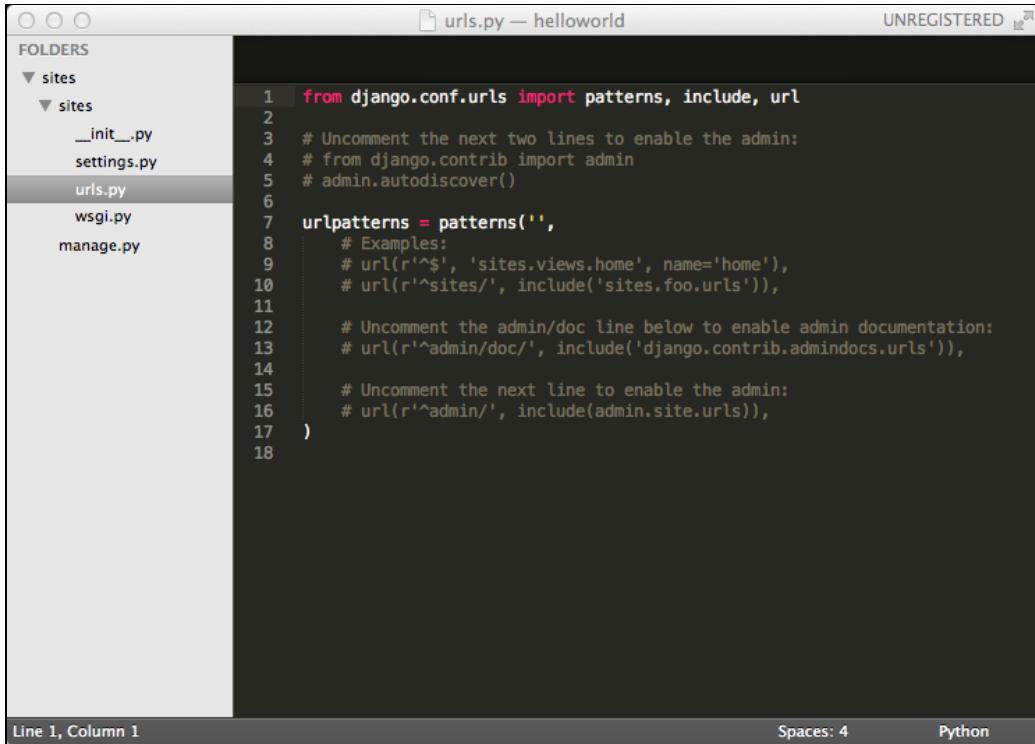
Exit the development server by pressing Control-C within the terminal.

Adding a Project to Sublime 2

It's much easier to work with Sublime 2 by adding all the Django files to a Sublime Project:

- Navigate to Project => Add folder to Project
- Find the first "sites" directory then click Open
- Save the project for easy access by navigating to Project => Save Project and save as `sites.sublime-project` within your "django" directory.

You should see both directories and all five files on the left pane:



```
urls.py — helloworld
UNREGISTERED 21

FOLDERS
sites
  sites
    __init__.py
    settings.py
    urls.py
    wsgi.py
  manage.py

1 from django.conf.urls import patterns, include, url
2
3 # Uncomment the next two lines to enable the admin:
4 # from django.contrib import admin
5 # admin.autodiscover()
6
7 urlpatterns = patterns('',
8     # Examples:
9     # url(r'^$', 'sites.views.home', name='home'),
10    # url(r'^sites/', include('sites.foo.urls')),
11
12    # Uncomment the admin/doc line below to enable admin documentation:
13    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
14
15    # Uncomment the next line to enable the admin:
16    # url(r'^admin/', include(admin.site.urls)),
17 )
18

Line 1, Column 1
Spaces: 4
Python
```

Create a new App

1. Now that the Django project and development environment are setup, let's create a new app. With virtualenv activated, navigate to your "sites" directory, and then run the following command:

```
python manage.py startapp helloworld
```

This will create a new directory called "helloworld", which includes the following files:

- **models.py**: This file is used to define your data models that are connected to the database.
- **tests.py**: This houses your test files used for setting up unit and integration tests (don't worry about this for now).
- **views.py**: This file is your application's controller (as mentioned above), defining the business logic in order to render a view to the browser.

Your Project structure should now look like this:

```
|── helloworld
|   ├── __init__.py
|   ├── models.py
|   ├── tests.py
|   └── views.py
├── manage.py
└── sites
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

What's the difference between a Project and an App? A project is the main web app/site, containing the settings, templates, and URL routes for a set of Django apps. Meanwhile, a Django app is just an application that has an individual function such as a blog or message forum. Each app must have a separate function, distinct from other apps. Django apps are really used to encapsulate common functionalities. Put another way, the project is your final product, comprised of separate functions from each app.

2. Next, we need to include the new app in the `settings.py` file so that Django knows that it exists. Scroll down to "INSTALLED_APPS" and add the app name, **helloworld**.

Your "INSTALLED_APPS" section should look like this-

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'helloworld',
    # Uncomment the next line to enable the admin:
    #'django.contrib.admin',
    # Uncomment the next line to enable admin
    documentation:
        # 'django.contrib.admindocs',
)
```

3. Open the `views.py` file and add the following code:

```
from django.http import HttpResponse

def hello_view(request):
    return HttpResponse('<html><body>Hello,
World!</body></html>')
```

What's going on here?

- This function takes a parameter, `request`, which is an object that has information about the request, from a user, that triggered the view.
- We named the function `hello_view` but as you will see in a second, this doesn't matter - you can name the function whatever you wish. By convention, though, make sure the function name represents the main objective of the function itself.
- A response object is then instantiated, which returns the text "Hello, World!" to the browser.

4. With the view setup, we now just need to add it to the `urls.py` file to link a URL to that specific view. With `urls.py` open, add this code to the `urlpatterns`:

```
urlpatterns = patterns('',
    url(r'^hello/$', 'helloworld.views.hello_view')
)
```

We'll go over the exact syntax for generating the views in the next chapter.

5. Let's test it out. Navigate to the first "sites" directory in your terminal, fire up the server (`python manage.py runserver`), and then open your browser to <http://localhost:8000/hello>.

It worked! You should see the "Hello, World!" text in the browser.

6. Navigate back to the root directory - <http://localhost:8000/>. You should see a 404 error, because we did not setup a view that maps to `/`.

Let's change that.

Open up your `urls.py` file again and update the code:

```
urlpatterns = patterns('',
    url(r'^hello/$', 'helloworld.views.hello_view'),
    url(r'^$', 'helloworld.views.hello_view')
)
```

Save the file and refresh the page. You should see the same "Hello, World!" text. So, we simply assigned or mapped two URLs (`/` and `/hello`) to that single view.

Homework

Experiment with adding additional text-based views and assigning them to URLs.

10.3) Templates

Django templates are really the same as web2py views, which are used for displaying HTML to the user. You can also embed Python into the templates. Let's modify the example above to include templates.

1. Navigate to the first "sites" directory and create a new directory called "templates". Your project structure should now look like this:

```
├── helloworld
│   ├── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── sites
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
    └── templates
```

2. Next, let's update the `settings.py` file to update the `TEMPLATE_DIRS`.

As stated in the file make sure to use strings and absolute paths - i.e., "/home/html/realpython/django/sites/templates" or "C:/www/realpython/django/sites/templates". Also, always use forward slashes regardless of whether you're working from a Windows or Unix environment.

My file looks like this, for example:

```
TEMPLATE_DIRS = (
    '/Users/michaelherman/desktop/realpython/django/sites/
    templates'
)
```

3. Let's update the `views.py` file with a new function():

```
from django.template import Context, loader
from datetime import datetime
from django.http import HttpResponseRedirect

def hello_view(request):
    return HttpResponseRedirect('<html><body>Hello,
World!</body></html>')

def better_hello(request)
    t = loader.get_template('betterhello.html')
    c = Context({'current_time': datetime.now(), })
    return HttpResponseRedirect(t.render(c))
```

So in the `better_hello()` function we use the `loader` method along with the `get_template` function to return a template called `betterhello.html`. The `Context` class takes a dictionary of variable names, as the dict's keys, and their associated values, as the dict's values (which is similar to the web2py syntax). Finally, the `render()` function returns the `Context` variables to the template.

4. Next, let's setup the template. Create a new HTML file called `betterhello.html` in the "template" directory and pass in the key from the dictionary surrounded by double curly braces `{{ }}`. Place the following code in the file:

```
<html>
<head><title>A Better Hello!</title>
<body>
    <p>Hello, World! This template was rendered on
{{current_time}}.</p>
</body>
</html>
```

5. Finally, we need to add the view to the `urls.py` file:

```
url(r'^better/$', 'helloworld.views.better_hello')
```

6. Fire up your server and navigate to <http://localhost:8000/better>. You should see something like this:

"Hello, World! This template was rendered on June 4, 2013, 11:54 a.m."

7. Notice the time. Is it correct? The time zone defaults to U.S. Central Time. If you would like to change it, open the *settings.py* file, and then change the COUNTRY/CITY based on the timezones found in [Wikipedia](#).

For example, if you change the time zone to `TIME_ZONE = 'America/Los_Angeles'`, and refresh <http://localhost:8000/better>, it should display U.S. Pacific Time.

Change the time zone so that the time is correct based on where you live.

10.4) Bloggy: A blog app

In this next project, we'll develop a simple blog.

Model

1. Navigate to your "django" directory and activate your virtualenv, then create a new project called **bloggy**:

Unix:

```
$ django-admin.py startproject bloggy
```

Windows:

```
$ django-admin.py startproject bloggy
```

2. Navigate into the newly created "bloggy" directory and launch the Django development server to ensure the new project was setup correctly:

```
$ python manage.py runserver
```

Open <http://localhost:8000/> and you should see the light-blue Welcome to Django screen.

3. Add your Django project folder to Sublime as a new project.
4. Now let's setup a relational database. Open up `settings.py` and append `sqlite3"` to the end of the `Engine` key and then add the path to the `Name` key. You can name the database whatever you'd like because if the database doesn't exist, Django will create it for you when you sync the database in the next step.

Again, make sure to use absolute paths and forward slashes - i.e., `"/home/html/realpython/django/bloggy/database_name.db"` or `"C:/www/realpython/django/bloggy/database_name.db"`.

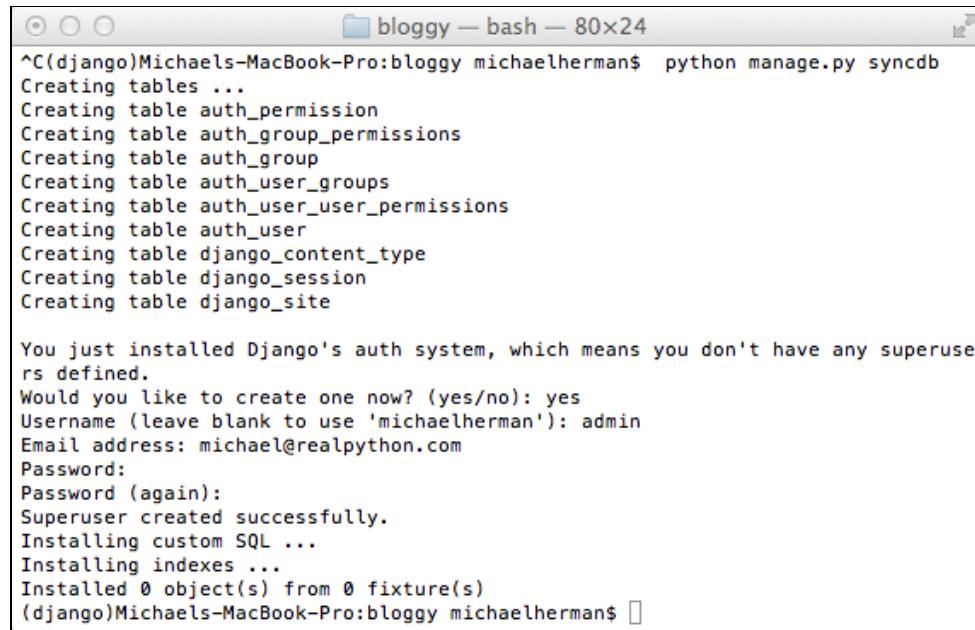
The results should look like something similar to this, depending upon your path:

```
'ENGINE': 'django.db.backends.sqlite3',
'NAME': '/Users/michaelherman/desktop/realpython/django/
bloggy/bloggy.db',
```

5. Next, create and sync the database:

```
$ python manage.py syncdb
```

This will create the basic tables and ask you to setup a superuser. Use "admin" for both your username and password:



```
^C(django)Michaels-MacBook-Pro:bloggy michaelherman$ python manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'michaelherman'): admin
Email address: michael@realpython.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
(django)Michaels-MacBook-Pro:bloggy michaelherman$
```

6. Start a new app:

```
$ python manage.py startapp blog
```

7. Before moving on, let's put this under version control with Git using the following three commands (feel free to also push to Github):

```
$ git init
$ git add *
$ git commit -m "Init Commit"
```

8. Finally, we can define the model using the Django ORM rather than straight SQL. Add the following code to *model.py*:

```
from django.db import models

class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=100)
    content = models.TextField()
```

This class, which inherits some of the basic properties from the standard Model class, defines the database table as well as each field - *created_at*, *title*, and *content*.

Much like SQLAlchemy and web2py's DAL, the Django ORM provides a database abstraction layer used for interacting with a database using CRUD (create, read, update, and delete) via Python objects⁴. Flip back to lesson 8.4 to see SQLAlchemy and web2py's DAL to compare the differences between the three abstraction layers.

Note that the primary key, a unique id, and the `created_at` timestamp will both be automatically generated for us when objects are added to the database. We just need to add the `title` and `content` when creating new objects (database rows).

9. Add the new app to the `settings.py` file:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog'
```

10. Before we sync the database, we can view the actual SQL statements that will be used by running the following command:

```
$ python manage.py sql blog
```

Your output should look like this:

```
BEGIN;  
CREATE TABLE "blog_post" (  
    "id" integer NOT NULL PRIMARY KEY,  
    "created_at" datetime NOT NULL,  
    "title" varchar(100) NOT NULL,  
    "content" text NOT NULL  
)  
;
```

Compare the field types here to the field types used in the ORM. You can view all the available field types on the Django [website](#).

11. We can also check to see if there are any errors in the data model:

```
$ python manage.py validate
```

12. And finally, execute the SQL statements:

```
$ python manage.py syncdb
```

Django Shell

Django provides an interactive Python shell for accessing the Django API.

1. Use the following command to start the Shell:

```
$ python manage.py shell
```

2. Let's add some data to the table we just created via the database API.⁵ Start by importing the model Class we created:

```
>>> from blog.models import Post
```

3. If you search for objects in the table (somewhat equivalent to `select * from blog_post`) you should find that it's empty:

```
>>> Post.objects.all()
```

So let's add some data!

4. To add new rows, we can use the following commands:

```
>>> p = Post(title="What Am I Good At?", content="What am I  
good at? What is my talent? What makes me stand out? These  
are the questions we ask ourselves over and over again and  
somehow can not seem to come up with the perfect answer.  
This is because we are blinded, we are blinded by our own  
bias on who we are and what we should be. But discovering  
the answers to these questions is crucial in branding  
yourself. You need to know what your strengths are in order  
to build upon them and make them better")  
>>> p.save()  
>>> p = Post(title="Charting Best Practices: Proper Data  
Visualization", content="Charting data and determining  
business progress is an important part of measuring  
success. From recording financial statistics to webpage  
visitor tracking, finding the best practices for charting  
your data is vastly important for your company's success.  
Here is a look at five charting best practices for optimal  
data visualization and analysis.")  
>>> p.save()  
>>> p = Post(title="Understand Your Support System Better  
With Sentiment Analysis", content="There's more to  
evaluating success than monitoring your bottom line. While  
analyzing your support system on a macro level helps to  
ensure your costs are going down and earnings are rising,  
taking a micro approach to your business gives you a  
thorough appreciation of your business' performance.  
Sentiment analysis helps you to clearly see whether your  
business practices are leading to higher customer  
satisfaction, or if you're on the verge of running clients  
away.")  
>>> p.save()
```

Remember that we don't need to add a primary `id` or the `created_at` timestamp as those are auto-generated.

Now if you search for all objects, three objects should be returned:

```
>>> Post.objects.all()  
[<Post: Post object>, <Post: Post object>, <Post: Post  
object>]
```

When we use the `all()` or `filter()` function, a `QuerySet` is returned, which is iterable.

Notice how `<Poll: Poll object>` returns absolutely no distinguishing information about the object. Let's change that. Open up your `models.py` file and add the following code:

```
def __unicode__(self):  
    return self.title
```

Your file should now look like this:

```
from django.db import models  
  
class Post(models.Model):  
    created_at = models.DateTimeField(auto_now_add=True)  
    title = models.CharField(max_length=100)  
    content = models.TextField()  
  
    def __unicode__(self):  
        return self.title
```

This is a bit confusing, as Python Classes are usually returned with `__str__`, not `__unicode__`. We use unicode because Django models use unicode by default⁶.

Save your `models.py` file, exit the Shell, re-open the Shell, import the Post model Class again (`from blog.models import Post`), and now run the query `Post.objects.all()`. You should now see -

```
[<Post: What Am I Good At?>, <Post: Charting Best Practices: Proper Data Visualization>, <Post: Understand Your Support System Better With Sentiment Analysis>]
```

This should be much easier to read and understand. We know there are three rows in the table, and we know their titles. Depending on how much information you want returned, you could add all the fields to the `model.py` file:

```
def __unicode__(self):
    return str(self.id) + " / " + str(self.created_at) + " / "
    + self.title + " / " + self.content + "\n"
```

Test this out. What does this return? Make sure to update this so it's just returning the title:

```
def __unicode__(self):
    return self.title
```

Open up your SQLite Browser to make sure the data was added correctly:

The screenshot shows the SQLite Database Browser interface. The title bar reads "SQLite Database Browser - /Users/michaelherman/Desktop/realpython/django/bloggy/blog...". The toolbar has icons for file operations like Open, Save, and Close, along with database-related icons like Create, Drop, and Modify. Below the toolbar are three tabs: "Database Structure", "Browse Data" (which is selected), and "Execute SQL". A "Table:" dropdown menu is set to "blog_post". There are "New Record" and "Delete Record" buttons. The main area displays a table with three rows:

| | id | created_at | title | content |
|---|-----------|-------------------|--------------------------|---|
| 1 | 1 | 2013-06-05 15:07 | What Am I Good At? | What am I good at? What is my talent? What make |
| 2 | 2 | 2013-06-05 15:08 | Charting Best Practices: | Charting data and determining business progress |
| 3 | 3 | 2013-06-05 15:08 | Understand Your Suppor | There's more to evaluating success than monitor |

At the bottom, there are navigation buttons for < and >, a page number indicator "1 - 3 of 3", a "Go to:" input field with value "0", and a scroll bar.

5. Let's look at how to query the data from the Shell.

- Return objects by a specific id:

```
>>> Post.objects.filter(id=1)
```

- Return objects by ids > 1:

```
>> Post.objects.filter(id__gt=1)
```

- Return objects where the title contains a specific word:

```
>>> Post.objects.filter(title__contains='data')
```

If you want more info on querying databases via the Django ORM in the Shell, take a look at the official [documentation](#). And if you want a challenge, add

more data and practice querying using straight SQL and then convert it over to objects via the Django ORM.

Unit Tests for models

Now that the database table is setup, let's write a quick unit test. It's common practice to write tests as you develop new models and functions. Make sure to include at least one test for each function within your *model.py* files.

1. Add the following code to *tests.py*:

```
from django.test import TestCase
from models import Post

class PostTests(TestCase):
    def test_str(self):
        my_title = Post(title='This is a basic title for a
basic test case')
        self.assertEquals(str(my_title), 'This is a basic
title for a basic test case',)
```

2. Run the test case:

```
$ python manage.py test blog
```

You can run all the tests in the Django project with the following command - `python manage.py test`; or you can run the tests from a specific app, like in the command above.

You should see the following output:

```
Creating test database for alias 'default'...
```

```
..
```

```
Ran 1 tests in 0.000s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

One thing to note is that since this test needed to add data to a database to run, it created a temporary database and then destroyed it after the test ran. This prevents the test from accessing the real database and possibly damaging the database by mixing test data with real data.

Remember: *Anytime you make changes to an existing model or function, re-run the tests. If they fail, find out why. You may need to re-write them depending on the changes made.*

Django Admin

Depending how familiar you are with the Django ORM and SQL statements in general, it's probably much easier to access and modify the data model using the Django Admin.

1. To access the admin, we need to enable the app and setup the URL.

- Open the `settings.py` file and uncomment the

`'django.contrib.admin'` line:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
    'django.contrib.admin'
)
```

- Then sync the database again:

```
$ python manage.py syncdb
```

- Open *urls.py* and uncomment these three lines-

```
from django.contrib import admin
admin.autodiscover()
url(r'^admin/', include(admin.site.urls)),
```

- Within your "blog" directory add a new file called *admin.py* that contains the following code:

```
from blog.models import Post
from django.contrib import admin

admin.site.register(Post)
```

2. Now let's access the Django Admin. Fire up the server and navigate to <http://localhost:8000/admin> within your browser. Enter your login credentials ("admin" and "admin"). You can now see the model Class:

The screenshot shows the Django administration interface. At the top, it says "Django administration" and "Site administration". Below that, there's a list of models: "Auth", "Groups", "Users", "Blog", "Posts", and "Sites". Each model has a "Add" button and a "Change" button next to it. The "Posts" row is highlighted with a blue background, and a red arrow points to the "Posts" link. The "Blog" section is also highlighted with a blue background.

3. Add some more posts, change some posts, delete some posts. Go crazy.

Templates and Views

As mentioned, Django's views are equivalent to the controllers in most other MVC-style web frameworks. The views are generally paired with templates to generate HTML to the browser. Let's look at how to add the basic templates and views to our blog.

Before we start, create a new directory within the "blog" directory called "templates", and then add the relative path to the `settings.py` file:

```
TEMPLATE_DIRS = (
    '/Users/michaelherman/desktop/realpython/django/bloggy/blog/
    templates')
```

Note: The above code will vary depending upon the path and environment (Windows or Unix) you are using.

index.html

- **view:** Add the following code to the `views.py` file:

```
from django.http import HttpResponseRedirect
from blog.models import Post
from django.template import Context, loader

def index(request):
    latest_posts =
Post.objects.all().order_by('-created_at')
    t = loader.get_template('index.html')
    c = Context({'latest_posts': latest_posts, })
    return HttpResponseRedirect(t.render(c))
```

- **template:** Create a new file called *index.html* within the "templates" file and add the following code:

```
<h1>Bloggy: a blog app</h1>
{% if latest_posts %}
<ul>
  {% for post in latest_posts %}
    <li>{{post.created_at}} | {{post.title}} |
      {{post.content}}</li>
  {% endfor %}
</ul>
{% endif %}
```

- **url:** Update the *urls.py* file:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/$', 'blog.views.index'),
)
```

- Let's test it out. Fire up the server and navigate to <http://localhost:8000/blog/>. You should have something that looks like this:

Bloggy: a blog app

- June 5, 2013, 10:07 a.m. | What Am I Good At? | What am I good at? What is my talent? What makes me stand out? These are the questions we ask ourselves over and over again and somehow can not seem to come up with the perfect answer. This is because we are blinded, we are blinded by our own bias on who we are and what we should be. But discovering the answers to these questions is crucial in branding yourself. You need to know what your strengths are in order to build upon them and make them better.
- June 5, 2013, 10:08 a.m. | Charting Best Practices: Proper Data Visualization | Charting data and determining business progress is an important part of measuring success. From recording financial statistics to webpage visitor tracking, finding the best practices for charting your data is vastly important for your company's success. Here is a look at five charting best practices for optimal data visualization and analysis.
- June 5, 2013, 10:08 a.m. | Understand Your Support System Better With Sentiment Analysis | There's more to evaluating success than monitoring your bottom line. While analyzing your support system on a macro level helps to ensure your costs are going down and earnings are rising, taking a micro approach to your business gives you a thorough appreciation of your business' performance. Sentiment analysis helps you to clearly see whether your business practices are leading to higher customer satisfaction, or if you're on the verge of running clients away.
- June 5, 2013, 11:13 a.m. | Sentiment Analysis: Feelings, Not Facts | Sentiment analysis is a process, which gathers the latest input from a human source or several human sources and uses it to determine the general opinion of a person, place or thing. For example, the old-fashioned comment boxes and comment cards is a form of sentiment analysis. Often called opinion gathering, sentiment analysis can be generated from various technological sources, like Twitter or Facebook. Like the old fashioned comment box, sentiment analysis can help owners or managers find the strengths and weaknesses of a business.
- June 5, 2013, 11:15 a.m. | Differentiating Between Bounce Rates | You must understand that bounce rate and exit rate are not the same. Bounce rate is the percentage of people who visit any page of your site without going further. They click off within seconds, or even a split second. Exit rate, on the other hand, is when people leave your site after staying for more than a few seconds. Every visitor needs to exit your website, not every one needs to bounce off

Let's clean this up a bit.

- **template (second iteration):** Update *index.html*

```
<html>
<head> <title>Bloggy: a blog app</title> </head>
<body>
    <h2>Welcome to Bloggy!</h2>
    {% if latest_posts %}
        <ul>
            {% for post in latest_posts %}
                <h3><a href="/blog/{{ post.id }}">{{ post.title }}</a></h3>
                <p><em>{{ post.created_at }}</em></p>
                <p>{{ post.content }}</p>
                <br/>
            {% endfor %}
        </ul>
    {% endif %}
</body>
</html>
```

- If you refresh the page now, you'll see that it's easier to read. Also, each post title is now a link. Try clicking on the link. You should see a 404 error because we have not set up the URL routing or the template. Let's do that now.
- Before moving on, notice how we used two kind of braces within the template. The first, `{ % % }`, is used to signify Python logic such as a IF statements or For loops, and the second, `{ { } }`, is used for inserting variables.

post.html

- **url:** Update the `urls.py` file:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/$', 'blog.views.index'),
    url(r'^blog/(?P<post_id>\d+)/$', 'blog.views.post'),
)
```

Take a look at the new URL. The pattern (`(?P<post_id>\d+)`) we used is made up of regular expressions. Click [here](#) to learn more about regular expressions. You can also test out regular expressions with the [Python Regular Expression Tool](#):

Python Regular Expression Testing Tool

| | |
|---|---|
| Regular Expression (regex) | <input type="text" value="blog/(?P<post_id>\d+)"/> |
| Flags | <input type="checkbox"/> Ignore case <input type="checkbox"/> Locale <input type="checkbox"/> Multi line <input type="checkbox"/> Dot all <input type="checkbox"/> Unicode <input type="checkbox"/> Verbose |
| String | <input type="text" value="blog/1"/> |
| Timeit | <input type="checkbox"/> Timeit loops <input type="text" value="10000"/> |
| <input type="button" value="Test Regex"/> | |

Code:

```
>>> regex = re.compile("blog/(?P<post_id>\d+)")
>>> r = regex.search(string)
>>> r
<_sre.SRE_Match object at 0x4b2a4b0c45dc9950>
>>> regex.match(string)
<_sre.SRE_Match object at 0x4b2a4b0c45c51768>

# List the groups found
>>> r.groups()
(u'1',)

# List the named dictionary objects found
>>> r.groupdict()
{u'post_id': u'1'}

# Run.findall
>>> regex.findall(string)
[u'1']
```

- **view:** Add a new function to `views.py`:

```
from django.shortcuts import get_object_or_404

def post(request, post_id):
    single_post = get_object_or_404(Post, pk=post_id)
    t = loader.get_template('post.html')
    c = Context({'single_post': single_post,})
    return HttpResponseRedirect(t.render(c))
```

This function accepts two parameters: the request object and the `post_id`, which is the number parsed from the URL by the regular expression we setup. Meanwhile the `get_object_or_404` method queries the database for objects by a specific type (id) and returns that object if found. If not found, it returns a 404 error.

- **template:** Create a new template called `post.html`:

```
<html>
<head> <title>Bloggy: {{ single_post.title }}</title>
</head>
<body>
  <h2>{{ single_post.title }}</h2>
  <ul>
    <p><em>{{ single_post.created_at }}</em></p>
    <p>{{ single_post.content }}</p>
    <br/>
  </ul>
  <p>Had enough? Return <a href="/blog">home</a>.<br/>
</body>
</html>
```

- Go back to your blog on the development server and test out the links for each post. They should all be working now.

As far as functionality is concerned, we could add a number of features, such as the ability to add new posts from the app itself (not just from the admin) as well as a comment section. However, we've explored this simple blog long enough. There's plenty of Django blog tutorials on the Internet. Let's add some basic styles, and then move on to something more interesting.

Homework

Please read about the [Django Models](#) for more information on the Django ORM syntax.

Styles

Before adding any styles, let's break our templates into parent and child templates, so that the child templates will inherit the HTML and styles from the parent template. We've covered this a number of times before so I won't go into great detail.

1. Create a new template file called *base.html*. This is the parent file. Add the following code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Bloggy: a blog app</title>
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <meta name="description" content="">
        <meta name="author" content="">

        <!-- Le styles -->
        <link href="http://twitter.github.io/bootstrap/assets/
css/bootstrap.css" rel="stylesheet">
        <style>
            body {
                padding-top: 60px; /* 60px to make the container go
all the way to the bottom of the topbar */
            }
        </style>
        <link href="http://twitter.github.io/bootstrap/assets/
css/bootstrap-responsive.css" rel="stylesheet">

        <!-- Le HTML5 shim, for IE6-8 support of HTML5 elements
-->
        <!--[if lt IE 9]>
            <script src="http://html5shim.googlecode.com/svn/
trunk/html5.js"></script>
        <![endif]-->
        <script src="http://twitter.github.io/bootstrap/assets/
js/jquery-1.8.1.min.js" type="text/javascript"></script>
        {%
            block extrahead %}
        {%
            endblock %}
        <script type="text/javascript">
            $(function() {
                {%
                    block jquery %}
                {%
                    endblock %}
            });
        </script>
    </head>
```

```
<body>

    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="btn btn-navbar" data-toggle="collapse"
data-target=".nav-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </a>
                <a class="brand" href="/blog">Bloggy</a>
                <div class="nav-collapse collapse">
                    <ul class="nav">
                        <li><a href="#">Add Post</a></li>
                    </ul>
                </div><!-- .nav-collapse -->
            </div>
        </div>
    </div>

    <div id="messages">
        {% if messages %}
            {% for message in messages %}
                <div class="alert alert-{{message.tags}}">
                    <a class="close" data-dismiss="alert">x</a>
                    {{message}}
                </div>
            {% endfor %}
        {% endif %}
    </div>

    <div class="container">
        {% block content %}
        {% endblock %}
    </div> <!-- /container -->
</body>
</html>
```

2. Update *index.html*:

```
{% extends 'base.html' %}

{% block content %}
  {% if latest_posts %}
    <ul>
      {% for post in latest_posts %}
        <h3><a href="/blog/{{ post.id }}"/>{{ post.title
} }</a></h3>
        <p><em>{{ post.created_at }}</em></p>
        <p>{{ post.content }}</p>
        <br/>
      {% endfor %}
    </ul>
  {% endif %}
  {% endblock %}
```

3. Update *post.html*:

```
{% extends 'base.html' %}

{% block content %}
  <h2>{{ single_post.title }}</h2>
  <ul>
    <p><em>{{ single_post.created_at }}</em></p>
    <p>{{ single_post.content }}</p>
    <br/>
  </ul>
  <p>Had enough? Return <a href="/blog">home</a>.<br/>
  {% endblock %}
```

Take a look at the results. Amazing what five minutes can do.

Finally, update Git. In your terminal, navigate to your *Bloggy* directory and run the following commands:

```
$ git init $ git add * $ git commit -m "updated"
```

South

Before moving on to the next chapter, let's look at how to handle database migrations (database schema changes) in Django.

1. Open up your `models.py` file and add two new fields to the table:

```
tag = models.CharField(max_length=20, blank=True, null=True)
image = models.ImageField(upload_to="images", blank=True,
null=True)
```

By setting `blank=True` and `null=True`, we are indicating that the field is not required and can be left blank when adding a new row to the table. The other field, as you probably guessed is for adding pictures to posts.

2. Sync the database in your terminal:

```
$ python manage.py syncdb
```

3. Fire up your development server and login to the admin page, and then try to add a new row to the database that includes a tag. You should see the error "table blog_post has no column named tag" because the field we tried to add didn't get added to the database. That's a problem. Fortunately, there is an easy solution: South. [7](#)

4. Install South via pip:

```
$ pip install south
```

5. Then make sure to add it to your INSTALLED_APPS within your `settings.py` file. Remove the `tag` and `image` fields from `models.py` sync the database and run the following code to initiate the migration:

```
$ python manage.py convert_to_south blog
```

6. Once again add the new fields to the model and run these commands to update the schema:

```
$ python manage.py schemamigration blog --auto  
$ python manage.py migrate blog
```

7. Run the server, go the Django admin, and you now should be able to add rows with a tag and/or and image. Boom.
8. Now let's update the the application so that images will be viewable on the post page.
 - Update *posts.html*:

```
{% extends 'base.html' %}  
  
{% block content %}  
<h2>{{ single_post.title }}</h2>  
<ul>  
    <p><em>{{ single_post.created_at }}</em></p>  
    <p>{{ single_post.content }}</p>  
    <p>Tag: {{ single_post.tag }}  
    <br/>  
    <p>  
        {% if single_post.image %}  
              
        {% endif %}  
    </ul>  
    <p>Had enough? Return <a href="/blog">home</a>.<br/>  
    {% endblock %}
```

- Update *settings.py*:

```
MEDIA_ROOT = '/Users/michaelherman/desktop/realpython/django/bloggy/blog/static/media'  
MEDIA_URL = '/media/'  
STATIC_ROOT = '/Users/michaelherman/desktop/realpython/django/bloggy/blog/static'  
STATIC_URL = '/static/'  
STATICFILES_DIRS = ('/Users/michaelherman/desktop/realpython/django/bloggy/blog/static/')
```

- Update *urls.py*:

```
urlpatterns = patterns('',  
    url(r'^admin/', include(admin.site.urls)),  
    url(r'^blog/$', 'blog.views.index'),  
    url(r'^media/(?P<path>.*)$',  
        'django.views.static.serve',{'document_root': '/Users/  
michaelherman/desktop/realpython/django/bloggy/bloggy/  
static/media'}),  
    url(r'^blog/(?P<post_id>\d+)/$', 'blog.views.post'),  
)
```

- Add an image. Refresh!

Homework

- Now that we can add a tag to our posts, on the *post.html* page, list out a tag associated with the post.

For example - "Tag: python"

- Go over the Django Quickstart at <http://realdjango.herokuapp.com/>
- Optional: You're ready to go through the Django [tutorial](#). For beginners, this tutorial can be pretty confusing. However, since you now have plenty of web development experience and have already created a basic Django app, you should have no trouble. Have fun! Learn something.

-
1. <http://www.djangosites.org/>
 2. <http://stackoverflow.com/questions/6621653/django-vs-model-view-controller>
 3. <http://www.djangobook.com/en/2.0/chapter01.html>
 4. <https://docs.djangoproject.com/en/1.5/topics/db/models/>
 5. <https://docs.djangoproject.com/en/1.5/topics/db/queries/>
 6. <https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>
 7. <http://south.aeracode.org/>

11) Django: Ecommerce Site

11.1) Overview

In the past ten chapters you've learned the web fundamentals from the ground up, hacked your way through a number of different exercises, asked questions, tried new things, etc. - and now it's time to put it all together and build a practical, real-world application, using the principles of rapid web development. After you complete this project, you should be able to take your new skills out into the real world and develop your own project.

As you probably guessed, we will be developing a basic ecommerce site.

First, let's talk about Rapid Web Development.

11.2) Rapid Web Development

What exactly is rapid web development? Well, as the name suggests it's about building a web site or application quickly and efficiently. The focus is on efficiency. It's relatively easy to develop a site quickly, but it's another thing altogether to develop a site quickly that meets the functional needs and requirements your potential users will demand.

As you have seen, development is broken into logical chunks. If we were starting from complete scratch we'd begin with prototyping to define the major features and design a mockup through iterations: Prototype, Review, Refine. After each stage you generally get more and more detailed, from low to high-fidelity, until you've hashed out all the features and prepared a mockup complex enough to add a web framework in order to apply the MVC-style architecture.

If you are beginning with low-fidelity, start prototyping with a pencil and paper. Avoid the temptation to use prototyping software until the latter stages, as these can be restricting. *Do not stifle your creativity in the early stages.*

As you define each function and apply a design, put yourself in the end users' shoes. What can he see? What can she do? Do they really care? For example, if one of your application's functions is to allow users to view a list of search results from an airline aggregator in pricing buckets, what does this look like? Are the results text or graphic-based? Can the user drill down on the ranges to see the prices at a granular level? And, most importantly, does the end user care? Will this be a differentiator in your product versus the competition? Perhaps not. But there should be some function(s) that do separate your product from your competitor's products. Or perhaps your functionality is the same - you just implement it better?

Finally, rapid web development is one of the most important skills a web developer can have, especially developers who work at startups. Speed is the main advantage that startups have over their larger, more established competition. The key is to understand each layer of the development process, from beginning to end.

11.3) Prototyping

Prototyping is the process of building a working model of your application, from a front-end perspective, allowing you to test and refine your application's main features and functions. Again, it's common practice to begin with a low-fidelity prototype.

From there you can start building a storyboard, which traces the users' movements. For example, if the user clicks the action button and s/he is taken to a new page, create that new page. Again, for each main function, answer these three questions:

1. What can he see?
2. What can she do?
3. Do they really care?

If you're building out a full-featured web application take the time to define in detail every interaction the user has with the website. Create a story board, and then

after plenty of iterations, build a high-fidelity prototype. One of the quickest means of doing this is via the front-end framework, Bootstrap.

First, let's get a basic Django Project and App setup. With this app, you can implement other pieces to develop your own web app.

In most cases, development begins with defining the model and constructing the database first. Since we're creating a rapid prototype, we'll be starting with the front-end first, adding the most important functions, then moving to the back-end. This is vital for when you develop your basic minimum viable prototype. The goal is to create an app quick to validate your product and business model. Once validated, you can finish developing your product, adding all components you need to transform your prototype into a project.

11.4) Setup your Project and App

Basic Setup

1. Create a new directory in "realpython" called "mvp" within your terminal, and then navigate into the new directory.
2. Setup virtualenv, and then activate:

```
$ virtualenv --no-site-packages myenv
```

Unix:

```
$ source myenv/bin/activate
```

Windows:

```
$ myenv\scripts\activate
```

3. Install Django:

```
$ pip install django==1.5
```

4. Start a new Django project:

```
$ django-admin.py startproject project_name
```

5. Add your Django project folder to Sublime as a new project.
6. Navigate into the "project_name" directory, and then place your project under version control:

```
$ git init  
$ git add .  
$ git commit -m "initial commit"
```

7. Setup a new app:

```
$ python manage.py startapp main
```

Due to the various functions of this project, we will be creating a number of different apps. Each app will have a different function within your main project.

Your project structure should now look like this:

```
├── manage.py  
└── main  
    ├── __init__.py  
    ├── models.py  
    ├── tests.py  
    └── views.py  
└── project_name  
    ├── __init__.py  
    ├── settings.py  
    ├── urls.py  
    └── wsgi.py
```

8. Add your app to the INSTALLED_APPS section of *settings.py* as well as your project's relative paths (add this to the top of *settings.py*):

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'main',
```

and

```
import os
PROJECT_ROOT = os.path.realpath(os.path.dirname(__file__))
SITE_ROOT = os.path.dirname(PROJECT_ROOT)
```

The realpath() function is used to access the relative path within your file system.¹

9. Create a new directory within the "project_name" directory called "templates", and then add the relative path to the *settings.py* file:

```
TEMPLATE_DIRS = (os.path.join(SITE_ROOT, 'templates'),)
```

10. Configure the admin page.

Uncomment the following url pattern in *settings.py*:

```
'django.contrib.admin',
```

Uncomment these three lines from *urls.py*:

```
from django.contrib import admin
admin.autodiscover()

url(r'^admin/', include(admin.site.urls)),
```

11. Commit your changes:

```
$ git add .
$ git commit -m "added main app"
```

Remember: Your development environment should be isolated from the rest of your development machine (via virtualenv) so your dependencies don't clash with one another. In addition, you should recreate the production environment as best you can on your development machine, meaning you should use the same dependency versions. We will set up a requirements.txt file later on to handle this. Finally, it may be necessary to develop on a virtual machine to recreate the exact OS and even the same release so that bugs found within production are easily reproducible in development. The bigger the app, the greater the need for this.

Add a landing page

Now that the main app is up, let's quickly add a barebones landing page.

1. Add the following code to the views.py file:

```
from django.shortcuts import render_to_response
from django.template import RequestContext

def index(request):
    return render_to_response('index.html',
context_instance=RequestContext(request))
```

This function, `index()`, takes a parameter, `request`, which is an object that has information about the user requesting the page from the browser. The function's response is to simply render the `index.html` template. In other

words, when a user navigates to the *index.html* page (the request), the django controller renders the *index.html* template (the response).

2. Next, we need to add a new pattern to *urls.py*:

```
url(r'^$', 'main.views.index', name='home'),
```

3. Finally, we need to create the *index.html* template. First create a new file called *base.html* (parent template) within the templates directory, and add the following code:

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="utf-8">
        <title>Your MVP</title>
        <meta name="viewport" content="width=device-width,
initial-scale=1.0">
        <meta name="description" content="Setting up an MVP with
Django by RealPython.com">
        <meta name="author" content="Michael Herman">
        <!-- Le styles -->
        <link href="http://www.realpython.com/css/bootstrap.css"
rel="stylesheet">
        <style>

            body {
                padding-top: 20px;
                padding-bottom: 40px;
            }

            /* Custom container */
            .container-narrow {
                margin: 0 auto;
                max-width: 700px;
            }
            .container-narrow > hr {
                margin: 30px 0;
            }

            /* Main marketing message and sign up button */
            .jumbotron {
                margin: 60px 0;
                text-align: center;
            }
            .jumbotron h1 {
                font-size: 72px;
                line-height: 1;
            }

        </style>
```

```
.jumbotron .btn {  
    font-size: 21px;  
    padding: 14px 24px;  
}  
  
/* Supporting marketing content */  
.marketing {  
    margin: 60px 0;  
}  
.marketing p + h4 {  
    margin-top: 28px;  
}  
  
</style>  
<!-- Le HTML5 shim, for IE6-8 support of HTML5 elements  
-->  
<!--[if lt IE 9]>  
    <script src="http://html5shim.googlecode.com/svn/trunk/  
html5.js">  
    </script>  
<![endif]-->  
<!-- Le fav and touch icons -->  
</head>  
<body>  
  
<div class="container-narrow">  
  
<div class="masthead">  
    <ul class="nav nav-pills pull-right">  
        <li><a href="{% url 'home' %}">Home</a></li>  
        <li><a href="#">About</a></li>  
        <li><a href="#">Contact</a></li>  
    </ul>  
    <h3><span class="fui-settings-16 muted">Your  
MVP!</span></h3>  
    </div>  
  
<hr>
```

```
{% if messages %}  
<div class="alert alert-success">  
    <div class="messages">  
        {% for message in messages %}  
            {{ message }}  
        {% endfor %}  
    </div>  
</div>  
{% endif %}  
  
<div class="container-narrow">  
    {% block content %}  
    {% endblock %}  
</div>  
  
<hr>  
  
<div class="footer">  
    <p>Copyright © 2013 <a  
 href="http://www.yoursite.com">Your MVP</a>. Developed by  
<a href="http://www.mherman.org">Your Name Here</a>.  
Powered by Django.</P></p>  
    </div>  
  
</div>  
</body>  
</head>
```

Next, add the *index.html* (child) template:

```
{% extends 'base.html' %}

{% block content %}

<div class="jumbotron">
    
    <h1>Please put some text here.</h1>
    <p class="lead">If you want, you can add some text here
as well. Or not.</p>
    <a class="btn btn-large btn-success"
href="contact">Contact us today to get
started</a><br/><br/><br/>
    <a href="https://twitter.com/RealPython"
class="twitter-follow-button" data-show-count="false"
data-size="large">Follow @RealPython</a>
    <script>!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0],p=/^http:/.test(d.location)?'http':'https';if
widgets.js';fjs.parentNode.insertBefore(js,fjs);}}(document,
'script', 'twitter-wjs');
    </script>
    &nbsp;&nbsp;&nbsp;&nbsp;
    <a href="https://twitter.com/intent/
tweet?screen_name=RealPython"
class="twitter-mention-button" data-size="large">Send us a
Tweet to @RealPython</a>
    <script>!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0],p=/^http:/.test(d.location)?'http':'https';if
widgets.js';fjs.parentNode.insertBefore(js,fjs);}}(document,
'script', 'twitter-wjs');
    </script>
    &nbsp;&nbsp;&nbsp;&nbsp;
    <a href="https://twitter.com/share"
class="twitter-share-button"
data-url="http://www.realpython.com" data-text="Django for
the Non-Programmer #realpython -"
data-size="large">Share!</a>
    <script>!function(d,s,id){var
js,fjs=d.getElementsByTagName(s)[0],p=/^http:/.test(d.location)?'http':'https';if
```

```
widgets.js';fjs.parentNode.insertBefore(js,fjs);} }(document,
'script', 'twitter-wjs');
    </script>
</div>

<hr>

<div class="row-fluid marketing">
    <div class="span6">
        <h3>(1) One</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
gravida nulla vitae arcu blandit, non congue elit tempus.
Suspendisse quis vestibulum diam.</p>
        <h3>(2) Two</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
gravida nulla vitae arcu blandit, non congue elit tempus.
Suspendisse quis vestibulum diam.</p>
        <h3>(3) Three</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
gravida nulla vitae arcu blandit, non congue elit tempus.
Suspendisse quis vestibulum diam.</p>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
gravida nulla vitae arcu blandit, non congue elit tempus.
Suspendisse quis vestibulum diam.</p>
    </div>

    <div class="span6">
        <h3>(4) Four</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
gravida nulla vitae arcu blandit, non congue elit tempus.
Suspendisse quis vestibulum diam.</p>
        <h3>(5) Five</h3>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent
```

```
gravida nulla vitae arcu blandit, non congue elit tempus.  
Suspendisse quis vestibulum diam.</p>  
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing  
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent  
gravida nulla vitae arcu blandit, non congue elit tempus.  
Suspendisse quis vestibulum diam.</p>  
    <h3>(6) Six</h3>  
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing  
elit. Mauris ut dui ac nisi vestibulum accumsan. Praesent  
gravida nulla vitae arcu blandit, non congue elit tempus.  
Suspendisse quis vestibulum diam.</p>  
    </p>  
    </div>  
  
</div>  
  
{ % endblock %}
```

4. Commit your changes:

```
$ git add .  
$ git commit -m "added templates and main view"
```

5. Fire up your server:

```
$ python manage.py runserver
```

Your MVP!

Home About Contact



Please put some text here.

If you want, you can add some text here as well. Or not.

Contact us today to get started

Looking good so far, but let's add some more pages. Before that, though, let's take a step back and talk about the design we used: Bootstrap.

Bootstrap

We used the Bootstrap front end framework to quickly add a great looking design. However, it should be customized, which is not difficult to do. In fact, as long as you have a basic understanding of CSS and HTML, you shouldn't have a problem. That said, customizing Bootstrap can be time-consuming. It takes practice to get fast at it. Try not to get discouraged as you build out your prototype. Work on one single area at a time, then take a break. Remember: It does not have to be perfect - it just has to give users, and potential viewers/testers of the prototype, a better sense of how your application works.

Make as many changes as you want. Learning CSS like this is just a trial and error process. You just have to make a few changes, then refresh the browser, and then

make more changes or update or revert old changes. Again, it takes time to know what will look good. Eventually, after much practice, you will find that you will be spending less and less time on each section, as you know how to get a good base quickly and then you can focus on creating something unique.

If you're working on your own application, [Jetstrap](#) is a great resource for creating quick Bootstrap prototypes. Try it out.

Homework

- Optional: If you want to learn more about CSS and Bootstrap quickly, go through the tutorial found [here](#).

Add an about page

1. First, let's add the Flatpages App, which will allow us to add basic pages with HTML content.^[^1]

Add flatpages to the INSTALLED_APPS section in *settings.py*:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'main',
    'django.contrib.flatpages',
```

Update the DATABASES section in *settings.py*

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(SITE_ROOT,
'test.db'),
    }
}
```

Again, use absolute paths and name the database whatever you want.

Update the MIDDLEWARE_CLASSES of *settings.py*:

```
MIDDLEWARE_CLASSES = (
'django.middleware.common.CommonMiddleware',
'django.contrib.sessions.middleware.SessionMiddleware',
'django.middleware.csrf.CsrfViewMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',
)
```

Add the following pattern to *urls.py*:

```
('^pages/', include('django.contrib.flatpages.urls')),
```

Add a new template folder within the "templates" directory called "flatpages".

Then add a default template by creating a new file, *default.html*, within that new directory. Add the following code to the file:

```
{% extends 'base.html' %}

{% block content %}
{{ flatpage.content }}
{% endblock %}
```

2. Sync the database and create a superuser:

```
$ python manage.py syncdb
```

The `syncdb` command searches your `models.py` files and creates database tables for them.

3. Update the About url in the `base.html` file:

```
<li><a href="/pages/about">About</a></li>
```

4. Launch the server and navigate to <http://localhost:8000/admin/>, and then add the following page within the Flatpages section:

Change flat page

| | |
|---|--|
| URL: | /pages/about/ |
| Example: '/about/contact/'. Make sure to have leading and trailing slashes. | |
| Title: | About |
| Content: | <pre> <p>You can add some text about yourself here. Write as much as you want. Then when you are done, just add a closing HTML paragraph tag</p> Bullet Point # 1 Bullet Point # 2 Bullet Point # 3 Bullet Point # 4 <p>You can add a link or an email address here if you want. Again, you don't have too, but I highly recommend it. Cheers</p> <h3>Ready? Contact us!</h3></pre> |

The HTML within the content portion:

```
<br/>

<p>You can add some text about yourself here. Then when you
are done, just add a closing HTML paragraph tag.</p>

<ul>
    <li>Bullet Point # 1</li>
    <li>Bullet Point # 2</li>
    <li>Bullet Point # 3</li>
    <li>Bullet Point # 4</li>
</ul>
<br/>

<p>You can add a link or an email address <a
href="http://www.realpython.com">here</a> if you want.
Again, you don't have to, but I <strong>highly</strong>
recommend it. Cheers </em></p>

<h3>Ready? Contact us!</h3>
```

Now navigate to <http://localhost:8000/pages/about/> to view the new About page.

Nice, right? Next, let's add a Contact Us page.

11.5) Contact App

1. Create a new app:

```
$ python manage.py startapp contact
```

2. Add the new app to your *settings.py* file:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'main',
    'django.contrib.flatpages',
    'contact',
```

3. Setup a new model to create a database table:

```
from django.db import models
import datetime

class ContactForm(models.Model):
    name = models.CharField(max_length=150)
    email = models.EmailField(max_length=250)
    topic = models.CharField(max_length=200)
    message = models.CharField(max_length=1000)
    timestamp = models.DateTimeField(auto_now_add=True,
default=datetime.datetime.now)

    def __unicode__(self):
        return self.email

class Meta:
    ordering = ['-timestamp']
```

4. Sync the database:

```
$ python manage.py syncdb
```

5. Add a view:

```
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect, HttpResponseRedirect
from django.template import RequestContext, loader
from .forms import ContactView
from django.contrib import messages

def contact(request):
    form = ContactView(request.POST or None)
    if form.is_valid():
        our_form = form.save(commit=False)
        our_form.save()
        messages.add_message(request, messages.INFO, 'Your
message has been sent. Thank you.')
        return HttpResponseRedirect('/')
    t = loader.get_template('contact.html')
    c = RequestContext(request, {'form': form,})
    return HttpResponseRedirect(t.render(c))
```

6. Update *urls.py* with the following pattern:

```
url(r'^contact/', 'contact.views.contact', name='contact'),
```

7. Create a new file within the "contacts" directory called *forms.py*:

```
from django.forms import ModelForm
from .models import ContactForm
from django import forms

class ContactView(ModelForm):
    message = forms.CharField(widget=forms.Textarea)
    class Meta:
        model = ContactForm
```

8. Add another new file called *admin.py*:

```
from django.contrib import admin
from .models import ContactForm

class ContactFormAdmin(admin.ModelAdmin):
    class Meta:
        model = ContactForm

admin.site.register(ContactForm, ContactFormAdmin)
```

9. Add a new file to the templates directory called *contact.html*:

```
{% extends 'base.html' %}

{% block content %}

<h3><center>Contact Us</center></h3>

<form action='.' method='POST'>
    {% csrf_token %}

    {{ form.as_table }}
    <br/>
    <button type='submit' class='btn'>Submit</button>
</form>

{% endblock %}
```

The `{% csrf_token %}` is used for security purposes: to ensure that the specific request came from your project.

10. Update the Contact url in the *base.html* file:

```
<li><a href="{% url 'contact' %}">Contact</a></li>
```

Fire up the server and load your app. Click the link for "contact". Test it out, then make sure that data is being added to the table within the Admin page.

Did you notice the Flash message? See if you can recognize the code for it in the `views.py` file. You can read more about the messages framework [here](#), which provides support for Flask-like flash messages.

11.6) User Registration upon Payment

In this last section, we will be looking at how to implement a payment and user registration/authentication method. For this payment model, registration is tied to payments. In other words, in order to register, users must first make a payment.

We will be using [Stripe](#) for payment processing. Take a look at [this](#) brief tutorial on implementing Flask and Stripe to get a sense of how Stripe works.

1. Update the relative path to STATICFILES_DIRS in the `settings.py` file and add the "static" directory to the "project_name" directory:

```
STATICFILES_DIRS = (os.path.join(SITE_ROOT, 'static'),)
```

2. Create a new app:

```
$ python manage.py startapp payments
```

3. Add the new app to `settings.py`:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',
    'main',
    'django.contrib.flatpages',
    'contact',
    'payments',
```

4. Install stripe:

```
$ pip install stripe==1.9.2
```

5. Update *models.py*:

```
from django.db import models

class User(models.Model):
    name = models.CharField(max_length=255)
    email = models.CharField(max_length=255, unique=True)
    password = models.CharField(max_length=60)
    last_4_digits = models.CharField(max_length=4,
    blank=True, null=True)
    stripe_id = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.email
```

This model replaces the default User model.

6. Sync the database:

```
$ python manage.py syncdb
```

7. Add the following import, `from payments import views`, and patterns to *urls.py*:

```
url(r'^sign_in$', views.sign_in, name='sign_in'),
url(r'^sign_out$', views.sign_out, name='sign_out'),
url(r'^register$', views.register, name='register'),
url(r'^edit$', views.edit, name='edit'),
```

8. Add the remaining templates and all of the static files from the accompanying "files" folder from chapter 11, lesson 11.6.

9. Add a new file called *admin.py*:

```
from django.contrib import admin
from .models import User

class UserAdmin(admin.ModelAdmin):
    class Meta:
        model = User

    admin.site.register(User,UserAdmin)
```

10. Add another new file called *forms.py*:

```
from django import forms
from django.core.exceptions import NON_FIELD_ERRORS
from django.utils.translation import ugettext_lazy as _

class PaymentForm(forms.Form):
    def addError(self, message):
        self._errors[NON_FIELD_ERRORS] =
self.error_class([message])

class SigninForm(PaymentForm):
    email = forms.EmailField(required = True)
    password = forms.CharField(required = True,
widget=forms.PasswordInput(render_value=False))

class CardForm(PaymentForm):
    last_4_digits = forms.CharField(required = True,
min_length = 4, max_length = 4, widget =
forms.HiddenInput())
    stripe_token = forms.CharField(required = True, widget
= forms.HiddenInput())

class UserForm(CardForm):
    name = forms.CharField(required = True)
    email = forms.EmailField(required = True)
    password = forms.CharField(required = True,
label=(u'Password'),
widget=forms.PasswordInput(render_value=False))
    ver_password = forms.CharField(required = True,
label=(u' Verify Password'),
widget=forms.PasswordInput(render_value=False))

    def clean(self):
        cleaned_data = self.cleaned_data
        password = cleaned_data.get('password')
        ver_password = cleaned_data.get('ver_password')
        if password != ver_password:
            raise forms.ValidationError('Passwords do not
match')
        return cleaned_data
```

11. Update *payments/views.py*:

```
from django.db import IntegrityError
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response, redirect
from django.template import RequestContext
from payments.forms import PaymentForm, SigninForm,
CardForm, UserForm
from payments.models import User
import project_name.settings as settings
import stripe
import datetime

stripe.api_key = settings.STRIPE_SECRET

def soon():
    soon = datetime.date.today() +
datetime.timedelta(days=30)
    return {'month': soon.month, 'year': soon.year}

def sign_in(request):
    user = None
    if request.method == 'POST':
        form = SigninForm(request.POST)
        if form.is_valid():
            results =
User.objects.filter(email=form.cleaned_data['email'])
            if len(results) == 1:
                if
results[0].check_password(form.cleaned_data['password']):
                    request.session['user'] = results[0].pk
                    return HttpResponseRedirect('/')
                else:
                    form.addError('Incorrect email address or
password')
            else:
                form.addError('Incorrect email address or
password')
        else:
            form = SigninForm()
```

```
print form.non_field_errors()

return render_to_response(
    'sign_in.html',
    {
        'form': form,
        'user': user
    },
    context_instance=RequestContext(request)
)

def sign_out(request):
    del request.session['user']
    return HttpResponseRedirect('/')

def register(request):
    user = None
    if request.method == 'POST':
        form = UserForm(request.POST)
        if form.is_valid():

            #update based on your billing method
            #subscription vs one time
            customer = stripe.Customer.create(
                email = form.cleaned_data['email'],
                description = form.cleaned_data['name'],
                card = form.cleaned_data['stripe_token'],
                plan="gold",
            )
            # customer = stripe.Charge.create(
            #     description = form.cleaned_data['email'],
            #     card = form.cleaned_data['stripe_token'],
            #     amount="5000",
            #     currency="usd"
            # )

            user = User(
                name = form.cleaned_data['name'],
                email = form.cleaned_data['email'],
```

```
        last_4_digits =
form.cleaned_data['last_4_digits'],
                stripe_id = customer.id,
                password = form.cleaned_data['password']
            )

        #
user.set_password(form.cleaned_data['password'])

        try:
            user.save()
        except IntegrityError:
            form.addError(user.email + ' is already a
member')
        else:
            request.session['user'] = user.pk
            return HttpResponseRedirect('/')

    else:
        form = UserForm()

    return render_to_response(
        'register.html',
        {
            'form': form,
            'months': range(1, 12),
            'publishable': settings.STRIPE_PUBLISHABLE,
            'soon': soon(),
            'user': user,
            'years': range(2011, 2036),
        },
        context_instance=RequestContext(request)
    )

def edit(request):
    uid = request.session.get('user')

    if uid is None:
        return HttpResponseRedirect('/')
```

```
user = User.objects.get(pk=uid)

if request.method == 'POST':
    form = CardForm(request.POST)
    if form.is_valid():

        customer =
stripe.Customer.retrieve(user.stripe_id)
        customer.card = form.cleaned_data['stripe_token']
        customer.save()

        user.last_4_digits =
form.cleaned_data['last_4_digits']
        user.stripe_id = customer.id
        user.save()

    return HttpResponseRedirect('/')

else:
    form = CardForm()

return render_to_response(
    'edit.html',
    {
        'form': form,
        'publishable': settings.STRIPE_PUBLISHABLE,
        'soon': soon(),
        'months': range(1, 12),
        'years': range(2011, 2036)
    },
    context_instance=RequestContext(request)
)
```

You can charge users either a one time charge or a recurring charge. This script is setup for the latter. To change to a one time charge, simply make the following changes to the file:

```
#update based on your billing method (subscription vs one
#time)
#customer = stripe.Customer.create(
#email = form.cleaned_data['email'],
#description = form.cleaned_data['name'],
#card = form.cleaned_data['stripe_token'],
#plan="gold",
#)
customer = stripe.Charge.create(
description = form.cleaned_data['email'],
card = form.cleaned_data['stripe_token'],
amount="5000",
currency="usd"
)
```

12. Update *main/views.py*:

```
from django.shortcuts import render_to_response
from django.template import RequestContext
from payments.models import Use

def index(request):
    uid = request.session.get('user')
    if uid is None:
        return render_to_response('index.html')
    else:
        return render_to_response('user.html', {'user':
User.objects.get(pk=uid)})
```

13. Update the *base.html* template:

```
{% load static %}

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <title>Your MVP</title>
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <meta name="description" content="Setting up an MVP with
Django by RealPython.com">
    <meta name="author" content="Michael Herman">
    <!-- Le styles -->
    <link href="http://www.realpython.com/css/bootstrap.css"
rel="stylesheet">
    <style>

        body {
            padding-top: 20px;
            padding-bottom: 40px;
        }

        /* Custom container */
        .container-narrow {
            margin: 0 auto;
            max-width: 700px;
        }
        .container-narrow > hr {
            margin: 30px 0;
        }

        /* Main marketing message and sign up button */
        .jumbotron {
            margin: 60px 0;
            text-align: center;
        }
        .jumbotron h1 {
            font-size: 72px;
```

```
        line-height: 1;
    }
.jumbotron .btn {
    font-size: 21px;
    padding: 14px 24px;
}

/* Supporting marketing content */
.marketing {
    margin: 60px 0;
}
.marketing p + h4 {
    margin-top: 28px;
}
</style>
<script src="https://js.stripe.com/v1/" type="text/javascript"></script>
<script type="text/javascript">
//<! [CDATA[
Stripe.publishableKey = '{{ publishable }}';
//]]>
</script>
<script src="{% get_static_prefix %}jquery.js"
type="text/javascript"></script>
<script src="{% get_static_prefix %}application.js"
type="text/javascript"></script>
</style>
</head>
<body>

<div class="container-narrow">
    <div class="masthead">
        <ul class="nav nav-pills pull-right">
            <li><a href="{% url 'home' %}">Home</a></li>
            <li><a href="/pages/about">About</a></li>
            <li><a href="{% url 'contact' %}">Contact</a></li>
            {% if user %}
                <li><a href="{% url 'sign_out' %}">Logout</a></li>
```

```
{% else %}
    <li><a href="{% url 'sign_in' %}">Login</a></li>
    <li><a href="{% url 'register'
%}">Register</a></li>
    {% endif %}
</ul>
<h3><span class="fui-settings-16 muted">Your
MVP!</span></h3>
</div>

<hr>

{% if messages %}
<div class="alert alert-success">
    <div class="messages">
        {% for message in messages %}
            {{ message }}
        {% endfor %}
    </div>
</div>
{% endif %}

<div class="container-narrow">
    {% block content %}
    {% endblock %}
</div>

<hr>

<div class="footer">
    <p>Copyright © 2013 <a
href="http://www.yoursite.com">Your MVP</a>. Developed by
<a href="http://www.mherman.org">Your Name Here</a>.
Powered by Django.</p></p>
</div>

</div> <!-- /container -->
```

14. Update the large button on *index.html*:

```
<a class="btn btn-large btn-success"  
href="/register">Register today to get  
started!</a><br/><br/><br/>
```

15. Add the following stripe keys to *settings.py*:

```
STRIPE_SECRET = 'sk_test_h36oicOrlA7ATkI9JJ6dUGyA'  
STRIPE_PUBLISHABLE = 'pk_test_8Xho4FfArFFuQspdH8V1KlHS'
```

Go ahead and test this out using the test keys. Use the following credit card number, 4242424242424242, and any 3 digits for the CVC code. Use any future date for the expiration date. Make sure you have a subscription plan in place. Run the following script to set one up:

```
import stripe  
stripe.api_key = "sk_test_zl9IWWHkI2JUvGnaLyiSSmxH"  
  
stripe.Plan.create(  
    amount=2000,  
    interval='month',  
    name='Amazing Gold Plan',  
    currency='usd',  
    id='gold')
```

Name

Email

Password

Verify Password

Credit card number

Security code (CVC)

Expiration date

Finally, after you process the dummy payment, make sure the user is added to the database as well as the [dashboard](#) on Stripe:

The screenshot shows the SQLite Database Browser interface. The title bar reads "SQLite Database Browser - /Users/michaelherman/Desktop/realpython/mvp/project_name/te...". The toolbar contains various icons for database management. Below the toolbar, there are three tabs: "Database Structure", "Browse Data" (which is selected), and "Execute SQL". A sub-header "Table: payments_user" is followed by "New Record" and "Delete Record" buttons. The main area displays a table with the following data:

| | <u>id</u> | <u>name</u> | <u>email</u> | <u>last_4_digits</u> | <u>stripe_id</u> | <u>created_at</u> | <u>updated_at</u> |
|---|-----------|----------------|---------------|----------------------|--------------------|---------------------|---------------------|
| 1 | 1 | Michael Herman | m@testing.com | 4242 | cus_2HiU1uq4XAEJT8 | 2013-07-29 05:12:00 | 2013-07-29 05:12:00 |
| 2 | 2 | John Eskey | jack@ryan.com | 4242 | cus_2Hj3mQGHDU0Kdb | 2013-07-29 05:52:00 | 2013-07-29 05:52:00 |

At the bottom, there are navigation buttons for pages 1-2 of 2, a "Go to:" input field with value 0, and a scroll bar.

The screenshot shows the Stripe dashboard interface. On the left, a sidebar menu includes sections for GENERAL (Dashboard, Payments, Customers), SUBSCRIPTIONS (Plans, Coupons), and REQUESTS (Events & Webhooks, Logs). The main content area displays customer details for 'jack@ryan.com' (cus_2Hj3mQGHDU0Kdb). It shows the customer's ID, creation date (Jul 29, 2013), email (jack@ryan.com), and description (John Eskey). Below this, there's a 'Cards' section showing a single card entry: Visa, last four digits 4242, expiration 8/2013, with options to Add Card, Edit, or Delete. Under 'Payments', a recent transaction is listed: \$20.00 - ch_2Hj3OYzujyZzXW, dated 2013/07/29 05:54:18. At the bottom, the 'Subscription' section indicates an active plan: Amazing Gold Plan (\$20.00/month).

Refer to the Stripe [documentation](#) and [API reference docs](#) for more info.

-
1. <http://docs.python.org/2/library/os.path.html>

Appendix A: Installing Python

Windows 7

Start by downloading Python 2.7.4 from the official Python [website](#). The Windows version is distributed as a MSI package. Once downloaded, double-click the file to install. By default this will install Python to C:\Python2.7.

You also need to add Python to your PATH environmental variables, so when you want to run a Python script, you do not have to type the full path each and every time, as this is quite tedious.

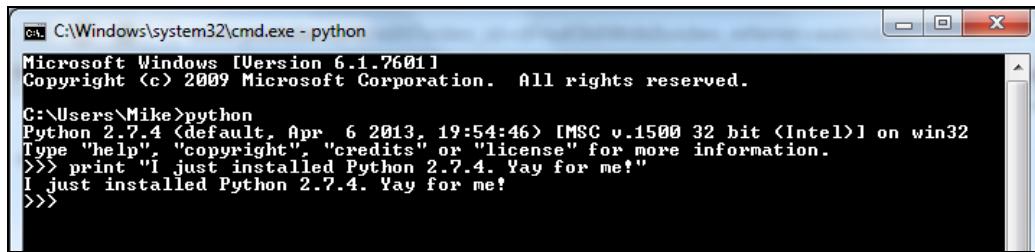
Since you downloaded Python version 2.7.4, you need to add the following directories to your PATH:

- C:\Python27\
- C:\Python27\Scripts\
- C:\PYTHON27\DLLs\
- C:\PYTHON27\LIB\

Open your powershell and run the following statement:

```
[Environment]::SetEnvironmentVariable("Path",
"$env:Path;C:\Python27\;C:\Python27\Scripts\;C:\PYTHON27\
DLLs\;C:\PYTHON27\LIB\", "User")
```

That's it. To test to make sure Python was installed correctly open your command prompt and then type `python` to load the Shell:



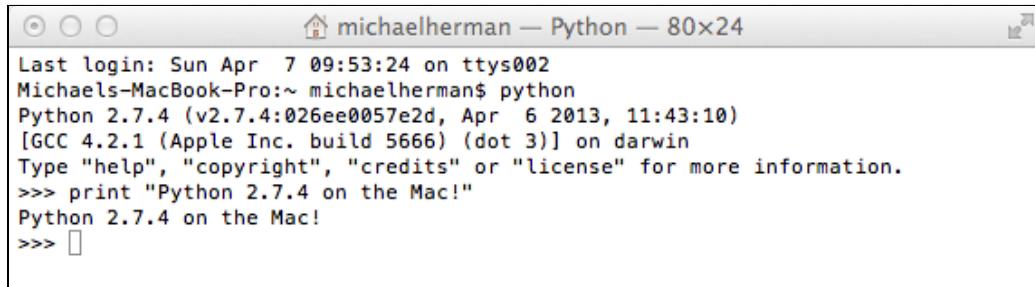
cmd C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright © 2009 Microsoft Corporation. All rights reserved.
C:\Users\Mike>python
Python 2.7.4 (default, Apr 6 2013, 19:54:46) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print "I just installed Python 2.7.4. Yay for me!"
I just installed Python 2.7.4. Yay for me!
>>>

Video

Watch the video [here](#) for assistance.

Mac OS X

All Mac OS X versions since 10.4 come with Python pre-installed. You can view the version by opening the terminal and typing `python` to enter the shell. The output will look like this:



michaelherman — Python — 80x24
Last login: Sun Apr 7 09:53:24 on ttys002
Michaels-MacBook-Pro:~ michaelherman\$ python
Python 2.7.4 (v2.7.4:026ee0057e2d, Apr 6 2013, 11:43:10)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Python 2.7.4 on the Mac!"
Python 2.7.4 on the Mac!
>>> █

If you don't currently have version 2.7.4, go ahead and upgrade by downloading the latest [installer](#).

Once downloaded, double-click the file to install.

Linux

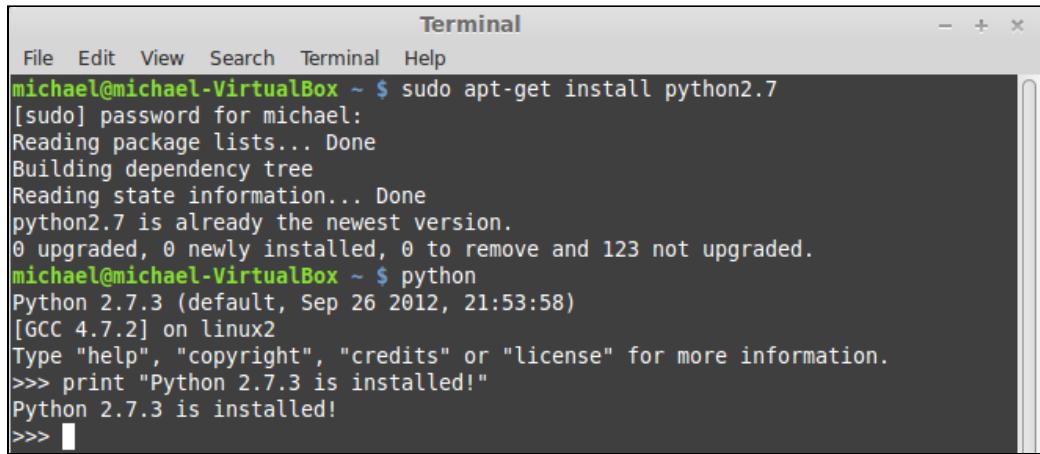
If you are using Ubuntu, Linux Mint, or another Debian-based system, enter the following command in your terminal:

```
sudo apt-get install python2.7
```

Or you can download the tarball directly from the official Python [website](#). Once downloaded, run the following commands:

```
$ tar -zvxf [mytarball.tar.gz]
$ ./configure
$ make
$ sudo make install
```

Once installed, fire up the terminal and type `python` to get to the shell:



The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area of the terminal shows the following command-line session:

```
michael@michael-VirtualBox ~ $ sudo apt-get install python2.7
[sudo] password for michael:
Reading package lists... Done
Building dependency tree
Reading state information... Done
python2.7 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 123 not upgraded.
michael@michael-VirtualBox ~ $ python
Python 2.7.3 (default, Sep 26 2012, 21:53:58)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Python 2.7.3 is installed!"
Python 2.7.3 is installed!
>>> 
```

Please note, as of April 7th, 2013, Python 2.7.4 must be downloaded via tarball for Linux. If you have problems just use the Package Manager to install 2.7.3.

If you have problems or have a different Linux distribution, you can always use your package manager or just do a Google search for how to install Python on your particular Linux distribution.

Appendix B: Supplementary Materials

1) Working with FTP

There are a number of different means of exchanging files over the Internet. One of the more popular ways is to connect to a FTP server to download and upload files.

FTP (File Transfer Protocol) is used for file exchange over the Internet. Much like HTTP and SMTP, which are used for exchanging web pages and email across the Internet respectively, FTP uses the TCP/IP protocols for transferring data.

In most cases, FTP is used to either upload a file (such as a web page) to a remote server or download a file from a server. In this lesson, we will be accessing an FTP server to view the main directory listing, upload a file, and then download a file.

Code:

```
# ex_appendixB.1a.py - FTP Directory Listing

import ftplib

server = ''
username = ''
password = ''

# Initialize and pass in FTP URL and login credentials (if
applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

# Create a list to receive the data
data = []

# Append the directories and files to the list
ftp.dir(data.append)

# Close the connection
ftp.quit()

# Print out the directories and files, line by line
for l in data:
    print(l)
```

Save the file as *ex_appendixB.1a.py*. **DO NOT** run it just yet.

What's going on here?

So, we imported the `ftplib` library, which provides Python all dependencies it needs to access remote servers, files, and directories. We then defined variables for the remote server and the login credentials to initialize the FTP connection. You can leave the username and password empty if you are connecting to an anonymous FTP server (which usually doesn't require a username or password). Next, we created a list to receive the directory listing, and used the `dir` command to append data to the list. Finally we disconnected from the server and then printed the directories, line by line, using a for loop.

Let's test this out with a public ftp site, using these server and login credentials:

- `server = 'ftp.secureftp-test.com'`
- `username = 'test'`
- `password = 'test'`

Keep everything else the same, and save the file again. Now you can run it.

If done correctly your output should look like this:

```
-r--r--r-- 1 ftp ftp          1471 Jun 25 2007 bookstore.xml
-r--r--r-- 1 ftp ftp          279658 Jun 25 2007 hamlet.xml
-r--r--r-- 1 ftp ftp          80641 Jun 25 2007 hamlet.zip
-r--r--r-- 1 ftp ftp          2164 Jun 25 2007 japanese.xml
-r--r--r-- 1 ftp ftp          0 Jul 03 2007 New Text
Document (2).txt
-r--r--r-- 1 ftp ftp          0 Jul 03 2007 New Text
Document.txt
-r--r--r-- 1 ftp ftp          4677 Jun 25 2007 nutrition.xml
-r--r--r-- 1 ftp ftp          8463 Jun 25 2007 pigs.xml
-r--r--r-- 1 ftp ftp          7581 Jun 25 2007 plants.xml
drwxr-xr-x 1 ftp ftp          0 Jun 25 2007 subdir1
drwxr-xr-x 1 ftp ftp          0 Jun 25 2007 subdir2
```

Next, let's take a look at how to download a file from a FTP server.

Code:

```
# ex_appendixB.1b.py - FTP File Download

import ftplib
import sys

server = 'ftp.secureftp-test.com'
username = 'test'
password = 'test'

# Defines the name of the file for download
file_name = sys.argv[1]

# Initialize and pass in FTP URL and login credentials (if
# applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

# Create a local file with the same name as the remote file
with open(file_name, "wb") as f:

    # Write the contents of the remote file to the local file
    ftp.retrbinary("RETR " + file_name, f.write)

# Closes the connection
ftp.quit()
```

When you run the file, make sure you specify a file name from the remote server on the command line. You can use any of the files you saw in the above script when we outputted them to the screen.

For example:

```
python ex_appendixB.1b.py bookstore.xml
```

Finally, let's take a look at how to upload a file to a FTP Server.

Code:

```
# ex_appendixB.1c.py - FTP File Upload

import ftplib
import sys

server = ''
username = ''
password = ''

# Defines the name of the file for upload
file_name = sys.argv[1]

# Initialize and pass in FTP URL and login credentials (if
# applicable)
ftp = ftplib.FTP(host=server, user=username, passwd=password)

# Opens the local file to upload
with open(file_name, "rb") as f:

    # Write the contents of the local file to the remote file
    ftp.storbinary("STOR " + file_name, f)

# Closes the connection
ftp.quit()
```

Save the file as `ex_appendixB.1c.py`. **DO NOT** run it.

Unfortunately, the public FTP site we have been using does not allow uploads. You will have to use your own server to test. Many free hosting services offer FTP access. You can set one up in less than fifteen minutes. Just search Google for "free hosting with ftp" to find a free hosting service. One good example is

www.0fees.net

After you setup your FTP Server, update the server name and login credentials in the above script, save the file, and then run it. *Again, specify the filename as one of the command line arguments. It should be any file found on your local directory.*

For example:

```
python ex_appendixB.1c.py uploadme.txt
```

Check to ensure that the file has been uploaded to the remote directory

It's best to send files in binary mode "rb" as this mode sends the raw bytes of the file. Thus, the file is transferred in its exact original form.

Homework

- See if you figure out how to navigate to a specific directory, upload a file, and then run a directory listing of that directory to see the newly uploaded file. Do this all in one script.

2) Working with SFTP

SFTP (Secure File Transfer Protocol) is used for securely exchanging files over the Internet. From an end-user's view it is very much like FTP, except that data is transported over a secure channel.

To use SFTP you have to install the pysftp library. There are other custom libraries for SFTP but this one is good if you want to run SFTP commands with minimal configuration.

Go ahead and install pysftp:

```
`pip install pysftp`
```

Now, let's try to list the directory contents from a remote server via SFTP.

Code:

```
# ex_appendixB.2a.py - SFTP Directory Listing

import pysftp

server = ''
username = ''
password = ''

# Initialize and pass in SFTP URL and login credentials (if
# applicable)
sftp = pysftp.Connection(host=server, username=username,
password=password)

# Get the directory and file listing
data = sftp.listdir()

# Closes the connection
sftp.close()

# Prints out the directories and files, line by line
for l in data:
    print(l)
```

Save the file as *ex_appendixB.2a.py*. Once again, **DO NOT** run it.

To test the code, you will have to use your own remote server that supports SFTP. Unfortunately, most of the free hosting services does not offer SFTP access. But don't worry, there are free sites that offer *free shell accounts* which normally include SFTP access. Just search Google for "free shell accounts". One good example is www.cjb.net

After you setup your SFTP Server, update the server name and login credentials in the above script, save the file, and then run it.

If done correctly, all the files and directories of the current directory of your remote server will be displayed.

Next, let's take a look at how to download a file from an SFTP server.

Code:

```
# ex_appendixB.2b.py - SFTP File Download

import pysftp
import sys

server = ''
username = ''
password = ''

# Defines the name of the file for download
file_name = sys.argv[1]

# Initialize and pass in SFTP URL and login credentials (if
# applicable)
sftp = pysftp.Connection(host=server, username=username,
password=password)

# Download the file from the remote server
sftp.get(file_name)

# Closes the connection
sftp.close()
```

When you run it, make sure you specify a file name from the remote server on the command line. You can use any of the files you saw in the above script when we outputted them to the screen.

For example:

```
python ex_appendixB.2b.py remotefile.csv
```

Finally, let's take a look at how to upload a file to an SFTP Server.

Code:

```
# ex_appendixB.2c.py - SFTP File Upload

import pysftp
import sys

server = 'shell.cjb.net'
username = 'tstssh007'
password = 'p@sssh5ll'

# Defines the name of the file for upload
file_name = sys.argv[1]

# Initialize and pass in SFTP URL and login credentials (if
# applicable)
sftp = pysftp.Connection(host=server, username=username,
password=password)

# Upload the file to the remote server
sftp.put(file_name)

# Closes the connection
sftp.close()
```

When you run, make sure you specify a file name from your local directory on the command line.

For example:

```
python ex_appendixB.2c.py uploadme.txt
```

Check to ensure that the file has been uploaded to the remote directory

3) Sending and Receiving Email

SMTP (Simple Mail Transfer Protocol) is the protocol that handles sending and routing email between mail servers. The `smtplib` module provided by Python is used to define the SMTP client session object implementation, which is used to

send mail through Unix mail servers. MIME is an Internet standard used for building the various parts of emails, such as "From", "To", "Subject", and so forth. We will be using that library as well.

Sending mail is simple.

Code:

```
# ex_appendixB.3a.py - Sending Email via SMTP (part 1)

import smtplib
from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText

# inputs for from, to, subject and body text
fromaddr = raw_input("Sender's email: ")
toaddr = raw_input('To: ')
sub = raw_input('Subject: ')
text = raw_input('Body: ')

# email account info from where we'll be sending the email from
smtp_host = 'smtp.mail.com'
smtp_port = '###'
user = 'username'
password = 'password'

# parts of the actual email
msg = MIMEMultipart()
msg['From'] = fromaddr
msg['To'] = toaddr
msg['Subject'] = sub
msg.attach(MIMEText(text))

# connect to the server
server = smtplib.SMTP()
server.connect(smtp_host, smtp_port)

# initiate communication with server
server.ehlo()

# use encryption
server.starttls()

# login to the server
server.login(user, password)
```

```
# send the email
server.sendmail(fromaddr, toaddr, msg.as_string())

# close the connection
server.quit()
```

Save as `ex_appendixB.3a.py`. DO NOT run.

Since this code is pretty self-explanatory (follow along with the comments), go ahead and update the the following variables: `smtp_host`, `smtp_port`, `user`, `password` to match your email account's SMTP info and login credentials you wish to send from.

Example:

```
# email account info from where we'll be sending the email from
smtp_host = 'smtp.gmail.com'
smtp_port = 587
user = 'hermanmu@gmail.com'
password = "it's a secret - sorry"
```

I suggest using a GMail account and sending and receiving from the same address at first to test it out, and then try sending from Gmail to a different email account, on a different email service. Once complete, run the file. As long as you don't get an error, the email should have sent correctly. Check your email to make sure.

Before moving on, let's clean up the code a bit:

```
# ex_appendixB.3b.py - Sending Email via SMTP (part 2)

import smtplib
from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText

def mail(fromaddr, toaddr, sub, text, smtp_host, smtp_port,
        user, password):

    # parts of the actual email
    msg = MIMEMultipart()
    msg['From'] = fromaddr
    msg['To'] = toaddr
    msg['Subject'] = sub
    msg.attach(MIMEText(text))

    # connect to the server
    server = smtplib.SMTP()
    server.connect(smtp_host,smtp_port)

    # initiate communication with server
    server.ehlo()

    # use encryption
    server.starttls()

    # login to the server
    server.login(user, password)

    # send the email
    server.sendmail(fromaddr, toaddr, msg.as_string())

    server.quit()

if __name__ == '__main__':
    fromaddr = 'hermanmu@gmail.com'
    toaddr = 'hermanmu@gmail.com'
    subject = 'test'
```

```
body_text = 'hear me?'
smtp_host = 'smtp.gmail.com'
smtp_port = '587'
user = 'hermanmu@gmail.com'
password = "it's a secret - sorry"

mail(fromaddr, toaddr, subject, body_text, smtp_host,
smtp_port, user, password)
```

Meanwhile, IMAP (Internet Message Access Protocol) is the Internet standard for receiving email on a remote mail server. Python provides the `imaplib` module as part of the standard library which is used to define the IMAP client session implementation, used for accessing email. Essentially, we will be setting up our own mail server.

Code:

```
# ex_appendixB.3c.py - Receiving Email via IMAPLIB

import imaplib

# email account info from where we'll be sending the email from
imap_host = 'imap.gmail.com'
imap_port = '993'
user = 'hermanmu@gmail.com'
password = "It's a secret - sorry!"

# login to the mail server
server = imaplib.IMAP4_SSL(imap_host, imap_port)
server.login(user, password)

# select the inbox
status, num = server.select('Inbox')

#fetch the email and the information you wish to display
status, data = server.fetch(num[0], '(BODY[TEXT])')

# print the results
print data[0][1]
server.close()
server.logout()
```

Notice how I used the same GMail account as in the last example. Make sure you tailor this to your own account settings. Essentially, this code is used to read the most recent email in the Inbox. This just so happened to be the email I sent myself in the last example.

If done correctly, you should see this:

```
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Test Python - What up!
```

"Test Python - What Up!" is the body of my email.

Also, in the program `num[0]` specifies the message we wish to view, while
`(BODY [TEXT])` displays the information from the email.

Homework

- See if you can figure out how to use a for loop to read the first 10 messages in your inbox.