



Real Python Part 2: Web Development with Python

Michael Herman

Contents

1	Preface	10
	Acknowledgements	10
	Thank you	11
	About the Author	12
	About the Editor	13
	License	14
2	Introduction	15
	Why Python?	16
	Who should take this Course?	17
	How to use this Course	18
	Conventions	19
	Errata	21
3	Getting Started	22
	Python Review	22
	Development Environments	23
	Installing SQLite	25
	Installing easy_install and pip	26
	Installing virtualenv	28
	Web Browsers	30
	Version Control	32
4	Interlude: Modern Web Development	36
	Overview	36
	Front-end, Back-end, and Middleware	37
	Model-View-Controller (MVC)	38

5	Flask: QuickStart	40
	Overview	41
	Installation	42
	Hello World	43
	Dynamic Routes	45
	Debug Mode	46
6	Interlude: Database Programming	48
	SQL and SQLite Basics	49
	Creating Tables	51
	Inserting Data	54
	Searching	58
	Updating and Deleting	59
	Working with Multiple Tables	61
	SQL Functions	65
	Example Application	67
7	Flask Blog App	70
	Project Structure	71
	Model	73
	Controller	74
	Views	75
	Templates	77
	Run the server!	79
	User Login	80
	Sessions and Login_required Decorator	83
	Show Posts	88
	Add Posts	89
	Style	91
	Conclusion	93
8	Interlude: Debugging in Python	94
	Workflow	95
	Post Mortem Debugging	101

9 Flask: FlaskTaskr	103
Overview	103
Initial Setup and Configuration	104
Extensions	117
Database Management via SQLAlchemy	118
User Registration	124
User Login	129
Database Relationships	131
Managing Sessions	139
Error Handling	141
Unit Testing	145
10 Interlude: Introduction to HTML and CSS	150
HTML	151
CSS	154
Chrome Developer Tools	157
11 Flask: FlaskTaskr (continued...)	158
Templates and Styling	158
Blueprints	164
Test Coverage	176
Logging	178
Deploying on Heroku	179
Fabric	181
Building a REST API	184
Boilerplate Template and Workflow	187
12 Interlude: Web Frameworks, Compared	190
Overview	190
Popular Frameworks	192
Components	193
What does this all mean?	194

13 web2py: QuickStart	195
Overview	195
Installation	197
Hello World	198
Deploying on PythonAnywhere	202
seconds2minutes App	203
14 Interlude: APIs	205
Introduction	205
Retrieving Web Pages	207
Web Services Defined	210
Working with XML	213
Working with JSON	216
Working with Web Services	220
Rotten Tomatoes API	229
15 web2py: QuickStart (continued...)	232
Sentiment Analysis	232
Sentiment Analysis Expanded	243
Movie Suggestor	253
Blog App	261
16 web2py: py2manager	264
Introduction	264
Setup	266
Database	268
URL Routing	275
Initial Views	277
Profile Page	280
Add Projects	282
Add Companies	284
Homepage	285
More Grids	287

Notes	289
Error Handling	291
Final Word	292
17 Interlude: Web Scraping and Crawling	293
HackerNews (BaseSpider)	296
Scrapy Shell	299
Wikipedia (BaseSpider)	301
Socrata (CrawlSpider and Item Pipeline)	304
Web Interaction	309
18 web2py: REST Redux	312
Introduction	312
Basic REST	313
Advanced REST	317
19 Django: Quickstart	322
Overview	322
Installation	324
Hello, World!	325
20 Interlude: Introduction to Javascript and jQuery	334
Handling the Event	335
Append the text to the DOM	337
Remove text from the DOM	338
21 Bloggy: A blog app	339
Model	340
Django Shell	344
Unit Tests for models	348
Django Admin	349
Custom Admin View	351
Templates and Views	352
Friendly Views	357

South	359
Styles	362
Update the Index View	365
Forms	368
22 Django: Ecommerce Site	376
Overview	376
Rapid Web Development	377
Prototyping	378
Setup your Project and App	379
Contact App	390
User Registration upon Payment	393
23 Appendix A: Installing Python	406
Windows 7	406
Mac OS X	408
Linux	409
24 Appendix B: Supplementary Materials	410
Working with FTP	410
Working with SFTP	414
Sending and Receiving Email	417

List of Figures

3.1	sqlite browser	25
3.2	firebug	31
4.1	model-view-controller	38
5.1	flask	40
6.1	sqlite	52
7.1	flask mvc	71
7.2	flask templates	78
7.3	session_token	86
7.4	flask blog	92
9.1	flasktaskr	112
9.2	flasktaskr main	116
9.3	flasktaskr sql	123
9.4	flask new table	126
9.5	sql relationships	131
9.6	erd schema	132
9.7	updated schema	135
9.8	one user	138
11.1	coverage report	176
13.1	web2py	195
13.2	web2py sites	198
13.3	web2py welcome	199

13.4 web2py generic view	201
13.5 pythonanywhere	202
13.6 web2py error	204
14.1 chrome dev tools	211
14.2 driving api	226
14.3 directions	227
14.4 json parsing	228
14.5 rotten tomatoes api	230
15.1 hurl	234
15.2 requests pulse	235
15.3 sentiment app 1	238
15.4 sentiment app 2	239
15.5 sentiment app 3	240
15.6 sentiment app 4	241
15.7 sentiment app 5	245
15.8 sentiment app 6	246
15.9 sentiment app 7	248
15.10sentiment app 8.png	252
15.11 suggest movie	256
15.12suggest movie again	258
15.13suggest movie final	259
16.1 sublime	267
16.2 web2py shell	270
16.3 profile pic	281
16.4 grid links	290
17.1 ebay	294
17.2 robots.txt	295
17.3 hackernews	298
17.4 yahoo finance	310
19.1 django logo	322

19.2 dev server	326
21.1 syncdb	341
21.2 sqlite browser django	347
21.3 django admin	350
21.4 bloggy basic view	353
21.5 regex test	355
21.6 bloggy_form	372
21.7 bloggy_form_2	374
22.1 mvp part 1	386
22.2 django flat page	389
22.3 register	403
22.4 stripe database	404
22.5 stripe	405
23.1 windows install	407
23.2 mac install	408
23.3 linux install	409

Chapter 1

Preface

Acknowledgements

Writing is an intense, solitary activity that requires discipline and repetition. Although much of what happens in that process is still 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 and Jeremy, authors of the other Real Python courses, for believing in me even when I did not believe in myself. They both are talented developers and natural leaders; I'm also proud to call them friends. Thanks to all my close friends and family (Mom, Dad, Jeff) for all your support and kindness. Derek, Josh, Danielle, Richard, 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. He's been hacking since developing his first project - a video game enthusiast website - back in 1999.

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.

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.

Chapter 2

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 it's 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. Participate in a hackathon. 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/algorithm 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.

NOTE: If you do decide to challenge yourself by finding and correcting code errors or areas that are unclear or lack clarity, please contact me at info@realpython.com so I can update the course. Any feedback is appreciated. This will benefit other readers, and you will receive credit.

Why Python?

Python is a beautiful language. It's easy to learn and fun, and its syntax (rules) 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.

As you found out in the previous course, Python is used in a number disciplines, making it extremely versatile. Such disciplines include:

1. System Administration,
2. 3D Animation and Image Editing,
3. Scientific Computing/Data Analysis,
4. Game Development, and
5. Web Development.

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 a project 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.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

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

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

How to use this Course

This course has roughly three sections: *Client-Side Programming*, *Server-Side Programming*, and *Web Development*. Although each is intended to be modular, the chapters within the sections build on one another - so working in order is recommended. 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!

Conventions

NOTE: Since this is the Alpha release, we do not have all the conventions in place. We are working on it. Patience.

Formatting

1. Code blocks will be used to present example code.

```
1 print "Hello world!"
```

2. Terminal commands follow the Unix format:

```
1 $ python hello-world.py
```

(dollar signs are not part of the command)

3. *Italic text* will be used to denote a file name:

hello-world.py.

Bold text will be used to denote a new or important term:

Important term: This is an example of what an important term should look like.

NOTES, WARNINGS, and SEE ALSO boxes appear as follows:

1. Notes:

NOTE: This is a note filled in with bacon ipsum text. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

2. Warnings:

WARNING: This is a warning also filled in with bacon ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa

pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

3. See Also:

SEE ALSO: This is a see also box with more tasty ipsum. Bacon ipsum dolor sit amet t-bone flank sirloin, shankle salami swine drumstick capicola doner porchetta bresaola short loin. Rump ham hock bresaola chuck flank. Prosciutto beef ribs kielbasa pork belly chicken tri-tip pork t-bone hamburger bresaola meatball. Prosciutto pork belly tri-tip pancetta spare ribs salami, porchetta strip steak rump beef filet mignon turducken tail pork chop. Shankle turducken spare ribs jerky ribeye.

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](#). Or submit an issue on the Real Python official support [repository](#). Thank you!

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

Chapter 3

Getting Started

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 2.7.6 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.x 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
- Operators
- Tuples
- Dictionaries
- Loops
- Functions
- Modules
- Booleans

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

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 [Sublime Text](#). If you are already familiar with these environments, and have Sublime installed, 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	cd	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	rm	delete a file
md	mkdir	create a new directory
rd	rmdir	remove a directory

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

```
1 $ python big_snake.py
```

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

Windows equivalent:

```
1 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:


```
1 $ python
2 >>> phrase = "The bigger the snake, the bigger the prey"
3 >>> print phrase
4 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. You can also just type `exit()` then press enter:

```
1 >>> exit()
2 $
```

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 a text editor called Sublime Text.

Sublime Text

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

You can download the latest versions for Windows and Unix [here](#). It's free to try for as long as you like, although it will bug you (about once a day) to buy a license.

There are plenty of other free text editors that you can use such as Notepad++ for Windows, TextWrangler for Mac, and the excellent, cross-platform editor gedit. You can use any of these, however all examples in this course will be completed in Sublime Text.

Never use a word processor like Microsoft Word as your text editor, because text editors, unlike word processors, just display raw, plain text.

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.

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:

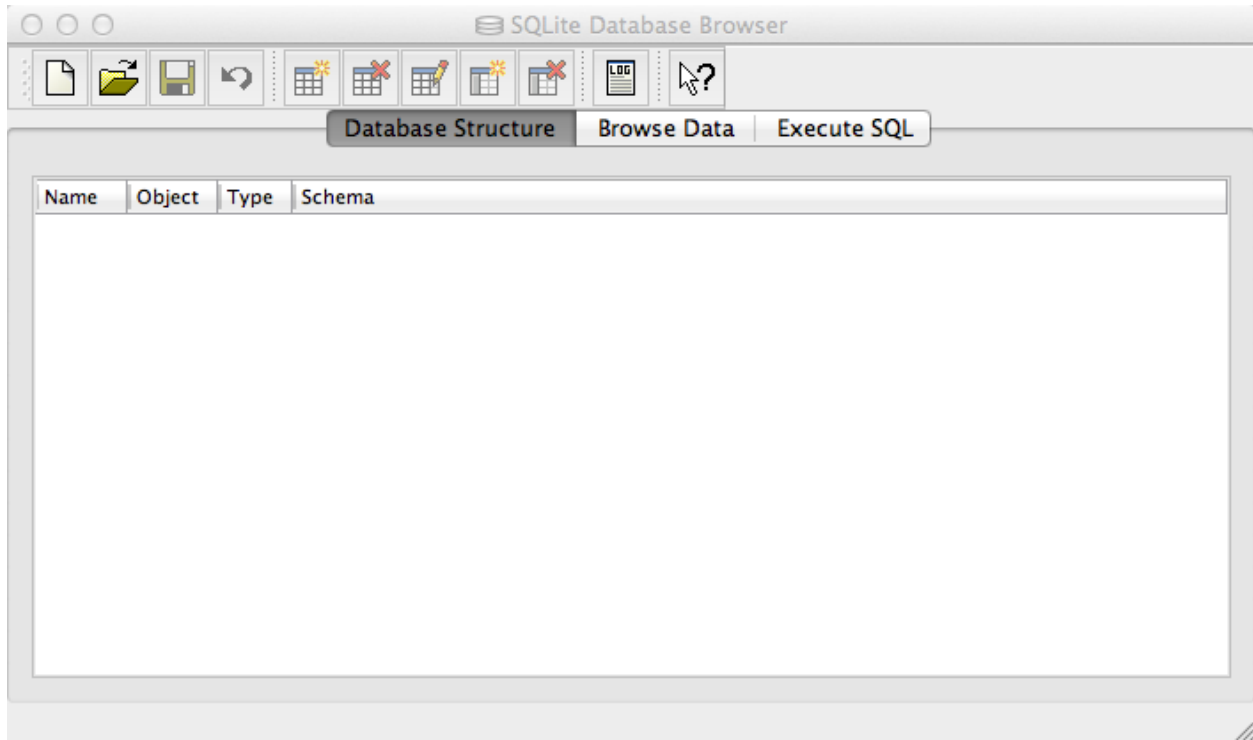


Figure 3.1: sqlite browser

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](#).

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. For example, if you wanted to work with a third party API like Twitter or Google Maps, you can save a lot of time by using a pre-built package, rather than building it yourself from scratch.

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](https://pypi.org/) (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 they are done downloading, pip waits until downloads are complete. So, if your connection is lost during download, you won’t have a partially installed package.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

Now, since pip is a wrapper that relies on easy_install/Setuptools, 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:

```
1 $ easy_install pip
```

To install a package:

```
1 $ pip install numpy
```

To download a specific version:

```
1 $ pip install numpy==1.6.1
```

To uninstall a package:

```
1 $ pip uninstall numpy
```

NOTE: 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.

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

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 *absolutely* essential 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:

```
1 $ virtualenv --version
```

If installed, the version number will be returned:

```
1 $ virtualenv --version
2 1.8.4
```

Use pip to install virtualenv on your system:

```
1 $ pip install virtualenv
```

Let's practice creating a virtualenv:

1. Navigate to your “RealPython” directory.
2. Create a new directory called “test”.

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

3. Navigate into the new directory, then run the following command to set up a virtualenv within that directory:

```
1 virtualenv --no-site-packages env
```

This created a new directory, “env”, 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.

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

Unix:

```
1 $ source env/bin/activate
```

Windows:

```
1 $ env\scripts\activate
```

This changes the context of your terminal in that folder, “test”, which 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:

```
1 $ source env/bin/activate
2 (env)Michaels-MacBook-Pro-2:sample 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.

Go ahead and deactivate the virtualenv.

Everytime you create a new project, install a new virtualenv.

Video

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

Web Browsers

You can either use FireFox or Chrome during this course, because we will make use of a powerful set of tools for web developers called FireBug or Chrome Developer Tools. These tools allow you to inspect and analyze various elements that make up a webpage. You can view HTTP requests, test CSS style changes, and debug code, among others.

If you choose to use FireFox, you will need to install FireBug. I *highly* recommend using Chrome, as most examples in this course will be shown with Chrome. Plus, Chrome comes with Developer Tools pre-installed.

Install Firebug

Follow these directions if you need to install FireBug.

[Download](#) the latest version of Firefox if you don't already have it. Then follow these steps:

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.

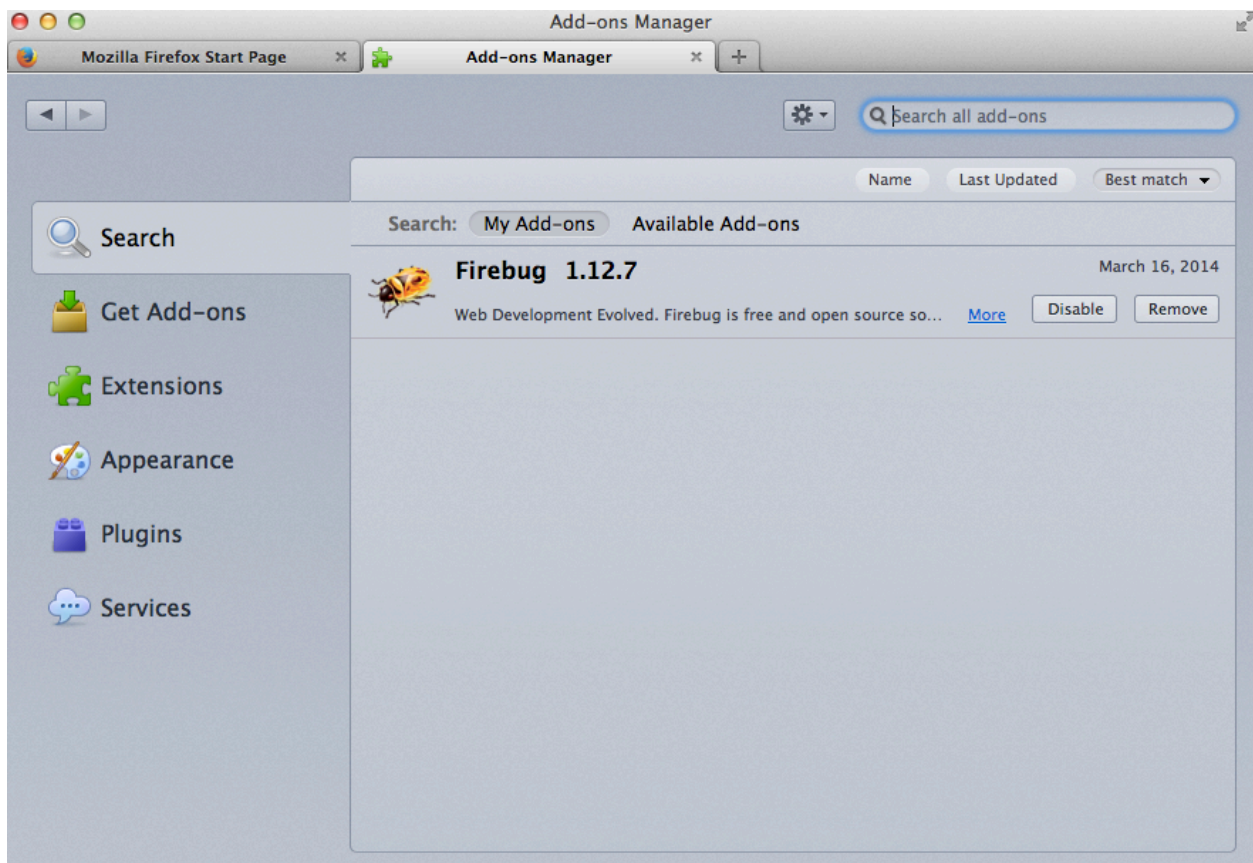


Figure 3.2: firebug

Version Control

Finally, to complete our development environment, we need to install a version control system. Such systems allow you to save different “versions” of a project. Over time, as your code and your project comes bigger, it may become necessary to backtrack to an earlier “version” to undo changes if a giant error occurs, for instance. It happens. We will be using Git for version control and Github for remotely hosting our code.

Setting up Git

It’s common practice to put projects under version control before you start developing by creating storage areas called repositories (or repos.)

This is an essential step in the development process. Again, such systems allow you to easily track changes when your codebase 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.

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. Make sure to download the version appropriate for your system.
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:

```
1 $ git config --global user.name "FIRST_NAME LAST_NAME"
2 $ git config --global user.email "MY_NAME@example.com"
```

3. Finally, I highly recommend reading Chapter 2 and 3 of the [Git Book](#). *Do not worry if you don’t understand everything, as long as you grasp the overall, high level concepts and work flow. You’ll learn better by doing, anyway.*

Introduction to Github

Github is related to Git, but it is distinct. While you use Git to create and maintain local repositories, Github is a social network used to remotely host those repositories so-

- Your projects are safe from potential damages to your local machine,
- You can show off your projects to potential employers (think of Github as your online resume), and

- Other users can collaborate on your project (open source!).

Get used to Github. You'll be using it a lot. It's the most popular place for programmers to host their projects.

1. Sign up for a new account on [Github](#). It's free!
2. Click the "New Repository" button on the main page. Or, simply follow [this](#) link.
3. Name the repository "RealPython", then in the description add the following text, "All my Real Python files!".
4. Click "Create Repository".

Congrats! You just setup a new repository on Github. Leave that page up while we create a local repository using Git.

Create a Local Repo

1. Go to your "RealPython" folder to initialize (create) your local repo:

```
1 $ git init
```

2. Next add a file called *readme.md*. It's convention to have such a file, stating the purpose of your project, so that when added to Github, other users can get a quick sense of what your project is all about.

```
1 $ touch README
```

3. Open that file in Sublime and just type "Hello, world! These are my Real Python repos." Save the file.
4. Now let's add the file to your local repo. First we need to take a "snapshot" of the project in it's current state:

```
1 $ git add .
```

This essentially adds all files that have either been created, updated, or deleted to a place called "staging". Don't worry about what that means; just know that your project is not yet part of the local repo yet.

5. Okay. So to add the project to your repo, you need to run the following command:

```
1 $ git commit -am "My initial commit message"
```

Sweet! Now your project has been committed (or added) to your local repo. Let's add (PUSH) it to Github now.

1. Add a link to your remote repository. Return to github. Copy the command to add your remote repo, then paste it into your terminal: `$ git remote add origin https://github.com/<YOUR-USERNAME>/`
2. Finally, PUSH (or send) the local repo to Github: `$ git push origin master`
3. Open your browser and refresh the Github page. You should now see the files from your local repository on Github.

That's all there is too it.

Review

Let's review.

When starting a new repository, you need to follow these steps:

1. Add the repo on Github.
2. Run the following commands in your local directory:

```
1 $ git init
2 $ git touch readme.md
3 $ git add .
4 $ git commit -am "message"
5 $ git remote add origin
   https://github.com/<YOUR-USERNAME>/<YOUR-REPO-NAME>.git
6 $ git push origin master
```

Again, this creates the necessary files and pushes them to the remote repository on Github.

3. Next, after your repo has been created locally and remotely - and you completed your first PUSH - you can follow this similar workflow to PUSH as needed:

```
1 $ git add .
2 $ git commit -am 'message'
3 $ git push origin master
```

NOTE: The string within the commit command should be replaced each time with a brief description of the changes made since the last PUSH.

4. That's it. With regard to Git, it's essential that you know how to:
 - Create a local and remote repo,
 - Add and commit, and
 - Push your local copy to Github.

We'll get into more specifics as we progress through the course.

Git Workflow

Now that you know how to setup a repo, let's review. So, you have one directory on your local computer called "realpython". The goal is to add all of our projects to that folder, and keep it in sync with our repo on Github.

To do so, each time you make any *major* changes to a specific project, commit them locally and then PUSH them to Github:

```
1 $ git add .
2 $ git commit -am "some message goes here about the commit"
3 $ git push origin master
```

Simple, right?

Next add a *.gitignore* file (no file extension!), which is a file that you can use to specify files you wish to ignore or keep out of your repository, both local and remote. What files should you keep out? If it's clutter (such as a *.pyc* file) or a secret (such as an API key), then keep it out of your public repo on Github.

Please read more about *.gitignore* [here](#). For now, add the files found [here](#). Start by adding "*.pyc" files

Save the *.gitignore* file in your main directory ("realpython"), then PUSH to Github:

```
1 $ git add .
2 $ git commit -am "added .gitignore file"
3 $ git push origin master
```

So, we have one directory, "realpython", that contains our local repository. Within that directory, we will add in all of your projects. Each project will contain a separate virtual environment.

Finally, we want to setup a *requirements.txt* file for *each* virtualenv. This file contains a list of packages you've installed via Pip. This is meant to be used by other developers to recreate the same development environment. To do this run the following command:

```
1 $ pip freeze > requirements.txt
```

Again, you have your main directory called "realpython" then withing that directory, you'll create serveral projet directories, each containing - a virtualenv and a *requirements.txt* file.

Homework

- Please read more about the basic Git commands [here](#).

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

Chapter 4

Interlude: Modern Web Development

Overview

Even though this is a Python course, we will be using a number of technologies such as HTML/CSS, JavaScript/jQuery, Ajax, REST, and SQL, to name a few. Put another way, you will be learning all the technologies needed, from the front-end to the back-end (and everything in between), which work in conjunction with Python web frameworks.

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

Before we start, let's look at an overview of modern web development.

Front-end, Back-end, and Middleware

Modern web applications are made up of three parts or layers, each of which encompasses 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 provides the structure, while CSS provides a pretty facade, and JavaScript/jQuery creates interaction. Essentially, this is what the end user sees when s/he interacts with your web application. The front-end is reliant on the application logic and data sources provided by the middleware and back-end (more on this later).
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 and often times 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.

Don't worry if you don't understand all of this. We will be discussing each of these technologies and how they fit into the whole throughout the course.

Developers adept in web architecture and in programming languages like Python, have traditionally worked on the back-end and middleware layers, while designers and those with HTML/CSS and JavaScript/jQuery skills, have focused on the front-end. These roles are becoming less and less defined, however - especially in start-ups. Traditional back-end 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 Foundation, 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 within each. There are pros and cons associated with this: It's great for experienced web developers, for example, who understand the automation (or *magic!*) behind the scenes.

Web frameworks simplify web development, by handling much of the superfluous, repetitious tasks. This can be very confusing for a beginner, though - which reinforces the need to start slow and work our way up.

Frameworks also separate the presentation (view) from the application logic (controller) and the underlying data (model) 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 *generally* operates in a triangular pattern:

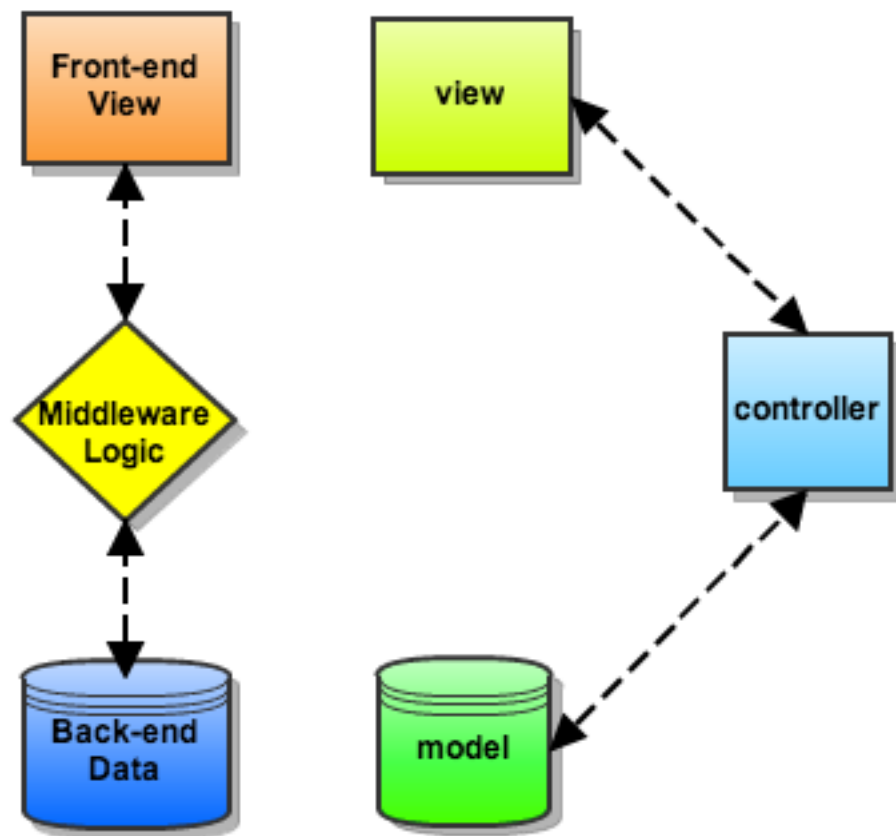


Figure 4.1: model-view-controller

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

Homework

- Read about the differences between a website (static) and a web application (dynamic) [here](#).

Chapter 5

Flask: QuickStart

This chapter introduces you to the Flask web framework, which is a web development framework for Python.



Figure 5.1: flask

Overview

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 complexity². Let's take a look.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

Installation

1. Within your “RealPython” directory create a new directory called “flask-hello-world”.
2. Navigate into the directory, create and activate a new virtualenv.
3. Now install Flask:

```
1 $ pip install flask
```

Hello World

Let's start with a quick Hello World example. This app will be contained entirely within a single file, *app.py*. Yes, it's that easy!

1. Open Sublime and within a new file add the following code:

```
1 # ---- Flask Hello World ---- #
2
3 # import the Flask class from the flask module
4 from flask import Flask
5
6 # create the application object
7 app = Flask(__name__)
8
9 # use decorators to link the function to a url
10 @app.route("/")
11 @app.route("/hello")
12
13 # define the view using a function, which returns a string
14 def hello_world():
15     return "Hello, World!"
16
17 # start the development server using the run() method
18 if __name__ == "__main__":
19     app.run()
```

2. Save the file as *app.py* and run it:

```
1 $ python app.py
```

3. 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.

NOTE: <http://127.0.0.1:5000> and <http://localhost:5000> are equivalent. So when you run locally, you can point your browser to either URL. Both will work.

4. 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?

Let's first look at the code without the function:

```

1 from flask import Flask
2
3 app = Flask(__name__)
4
5 if __name__ == "__main__":
6     app.run()

```

1. First, we imported the Flask class from the flask library in order to create our web application.
2. Next, we created an instance of Flask class and assigned it to the variable 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 (as a module) rather than importing it. Does this make sense? If not, check out [this link](#). You can read more about [modules](#) from the Python official documentation.

Now for the functions:

```

1 # use decorators to link the function to a url
2 @app.route("/")
3 @app.route("/hello")
4
5 # define the view using a function, which returns a string
6 def hello_world():
7     return "Hello, World!"

```

1. Here we applied two [decorators](#) - @app.route("/") and @app.route("/hello") to the hello_world() function. ³ These decorators are used to define routes. In other words, we created two routes - / and /hello - which are bound to our main url, http://127.0.0.1:5000. Thus, we are able to access the function by navigating to either http://127.0.0.1:5000 or http://127.0.0.1:5000/hello.
2. The function simply returned the string "Hello, World!".

Finally, we need to commit to Github. Move back to the "realpython" directory:

```

1 $ git add .
2 $ git commit -am "flask-hello-world"
3 $ git push origin master

```

From now on, I will remind you by just telling you to commit and PUSH to Github and the end of each chapter. Make sure you remember to do it after each lesson, though!

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

Dynamic Routes

Thus far, we've only looked at static routes. Let's quickly look at a dynamic route.

1. Add a new static route to *app.py*:

```
1 # dynamic route
2 @app.route("/test")
3 def search():
4     return "Hello"
```

Test it out.

2. Now to make it dynamic update first the route to take a query parameter:

```
1 @app.route("/test/<search_query>")
```

3. Next, update the function so that it takes the query parameter as an argument, then returns it to the screen:

```
1 def search(search_query):
2     return search_query
```

Navigate to <http://localhost:5000/test/hi>. You should see “hi” on the page. Test it out some more.

Debug Mode

Flask provides helpful error messages, and prints stack traces directly in the browser, making debugging much easier. To enable these features along with [automatic reload](#) simply add the following snippet to your *app.py* file:

```
1 app.config["DEBUG"] = True
```

The code should now look like this:

```
1 # import the Flask class from the flask module
2 from flask import Flask
3
4 # create the application object
5 app = Flask(__name__)
6
7 # error handling
8 app.config["DEBUG"] = True
9
10 # use decorators to link the function to a url
11 @app.route("/")
12 @app.route("/hello")
13
14 # define the view using a function, which returns a string
15 def hello_world():
16     return "Hello, World!"
17
18 # dynamic route
19 @app.route("/test/<search_query>")
20 def search(search_query):
21     return search_query
22
23 # start the development server using the run() method
24 if __name__ == "__main__":
25     app.run()
```

Again, with this simple modification, not only do we have error handling - but automatic reload as well.

Check your terminal, you should see:

```
1 * Restarting with reloader
```

Essentially, any time you make changes to the code and save them, they will be auto loaded. You do not have to restart your server to refresh the changes; you just need to refresh your browser.

Edit the string returned by the `hello_world()` function to:

```
1 return "Hello, World!?!?!?"
```

Save your code. Refresh the browser. You should see the new string.

We'll look at error handling in a later chapter. Until then, make sure you commit and PUSH your code to Github.

Kill the sever, then deactivate your virtualenv.

SEE ALSO: Want to use a different debugger? See [Working with Debuggers](#).

Chapter 6

Interlude: Database Programming

Before moving on to a more advanced Flask application, let's look at database programming.

Nearly every web application has to store information. One of the most common methods of storing (or persisting) information is to use a relational database. In this chapter, we'll look at the basics of relational databases as well as SQL (Structured Query Language).

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, which are uniquely identified by their name. Each table is comprised of columns called fields and rows called records (or data). Here is a sample table called “Employees”:

EmpNo	EmpName	DeptNo	DeptName
111	Michael	10	Development
112	Fletcher	20	Sales
113	Jeremy	10	Development
114	Carol	20	Sales
115	Evan	20	Sales

Records from one table can be linked to records in another table to create relationships. More on this later.

Most relational databases use SQL 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	inserts 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 (Object Relational Mapping) 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.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

Creating Tables

Let's begin. Make sure you've created a "sql" directory (again within your "realpython" directory). Then create and activate your virtualenv.

Use the Create Table statement to create a new table. Here is the basic format:

```
1 create table table_name
2 (column1 data_type,
3  column2 data_type,
4  column3 data_type);
```

Let's create a basic Python script to do this:

```
1 # Create a SQLite3 database and table
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create a new database if the database doesn't already exist
8 conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 # create a table
14 cursor.execute("""CREATE TABLE population
15                  (city TEXT, state TEXT, population INT)
16                  """)
17
18 # close the database connection
19 conn.close()
```

Save the file as *sqla.py*. Run the file from your terminal:

```
1 $ python sqla.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?

1. We imported the `sqlite3` library to communicate with SQLite.

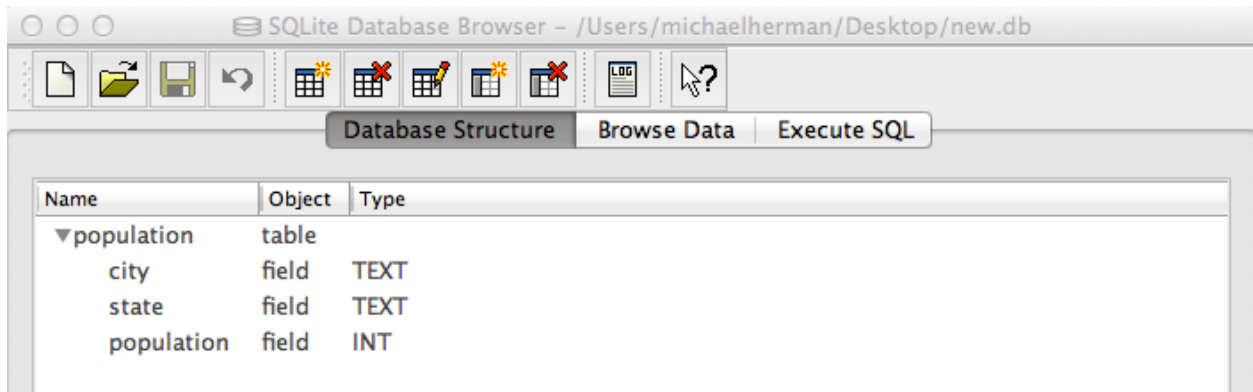


Figure 6.1: sqlite

2. 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.)
3. We then created a cursor object, which lets us execute a SQL query or command against data within the database.
4. 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".

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

```
1 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.

You can delete a table by using the Drop Table. This of course deletes the table and all the data associated with that table. *Use with caution.* `drop table table_name`

Don't forget to commit to Git!

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.

Inserting Data

Now that we have a table created, we can populate it with some actual data by adding new rows to the table. Strings should be enclosed in single quotes.

```
1 # INSERT Command
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create the connection object
8 conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 # insert data
14 cursor.execute("INSERT INTO population VALUES('New York City', 'NY', 8200000)")
15 cursor.execute("INSERT INTO population VALUES('San Francisco', 'CA', 800000)")
16
17 # commit the changes
18 conn.commit()
19
20 # close the database connection
21 conn.close()
```

Save the file as *sqlb.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.

1. As in the example from the previous lesson, we imported the `sqlite3` library, established the database connection, and created the cursor object.
2. We then used the `INSERT INTO` SQL command to insert data into the “population” table. Note how each item (except the integers) 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.
3. 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:

```
1 import sqlite3
2 with sqlite3.connect("new.db") as connection:
3     c = connection.cursor()
4     c.execute("INSERT INTO population VALUES('New York City', 'NY', 8200000)")
5     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.

```
1 # executemany() method
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # insert multiple records using a tuple
10    cities = [
11        ('Boston', 'MA', 600000),
12        ('Chicago', 'IL', 2700000),
13        ('Houston', 'TX', 2100000),
14        ('Phoenix', 'AZ', 1500000)
15    ]
16
17    # insert data into table
18    c.executemany('INSERT INTO population VALUES(?, ?, ?)', cities)
```

Save the file as `sqlc.py` 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). Parameterized 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.

```
1 # import from CSV
2
3 # import the csv library
4 import csv
5
6 import sqlite3
7
8 with sqlite3.connect("new.db") as connection:
9     c = connection.cursor()
10
11     # open the csv file and assign it to a variable
12     employees = csv.reader(open("employees.csv", "rU"))
13
14     # create a new table called employees
15     c.execute("CREATE TABLE employees(firstname, lastname)")
16
17     # insert data into table
18     c.executemany("INSERT INTO employees(firstname, lastname) values (?, ?)",
19                  employees)
```

Run the file. Now if you look in SQLite, you should see a new table called “employees” with 20 rows of data in it.

Try/Except

Remember when I said, “if you did not receive an error, then you can assume the code ran correctly”. Well, what happens if you did see an error? We want to handle this gracefully. Let's rewrite this using Try/Except:

```
1 # INSERT Command with Error Handler
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create the connection object
```

```

8 conn = sqlite3.connect("new.db")
9
10 # get a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 try:
14     # insert data
15     cursor.execute("INSERT INTO populations VALUES('New York City', 'NY',
16     8200000)")
17     cursor.execute("INSERT INTO populations VALUES('San Francisco', 'CA',
18     800000)")
19
20     # commit the changes
21     conn.commit()
22 except sqlite3.OperationalError:
23     print "Oops!  Something went wrong. Try again..."
24
25 # close the database connection
26 conn.close()

```

Notice how I intentionally named the table “populations” instead of “population”. Oops. Any idea how you could throw an exception, but also provide the user with relevant information about how to correct the issue? Google it!

Searching

Let's now look at how to retrieve data:

```
1 # SELECT statement
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # use a for loop to iterate through the database, printing the results line
10    # by line
11    for row in c.execute("SELECT firstname, lastname from employees"):
12        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 characters². 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:

```
1 # SELECT statement, remove unicode characters
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("SELECT firstname, lastname from employees")
10
11    # fetchall() retrieves all records from the query
12    rows = c.fetchall()
13
14    # output the rows to the screen, row by row
15    for r in rows:
16        print r[0], r[1]
```

1. First, the `fetchall()` method retrieved all records from the query and stored them as a tuple - or, more precisely: tuples within a tuple.
2. We then assigned the records to the “rows” variable.
3. Finally, we printed the values using index notation, `print r[0], r[1]`.

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

Updating and Deleting

This lesson covers how to use the UPDATE and DELETE SQL commands to change or delete records that match a specified criteria.

```
1 # UPDATE and DELETE statements
2
3 import sqlite3
4
5 with sqlite3.connect("new.db") as connection:
6     c = connection.cursor()
7
8     # update data
9     c.execute("UPDATE population SET population = 9000000 WHERE city='New York City'")
10
11    # delete data
12    c.execute("DELETE FROM population WHERE city='Boston'")
13
14    print "\nNEW DATA:\n"
15
16    c.execute("SELECT * FROM population")
17
18    rows = c.fetchall()
19
20    for r in rows:
21        print r[0], r[1], r[2]
```

1. 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.
2. We then displayed the results using the SELECT command.
3. 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:

```
1 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 previous homework assignment, add (INSERT) 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.

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.

```
1 # executemany() method
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # insert multiple records using a tuple
10    # (you can copy and paste the values)
11    cities = [
12        ('Boston', 'MA', 600000),
13        ('Los Angeles', 'CA', 38000000),
14        ('Houston', 'TX', 2100000),
15        ('Philadelphia', 'PA', 1500000),
16        ('San Antonio', 'TX', 1400000),
17        ('San Diego', 'CA', 130000),
18        ('Dallas', 'TX', 1200000),
19        ('San Jose', 'CA', 900000),
20        ('Jacksonville', 'FL', 800000),
21        ('Indianapolis', 'IN', 800000),
22        ('Austin', 'TX', 800000),
23        ('Detroit', 'MI', 700000)
24    ]
25
26    c.executemany("INSERT INTO population VALUES(?, ?, ?)", cities)
27
28    c.execute("SELECT * FROM population WHERE population > 1000000")
29
30    rows = c.fetchall()
31
32    for r in rows:
33        print r[0], r[1], r[2]
```

Check SQLite to ensure the data was entered properly.

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:

```
1 # Create a table and populate it with data
```

```

2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("""CREATE TABLE regions
10                (city TEXT, region TEXT)
11                """)
12
13     # (again, copy and paste the values if you'd like)
14     cities = [
15         ('New York City', 'Northeast'),
16         ('San Francisco', 'West'),
17         ('Chicago', 'Midwest'),
18         ('Houston', 'South'),
19         ('Phoenix', 'West'),
20         ('Boston', 'Northeast'),
21         ('Los Angeles', 'West'),
22         ('Houston', 'South'),
23         ('Philadelphia', 'Northeast'),
24         ('San Antonio', 'South'),
25         ('San Diego', 'West'),
26         ('Dallas', 'South'),
27         ('San Jose', 'West'),
28         ('Jacksonville', 'South'),
29         ('Indianapolis', 'Midwest'),
30         ('Austin', 'South'),
31         ('Detroit', 'Midwest')
32     ]
33
34     c.executemany("INSERT INTO regions VALUES(?, ? )", cities)
35
36     c.execute("SELECT * FROM regions ORDER BY region ASC")
37
38     rows = c.fetchall()
39
40     for r in rows:
41         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

The real power of relational tables comes from the ability to link data from two or more tables. This is achieved by using the JOIN command.

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

Code:

```
1 # JOINing data from multiple tables
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     # retrieve data
10    c.execute("""SELECT population.city, population.population,
11                regions.region FROM population, regions
12                WHERE population.city = regions.city""")
13
14    rows = c.fetchall()
15
16    for r in rows:
17        print r[0], r[1], r[2]
```

Take a look at the SELECT statement.

1. 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).
2. 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:

```
1 # JOINing data from multiple tables - cleanup
2
3
4 import sqlite3
5
6 with sqlite3.connect("new.db") as connection:
7     c = connection.cursor()
8
9     c.execute("""SELECT DISTINCT population.city, population.population,
10                regions.region FROM population, regions WHERE
11                population.city = regions.city ORDER by population.city ASC""")
12
```



```
13 rows = c.fetchall()
14
15 for r in rows:
16     print "City: " + r[0]
17     print "Population: " + str(r[1])
18     print "Region: " + r[2]
19     print
```

Homework

- 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). Make sure to research how to
- 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.

SQL Functions

SQLite has many built-in functions for aggregating and calculating data returned from a SELECT statement.

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

```
1 # SQLite Functions
2
3 import sqlite3
4
5 with sqlite3.connect("new.db") as connection:
6     c = connection.cursor()
7
8     # create a dictionary of sql queries
9     sql = {'average': "SELECT avg(population) FROM population",
10           'maximum': "SELECT max(population) FROM population",
11           'minimum': "SELECT min(population) FROM population",
12           'sum': "SELECT sum(population) FROM population",
13           'count': "SELECT count(city) FROM population"}
14
15     # run each sql query item in the dictionary
16     for keys, values in sql.iteritems():
17
18         # run sql
19         c.execute(values)
20
21         # fetchone() retrieves one record from the query
22         result = c.fetchone()
23
24         # output the result to screen
25         print keys + ":", result[0]
```

1. Essentially, we created a dictionary of SQL statements and then looped through the dictionary, executing each statement.

2. 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!**

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 (drawing) 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 *assignment3a.py* and *assignment3b.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.

First Script

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

```
1 import sqlite3
2 import random
```

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

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

3. Open the cursor:

```
1 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):

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

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

```

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

```

Full Code:

```

1 # Assignment 3a - insert random data
2
3
4 # import the sqlite3 library
5 import sqlite3
6 import random
7
8 with sqlite3.connect("newnum.db") as connection:
9     c = connection.cursor()
10
11     # delete database table if exist
12     c.execute("DROP TABLE if exists numbers")
13
14     # create database table
15     c.execute("CREATE TABLE numbers(num int)")
16
17     # insert each number to the database
18     for i in range(100):
19         c.execute("INSERT INTO numbers VALUES(?)", (random.randint(0,100),))

```

Second Script

Code:

```

1 # Assignment 3b - prompt the user
2
3
4 # import the sqlite3 library
5 import sqlite3
6
7 # create the connection object
8 conn = sqlite3.connect("newnum.db")
9
10 # create a cursor object used to execute SQL commands
11 cursor = conn.cursor()
12
13 prompt = """
14 Select the operation that you want to perform [1-5]:
15 1. Average
16 2. Max

```

```

17 3. Min
18 4. Sum
19 5. Exit
20 """
21
22 # loop until user enters a valid operation number [1-5]
23 while True:
24     # get user input
25     x = raw_input(prompt)
26
27     # if user enters any choice from 1-4
28     if x in set(["1", "2", "3", "4"]):
29         # parse the corresponding operation text
30         operation = {1: "avg", 2: "max", 3: "min", 4: "sum"}[int(x)]
31
32         # retrieve data
33         cursor.execute("SELECT {}(num) from numbers".format(operation))
34
35         # fetchone() retrieves one record from the query
36         get = cursor.fetchone()
37
38         # output result to screen
39         print operation + ": %f" % get[0]
40
41     # if user enters 5
42     elif x == "5":
43         print "Exit"
44
45     # exit loop
46     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.

Also, I hope you committed and PUSHed to Github after each lesson. If not, shame on you. *Do it now.*

Chapter 7

Flask Blog App

Let's build a blog!

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. Within your “realpython” directory create a “flask-blog” directory.
2. Create and activate a virtualenv.
3. Make sure to place your app under version control (back in your “realpython” directory).
4. Set up the following files and directories:

```
1
2  blog.py
3  static
4      css
5      img
6      js
7  templates
```

First all of the logic (Python/Flask code) goes in the *blog.py* file. Our “static” directory holds static files like Javascript files, CSS stylesheets, and images. Finally, the “template” folder will house all of our HTML files.

The important thing to note is that the *blog.py* acts as the application controller. Flask works with a client-server model. The server receives HTTP request from the client (i.e., the web browser), then returns content back to the client in the form of a response:

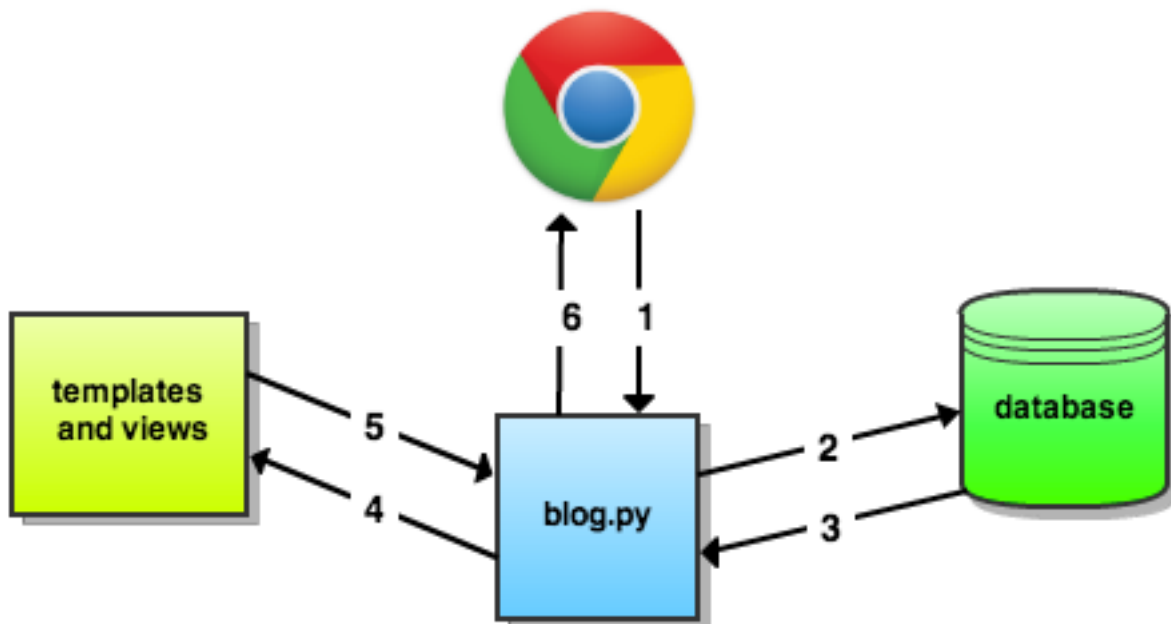


Figure 7.1: flask mvc

5. Install flask:

```
1 pip install flask
```

Let's build our app!

Model

Our database has one table called *posts* with two fields - *title* and *post*. We can use the following script to create and populate the database:

```
1 # sql.py - Create a SQLite3 table and populate it with data
2
3
4 import sqlite3
5
6 # create a new database if the database doesn't already exist
7 with sqlite3.connect("blog.db") as connection:
8
9     # get a cursor object used to execute SQL commands
10    c = connection.cursor()
11
12    # create the table
13    c.execute("""CREATE TABLE posts
14               (title TEXT, post TEXT)
15               """)
16
17    # insert dummy data into the table
18    c.execute('INSERT INTO posts VALUES("Good", "I\'m good.")')
19    c.execute('INSERT INTO posts VALUES("Well", "I\'m well.")')
20    c.execute('INSERT INTO posts VALUES("Excellent", "I\'m excellent.")')
21    c.execute('INSERT INTO posts VALUES("Okay", "I\'m okay.")')
```

Save the file within your “flask-blog” directory. Run it. Then, check the SQLite Browser to ensure the table was created correctly and populated with data. *Notice how we escaped the apostrophe in the INSERT statements.*

Controller

Like the controller in the `hello_world` app (*app.py*), this script will define the imports, configurations, and each view.

```
1 # blog.py - controller
2
3
4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
8
9 # configuration
10 DATABASE = 'blog.db'
11
12 app = Flask(__name__)
13
14 # pulls in app configuration by looking for UPPERCASE variables
15 app.config.from_object(__name__)
16
17 # function used for connecting to the database
18 def connect_db():
19     return sqlite3.connect(app.config['DATABASE'])
20
21 if __name__ == '__main__':
22     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.¹

Make sure you understand how this is working:

```
1 # configuration
2 DATABASE = 'blog.db'
3
4 ...
5
6 # pulls in app configuration by looking for UPPERCASE variables
7 app.config.from_object(__name__)
```

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

Views

After a user logs in, s/he 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.

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4 {% endblock %}
```

We also need a login page:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <h3>Please login to access your blog.</h3>
5     <p>Temp Login: <a href="/main">Login</a></p>
6 {% endblock %}
```

Save these files as *main.html* and *login.html* respectively in the “templates” directory. I know you have questions about the strange code (i.e., `{% extends "template.html" %}`) 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:

```
1 @app.route('/')
2 def login():
3     return render_template('login.html')
4
5 @app.route('/main')
6 def main():
7     return render_template('main.html')
```

Updated code:

```
1 # blog.py - controller
2
3
4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
8
9 # configuration
10 DATABASE = 'blog.db'
11
12 app = Flask(__name__)
```

```

13
14 # pulls in configurations by looking for UPPERCASE variables
15 app.config.from_object(__name__)
16
17 # function used for connecting to the database
18 def connect_db():
19     return sqlite3.connect(app.config['DATABASE'])
20
21 @app.route('/')
22 def login():
23     return render_template('login.html')
24
25 @app.route('/main')
26 def main():
27     return render_template('main.html')
28
29 if __name__ == '__main__':
30     app.run(debug=True)

```

In the first function, `login()`, we mapped the URL `/` to the function, which in turn sets the route to *login.html* in the templates directory. How about the `main()` function? What's going on there? Explain it to yourself. Say it out loud.

Templates

Templates (*template.html*, in our case) 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 back-end functionality.

Further, templates make it much easier to combine HTML and Python in a programic-like manner.

Let's start with a basic template:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Welcome, friends!</title>
5   </head>
6   <body>
7     <div class="container">
8       {% block content %}
9       {% endblock %}
10    </div>
11  </body>
12 </html>
```

Save this as *template.html* within your templates directory.

There's a relationship between the template, *template.html*, and the views, *login.html* and *main.html*. This is called template inheritance. Essentially, our views extend, or are a child of, *template.html*. This is achieved by using the `{% extends "template.html" %}`. This tag establishes the relationship between the template and views. When Flask renders *main.html* it must first render *template.html*.

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*, are filled 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:

NOTE: Anything surrounded by `{% %}` is Python-like code, while objects surrounded by `{{ }}` are variables.²

There are a number of different templating formats. Flask uses Jinja2 templates. If interested, you can read more about them [here](#).

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

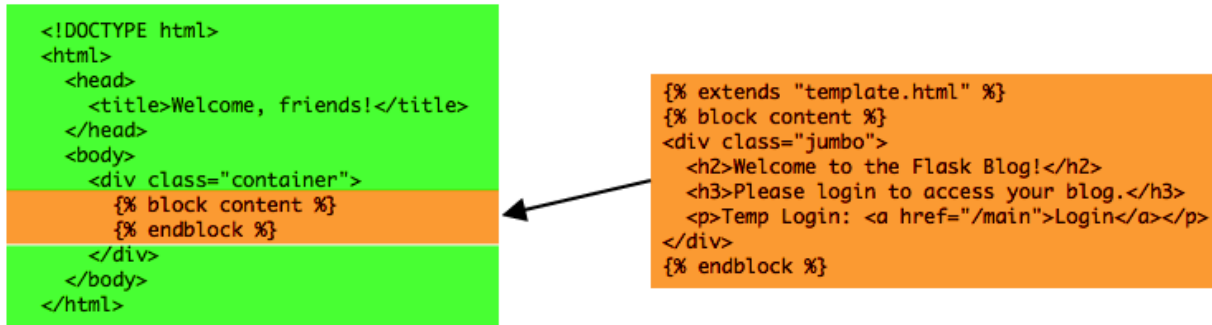


Figure 7.2: flask templates

Homework

- Read more about Jinja templating from [this](#) blog post

Run the server!

All right! Fire up your server (`python blog.py`), 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. Make sure all your files are saved. Try again. If you are still having problems, double-check your code against mine.

What's going on?

So, when `render_template()`, Flask immediately recognizes that *login.html* extends *template.html*. Flask renders *template.html*, then fills in the block tags, `{% block content %}` and `{% endblock %}`, with the code found in *login.html*.

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, 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*:

```
1 USERNAME = 'admin'
2 PASSWORD = 'admin'
```

Also in the configuration section, add the `secret_key`, which is used for managing user sessions:

```
1 SECRET_KEY = 'hard_to_guess'
```

WARNING: 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. ³

Updated *blog.py* configuration:

```
1 # configuration
2 DATABASE = 'fblog.db'
3 USERNAME = 'admin'
4 PASSWORD = 'admin'
5 SECRET_KEY = 'hard to guess'
```

Update the `login()` function in the *blog.py* file to match the following code:

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     if request.method == 'POST':
5         if request.form['username'] != app.config['USERNAME'] or
6         request.form['password'] != app.config['PASSWORD']:
7             error = 'Invalid Credentials. Please try again.'
8         else:
9             session['logged_in'] = True
10            return redirect(url_for('main'))
11    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

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

and the session key, `logged_in`, is set to `True`. If the wrong information is entered, an error message is flashed to the user.

Walk through this function line by line, saying *aloud* what each line accomplishes.

Now, we need to update *login.html* to include the HTML form:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <h3>Please login to access your blog.</h3>
5     <form action="" method="post">
6         Username: <input type="text" name="username" value="{{
7             request.form.username }}">
8         Password: <input type="password" name="password" value="{{
9             request.form.password }}">
10        <p><input type="submit" value="Login"></p>
11    </form>
12 {% endblock %}
```

If you are unfamiliar with how HTML forms work, please visit this [link](#).

Next, add a function for logging out to *blog.py*:

```
1 @app.route('/logout')
2 def logout():
3     session.pop('logged_in', None)
4     flash('You were logged out')
5     return redirect(url_for('login'))
```

The `logout()` function uses `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 %}`)-

```
1 {% for message in get_flashed_messages() %}
2     <div class="flash">{{ message }}</div>
3 {% endfor %}
4 {% if error %}
5     <p class="error"><strong>Error:</strong> {{ error }}
6 {% endif %}
```

-so that the template now looks like this:

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Welcome, friends!</title>
5     </head>
```

```

6 <body>
7   <div class="container">
8     {% for message in get_flashed_messages() %}
9       <div class="flash">{{ message }}</div>
10    {% endfor %}
11    {% if error %}
12      <p class="error"><strong>Error:</strong> {{ error }}
13    {% endif %}
14    <!-- inheritance -->
15    {% block content %}
16    {% endblock %}
17    <!-- end inheritance -->
18  </div>
19 </body>
20 </html>

```

Finally, add a logout link to the *main.html* page:

```

1 {% extends "template.html" %}
2 {% block content %}
3   <h2>Welcome to the Flask Blog!</h2>
4   <p><a href="/logout">Logout</a></p>
5 {% endblock %}

```

View it! Fire up the server (if needed). Manually test everything out. Make sure you can login and logout and that the appropriate messages are displayed, depending on the situation.

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 in a secure manner, usually as a token, within a cookie. In this case, when the session key, `logged_in`, is set to `True`, the user has the right(s) to view the *main.html* page. Go back and take a look at the `login()` function so you can see this logic.

The `login_required` decorator, meanwhile, checks to make sure that a user is authorized (e.g., `logged_in`) before allowing access to certain page. To implement this, we will set up a new function which will be used to restrict access to *main.html*.

Start by importing `functools` within your controller, *blog.py*:

```
1 from functools import wraps
```

`Functools` is a module used for extending the capabilities of functions with other functions.⁴ First, setup the new function in *blog.py*:

```
1 def login_required(test):
2     @wraps(test)
3     def wrap(*args, **kwargs):
4         if 'logged_in' in session:
5             return test(*args, **kwargs)
6         else:
7             flash('You need to login first.')
8             return redirect(url_for('login'))
9     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:

```
1 @login_required
2 @app.route('/main')
3 def main():
4     return render_template('main.html')
```

Updated code:

```
1 # blog.py - controller
2
3
```

⁴<http://south.aeracode.org/>

```

4 # imports
5 from flask import Flask, render_template, request, session, \
6     flash, redirect, url_for, g
7 import sqlite3
8 from functools import wraps
9
10 # configuration
11 DATABASE = 'blog.db'
12 USERNAME = 'admin'
13 PASSWORD = 'admin'
14 SECRET_KEY = 'hard_to_guess'
15
16 app = Flask(__name__)
17
18 # pulls in configurations by looking for UPPERCASE variables
19 app.config.from_object(__name__)
20
21 # function used for connecting to the database
22 def connect_db():
23     return sqlite3.connect(app.config['DATABASE'])
24
25 def login_required(test):
26     @wraps(test)
27     def wrap(*args, **kwargs):
28         if 'logged_in' in session:
29             return test(*args, **kwargs)
30         else:
31             flash('You need to login first.')
32             return redirect(url_for('login'))
33     return wrap
34
35 @app.route('/', methods=['GET', 'POST'])
36 def login():
37     error = None
38     if request.method == 'POST':
39         if request.form['username'] != app.config['USERNAME'] or
40 request.form['password'] != app.config['PASSWORD']:
41             error = 'Invalid Credentials. Please try again.'
42         else:
43             session['logged_in'] = True
44             return redirect(url_for('main'))
45     return render_template('login.html', error=error)
46
47 @app.route('/main')
48 @login_required
49 def main():
50     return render_template('main.html')

```

```

50
51 @app.route('/logout')
52 def logout():
53     session.pop('logged_in', None)
54     flash('You were logged out')
55     return redirect(url_for('login'))
56
57 if __name__ == '__main__':
58     app.run(debug=True)

```

When a GET request is sent to access *main.html* to view the HTML, it first hits the `@login_required` decorator and the entire function (`main()`) 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.

NOTE: Notice how we had to specify a POST request. By default, routes are setup automatically to answer/respond to GET requests. If you need to add different HTTP methods, such as a POST, you must add the `methods` argument to the decorator.

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:

```

1 127.0.0.1 - - [01/Feb/2014 11:37:56] "POST / HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 11:37:56] "GET /main HTTP/1.1" 200 -

```

Here, 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.

Once logged in, the session token is stored on the client side within a cookie. You can view this token in your browser by opening up Developer Tools in Chrome, clicking the “Resources” tab, then looking at your cookies:

2. Logout:

```

1 127.0.0.1 - - [01/Feb/2014 11:38:53] "GET /logout HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 11:38:53] "GET / HTTP/1.1" 200 -

```

When you logged out, you are actually issuing a GET request that responds by redirecting you to *login.html*. Again, this request was successful.

3. Login failed:

```

1 127.0.0.1 - - [01/Feb/2014 11:40:07] "POST / HTTP/1.1" 200 -

```

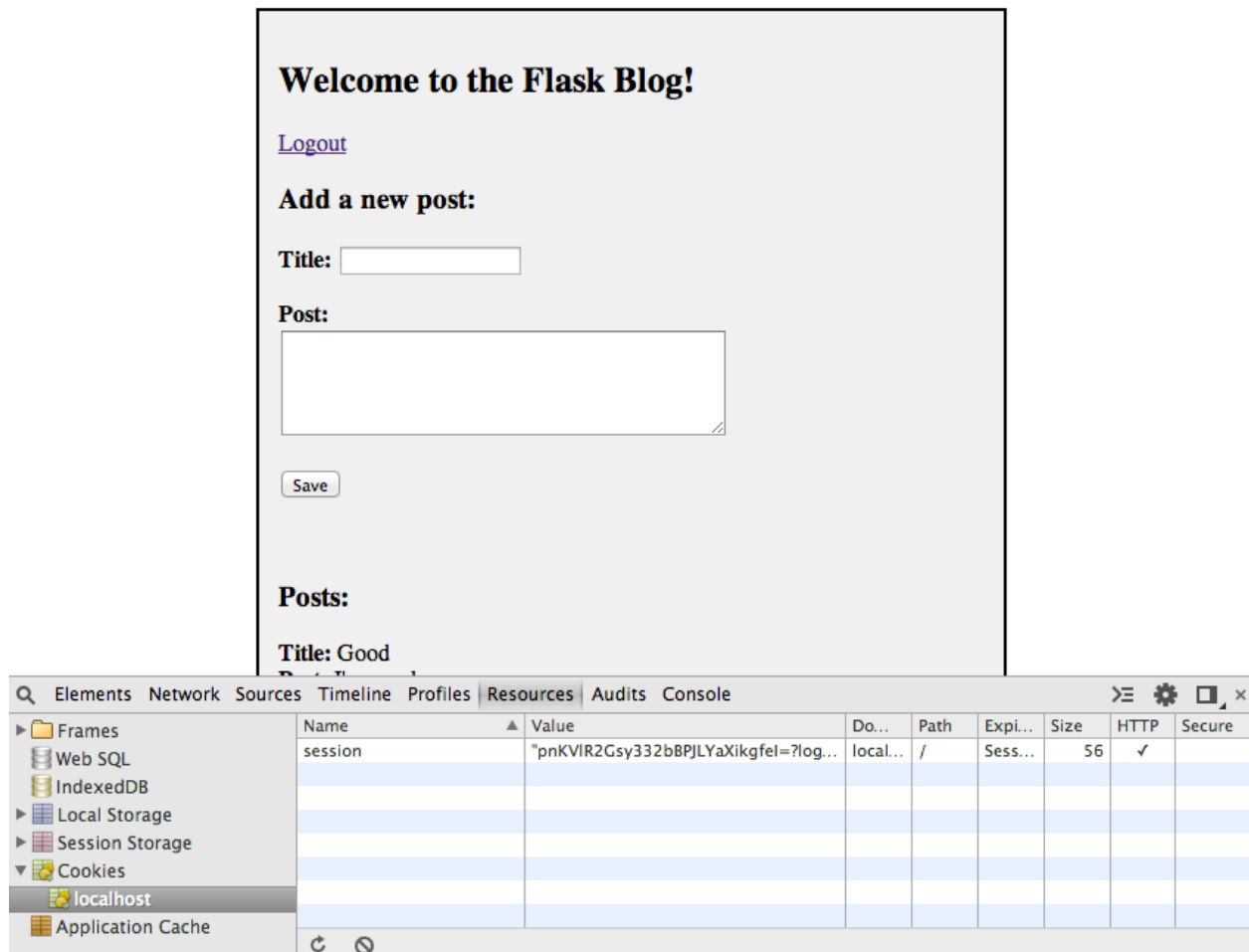


Figure 7.3: session_token

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:

```
1 127.0.0.1 - - [01/Feb/2014 11:44:56] "GET /main HTTP/1.1" 302 -  
2 127.0.0.1 - - [01/Feb/2014 11:44:56] "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/stack trace 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:

```
1 127.0.0.1 - - [01/Feb/2014 11:52:31] "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 basic security is set up, we need to display some information to the user. Otherwise, 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*:

```
1 @app.route('/main')
2 @login_required
3 def main():
4     g.db = connect_db()
5     cur = g.db.execute('select * from posts')
6     posts = [dict(title=row[0], post=row[1]) for row in cur.fetchall()]
7     g.db.close()
8     return render_template('main.html', posts=posts)
```

1. `g.db = connect_db()` connects to the database.
2. `cur = g.db.execute('select * from posts')` then fetches data from the “posts” table.
3. `posts = [dict(title=row[0], post=row[1]) for row in cur.fetchall()]` assigns the data retrieved from the database to a dictionary, which is assigned to the variable `posts`.
4. `posts=posts` passes that variable to the *main.html* file.

We next need to edit *main.html* to loop through the dictionary in order to display the tiles and posts:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <p><a href="/logout">Logout</a></p>
5     <br/>
6     <br/>
7     <h3>Posts:</h3>
8     {% for p in posts %}
9         <strong>Title:</strong> {{ p.title }} <br/>
10        <strong>Post:</strong> {{ p.post }} <br/>
11        <br/>
12    {% endfor %}
13 {% 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.

Check this out in our browser!

Add Posts

Finally, users need the ability to add new posts. We can start by adding a new function to *blog.py* called `add()`:

```
1 @app.route('/add', methods=['POST'])
2 @login_required
3 def add():
4     title = request.form['title']
5     post = request.form['post']
6     if not title or not post:
7         flash("All fields are required. Please try again.")
8         return redirect(url_for('main'))
9     else:
10        g.db = connect_db()
11        g.db.execute('insert into posts (title, post) values (?, ?)',
12                     [request.form['title'], request.form['post']])
13        g.db.commit()
14        g.db.close()
15        flash('New entry was successfully posted!')
16        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.

NOTE: The above description is a high-level overview of what's really happening. Get granular with it. Read what each line accomplishes aloud. Repeat to a friend.

Next, add the HTML form to the *main.html* page:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h2>Welcome to the Flask Blog!</h2>
4     <p><a href="/logout">Logout</a></p>
5     <h3>Add a new post:</h3>
6     <form action="{ url_for('add') }" method="post" class="add">
7         <label><strong>Title:</strong></label>
8         <input name="title" type="text">
9         <p><label><strong>Post:</strong></label><br>
10        <textarea name="post" rows="5" cols="40"></textarea></p>
11        <input class="button" type="submit" value="Save">
12    </form>
13    <br>
14    <br>
15    <h3>Posts:</h3>
16    {% for p in posts %}
```

```
17 <strong>Title:</strong> {{ p.title }} <br>
18 <strong>Post:</strong> {{ p.post }} <br>
19 <br>
20 {% endfor %}
21 {% endblock %}
```

Test this out! Make sure to add a post.

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:

```
1 127.0.0.1 - - [01/Feb/2014 12:14:24] "POST /add HTTP/1.1" 302 -
2 127.0.0.1 - - [01/Feb/2014 12:14:24] "GET /main HTTP/1.1" 200 -
```

Style

All right. Now that the app is working properly, let's make it look a bit nicer. To do this, we can edit the HTML, CSS, and/or Javascript. I'll show you a more in-depth example in a latter chapter. For now, though, let's just create something very simple. *Also, we will be covering HTML and CSS in more depth in a latter chapter as well. For now, please just follow along.*

```
1 .container {  
2     background: #f4f4f4;  
3     margin: 2em auto;  
4     padding: 0.8em;  
5     width: 30em;  
6     border: 2px solid #000;  
7 }  
8  
9 .flash, .error {  
10     background: #000;  
11     color: #fff;  
12     padding: 0.5em;  
13 }
```

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:

```
1 <link rel="stylesheet" href="{ url_for('static', filename='css/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. :)



Welcome to the Flask Blog!

Please login to access your blog.

Username: Password:

Figure 7.4: flask blog

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?

Make sure to commit to Git and then PUSH to Github!

Homework

- Now take a look at the accompanying [video](#) to see how to deploy your app on PythonAnywhere.

Chapter 8

Interlude: Debugging in Python

When solving complicated coding problems, it's important to use an interactive debugger for examining executed code line by line. Python provides such a tool called `pdb` (or “Python DeBugger”) within the standard library, where you can set breakpoints, step through your code, and inspect the stack.

Always keep in mind that while `pdb`'s primary purpose is debugging code, it's more important that you *understand* what's happening in your code while debugging. This in itself will help with debugging.

Workflow

Let's look at a simple example.

1. Save the following code as *pdb_ex.py* in the “debugging” folder:

```
1 import sys
2 from random import choice
3
4 random1 = [1,2,3,4,5,6,7,8,9,10,11,12]
5 random2 = [1,2,3,4,5,6,7,8,9,10,11,12]
6
7 while True:
8     print "To exit this game type 'exit'"
9     answer = raw_input("What is {} times {}? ".format(choice(random2),
10 choice(random1)))
11
12     # exit
13     if answer == "exit":
14         print "Now exiting game!"
15         sys.exit()
16
17     # determine if number is correct
18     elif answer == choice(random2) * choice(random1):
19         print "Correct!"
20     else:
21         print "Wrong!"
```

Run it. See the problem? There's either an issue with the multiplication or the logic within the `if` statement.

Let's debug!

2. Import the `pdb` module:

```
1 import pdb
```

3. Next, add `pdb.set_trace()` within the function to set your first breakpoint:

```
1 import pdb
2 import sys
3 from random import choice
4
5 random1 = [1,2,3,4,5,6,7,8,9,10,11,12]
6 random2 = [1,2,3,4,5,6,7,8,9,10,11,12]
7
8 while True:
```



```

9     print "To exit this game type 'exit'"
10    pdb.set_trace()
11    answer = raw_input("What is {} times {}? ".format(choice(random2),
12                                                         choice(random1)))
13
14    # exit
15    if answer == "exit":
16        print "Now exiting game!"
17        sys.exit()
18
19    # determine if number is correct
20    elif answer == choice(random2) * choice(random1):
21        print "Correct!"
22    else:
23        print "Wrong!"

```

4. When you run the code you should see the following output:

```

1  $ python pdb_ex.py
2    To exit this game type 'exit'
3    > /debugger/pdb_ex2.py(11)<module>()
4    -> answer = raw_input("What is {} times {}? ".format(choice(random2),
5                                                         choice(random1)))
6    (Pdb)

```

Essentially, when the Python interpreter runs the `pdb.set_trace()` line, the program stops and you'll see the next line in the program as well as a prompt (or console), waiting for your input.

From here you can start stepping through your code to see what happens line by line. Check out the list of commands you have access to [here](#). There's quite a lot of commands, which is daunting - but on a day-to-day basis, you'll only use a few common commands:

- `n`: step forward one line
- `p <variable name>`: prints the current value of the provided variable
- `l`: displays the entire program along with where the current break point is
- `q`: exits the debugger and the program
- `c`: exits the debugger and the program continues to run
- `b <line #>`: adds a breakpoint at a specific line #

NOTE If you don't remember the list of commands you can always type `?` or `help` to see the entire list.

Let's debug this together.

5. First, see what the value of `answer` is:

```

1 (Pdb) n
2 What is 7 times 8? 56
3 > /debugger/pdb_ex2.py(14)<module>()
4 -> if answer == "exit":
5 (Pdb) p answer
6 '56'

```

6. Next, let's continue through the program to see if that value (56) changes:

```

1 (Pdb) n
2 > /debugger/pdb_ex2.py(19)<module>()
3 -> elif answer == choice(random2) * choice(random1):
4 (Pdb) n
5 > /debugger/pdb_ex2.py(22)<module>()
6 -> print "Wrong!"
7 (Pdb) p answer
8 '56'

```

So, the answer does not change. There must be something wrong with the program logic in the `if` statement, starting with the `elif`.

7. Update the code for testing:

```

1 import pdb
2 import sys
3 from random import choice
4
5 random1 = [1,2,3,4,5,6,7,8,9,10,11,12]
6 random2 = [1,2,3,4,5,6,7,8,9,10,11,12]
7
8 while True:
9     print "To exit this game type 'exit'"
10    pdb.set_trace()
11    answer = raw_input("What is {} times {}? ".format(choice(random2),
12                                                         choice(random1)))
13
14    # exit
15    if answer == "exit":
16        print "Now exiting game!"
17        sys.exit()
18
19    test = int(choice(random2))*int(choice(random1))
20    # # determine if number is correct
21    # elif answer == choice(random2) * choice(random1):
22    #     print "Correct!"
23    # else:
24    #     print "Wrong!"

```

We just took the value we are using to test our answer against and set it to a variable.

8. Debug time:

```
1 $ python pdb_ex.py
2 To exit this game type 'exit'
3 > /debugger/pdb_ex2.py(11)<module>()
4 -> answer = raw_input("What is {} times {}? ".format(choice(random2),
5     choice(random1)))
6 (Pdb) n
7 What is 3 times 4? 12
8 > /debugger/pdb_ex2.py(14)<module>()
9 -> if answer == "exit":
10 (Pdb) n
11 > /debugger/pdb_ex2.py(18)<module>()
12 -> test = choice(random2) * choice(random1)
13 (Pdb) n
14 > /debugger/pdb_ex2.py(8)<module>()
15 -> while True:
16 (Pdb) p test
70
```

There's our answer. The value in the elif varies from the answer. Thus, the elif will always return "Wrong!".

9. Refactor:

```
1 import pdb
2 import sys
3 from random import choice
4
5 random1 = [1,2,3,4,5,6,7,8,9,10,11,12]
6 random2 = [1,2,3,4,5,6,7,8,9,10,11,12]
7
8 while True:
9     print "To exit this game type 'exit'"
10    num1 = choice(random2)
11    num2 = choice(random1)
12    answer = raw_input("What is {} times {}? ".format(choice(random2),
13        choice(random1)))
14
15    # exit
16    if answer == "exit":
17        print "Now exiting game!"
18        sys.exit()
```

```

19     # determine if number is correct
20     elif answer == num1 * num2:
21         print "Correct!"
22         break;
23     else:
24         print "Wrong!"

```

Ultimately, the program was generating new numbers for comparison within the elif causing the user input to be wrong each time.

10. Breakpoints

One thing we didn't touch on is setting breakpoints, which allows you to pause code execution at a certain line. To set a breakpoint while debugging, you simply call the break command then add the line number that you wish to break on: `b <line #>`

Simple example:

```

1 import pdb
2
3 def add_one(num):
4     result = num + 1
5     print result
6     return result
7
8 def main():
9     pdb.set_trace()
10    for num in xrange(0,10):
11        add_one(num)
12
13 if __name__ == "__main__":
14     main()

```

Save this as *pdb_ex2.py*.

Now, watch how to use the break command:

```

1 $ python pdb_ex2.py
2 >
   /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
3 -> for num in xrange(0,10):
4 (Pdb) b 4
5 Breakpoint 1 at
   /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
6 (Pdb) c
7 >
   /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
8 -> result = num + 1
9 (Pdb) args

```

```

10 num = 0
11 (Pdb) b 11
12 Breakpoint 2 at
    /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
13 (Pdb) c
14 1
15 >
    /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
16 -> add_one(num)
17 (Pdb) b 4
18 Breakpoint 3 at
    /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
19 (Pdb) c
20 >
    /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
21 -> result = num + 1
22 (Pdb) args
23 num = 1
24 (Pdb) c
25 2
26 >
    /Users/michaelherman/Documents/repos/realpython/course-files/book2/debugging/pdb_ex2.p
27 -> add_one(num)

```

Here, we started the debugger on line 9, then set a breakpoint on line 4. We continued the program until it hit that breakpoint. Then we checked the value of `num` - 0. We set another break at line 11, then continued again and saw that the result was 1 - `result = 0 + 1` - which is what we expected. Then we did the same process again and found that the next result was be 2 based on the value of `num` - `result = 1 + 1`.

Hope that makes sense.

Post Mortem Debugging

You can also use PDB to debug code that's already crashed, after the fact. Take the following code, for example:

```
1 def add_one_hundred():
2     again = 'yes'
3     while again == 'yes':
4         number = raw_input('Enter a number between 1 and 10: ')
5         new_number = (int(number) + 100)
6         print '{} plus 100 is {}'.format(number, new_number)
7         again = raw_input('Another round, my friend? (`yes` or `no`) ')
8     print "Goodbye!"
```

This function simply adds 100 to a number inputted by the user, then outputs the results to the screen.

What happens if you enter a string instead of a integer?

```
1 >>> from post_mortem_pdb import *
2 >>> add_one_hundred()
3 Enter a number between 1 and 10: test
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "post_mortem_pdb.py", line 5, in add_one_hundred
7     new_number = (int(number) + 100)
8 ValueError: invalid literal for int() with base 10: 'test'
9 >>>
```

Now we can use the PDB to start debugging where the exception occurred:

```
1 >>> import pdb; pdb.pm()
2 >
3     /Users/michaelherman/Documents/repos/realpython/book2-exercises/debugging/post_mortem_pdb.py
4 -> new_number = (int(number) + 100)
   (Pdb)
```

Start debugging!

So we know that the line `new_number = (int(number) + 100)` broke the code - because you can't convert a string to an integer.

This is a simple example, but imagine how useful this would be in a large program with multiple scripts that you can't fully visualize. You can immediately jump back into the program where the exception was thrown and start the debugging process. This can be incredibly useful. We'll look at example of just that when we start working with Flask and Django.

Homework

- Watch these two videos on debugging: [one](#) and [two](#).

Chapter 9

Flask: FlaskTaskr

Overview

In this section, we will develop a task manager called **FlaskTaskr**. We'll start by creating a simple skeleton app similar to the workflow from the blog app that we created in the last chapter, but we'll be adding plenty of new features and extensions 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.
2. New users can register on a registration page.
3. Once signed in, users can add new tasks. Each task consists of a name, due date, priority, status, and an auto-incremented ID.
4. Users can view all uncompleted tasks from the same screen.
5. 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. We'll be using the same process - but it will go much faster. *Make sure to commit your changes to local repo and PUSH to Github after each lesson.*

Homework

- Read about the differences between sessions and cookies [here](#).

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 explain what was already learned.

Let's get started.

1. Navigate to your "realpython" directory, and create a new directory for this chapter, "flask-taskr". Navigate into the newly created directory.
2. Create and activate a new virtualenv.
3. Install Flask:

```
1 $ pip install flask==0.10.1
```

Note how we specified the version.

5. Create a new directory called "app", which is the *project root directory*.
6. Setup the following files and directories within the root:

```
1
2 views.py
3 static
4     css
5     img
6     js
7 templates
```

Configuration

Remember how we placed all of our blog app's configurations directly in the controller? Well, it's best practice to actually place these in a separate file, then import that file into the controller.

Create a configuration file called *config.py* and save it in the project root:

```
1 # config.py
2
3
4 import os
5
6 # grabs the folder where the script runs
7 basedir = os.path.abspath(os.path.dirname(__file__))
8
9 DATABASE = 'flasktaskr.db'
```

```

10 USERNAME = 'admin'
11 PASSWORD = 'admin'
12 CSRF_ENABLED = True
13 SECRET_KEY = 'my_precious'
14
15 # defines the full path for the database
16 DATABASE_PATH = os.path.join(basedir, DATABASE)

```

Put simply, the `CSRF_ENABLED` config setting activates [cross-site request forgery](#) prevention, which makes your app more secure. The `SECRET_KEY` config [setting](#) is used in conjunction with `CSRF_ENABLED` in order to create a cryptographic token that is used to validate a form. Always make sure to set the secret key to something that is nearly impossible to guess. Use a [random key generator](#).

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.

```

1 # db_create.py
2
3
4 import sqlite3
5 from config import DATABASE_PATH
6
7 with sqlite3.connect(DATABASE_PATH) as connection:
8
9     # get a cursor object used to execute SQL commands
10    c = connection.cursor()
11
12    # create the table
13    c.execute("""CREATE TABLE ftasks(task_id INTEGER PRIMARY KEY AUTOINCREMENT,
14        name TEXT NOT NULL, due_date TEXT NOT NULL, priority INTEGER NOT NULL,
15        status INTEGER NOT NULL)""")
16
17    # insert dummy data into the table
18    c.execute('INSERT INTO ftasks (name, due_date, priority, status)
19VALUES("Finish this tutorial", "02/03/2014", 10, 1)')
20    c.execute('INSERT INTO ftasks (name, due_date, priority, status)
21VALUES("Finish Real Python Course 2", "02/03/2014", 10, 1)')

```

Two things to note:

1. Notice how we did not need to specify the “task_id” when entering (INSERT INTO command) 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? ... SQLite Browser.

Controller

Add the following code to *views.py*:

```
1 # views.py
2
3
4 from flask import Flask, flash, redirect, render_template, request, \
5     session, url_for, g
6 from functools import wraps
7 import sqlite3
8
9 app = Flask(__name__)
10 app.config.from_object('config')
11
12 def connect_db():
13     return sqlite3.connect(app.config['DATABASE'])
14
15 def login_required(test):
16     @wraps(test)
17     def wrap(*args, **kwargs):
18         if 'logged_in' in session:
19             return test(*args, **kwargs)
20         else:
21             flash('You need to login first.')
22             return redirect(url_for('login'))
23     return wrap
24
25 @app.route('/logout/')
26 def logout():
27     session.pop('logged_in', None)
28     flash('You are logged out. Bye. :(')
29     return redirect(url_for('login'))
30
31 @app.route('/', methods=['GET', 'POST'])
32 def login():
33     error = None
34     if request.method == 'POST':
```

```

35     if request.form['username'] != app.config['USERNAME'] or
request.form['password'] != app.config['PASSWORD']:
36         error = 'Invalid Credentials. Please try again.'
37     else:
38         session['logged_in'] = True
39         return redirect(url_for('tasks'))
40     return render_template('login.html', error=error)

```

Save this file in the “app” directory. You’ve seen this all before, in the last chapter. 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 a user logs in, s/he will be 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 and base templates as well as an external stylesheet.

Templates and Styles

Login template:

```

1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Welcome to FlaskTaskr.</h1>
4     <h3>Please login to access your task list.</h3>
5     <form method="post" action="/">
6         Username: <input type="text" name="username" value="{{
7             request.form.username }}">
8         Password: <input type="password" name="password" value="{{
9             request.form.password }}">
10        <input type="submit" value="Login">
11    </form>
12    <p><em>Use 'admin' for the username and password.</em></p>
13 {% endblock %}

```

Save this as *login.html* in the “templates” directory.

Base template:

```

1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Welcome to FlaskTaskr!!</title>
5         <link rel="stylesheet" href="{{ url_for('static', filename='css/styles.css')
6             }}">
7     </head>
8     <body>
9         <div class="page">
10             {% for message in get_flashed_messages() %}

```

```

10     <div class="flash">{{ message }}</div>
11     <br/>
12     {% endfor %}
13     {% if error %}
14     <div class="error"><strong>Error:</strong> {{ error }}</div>
15     {% endif %}
16     {% block content %}
17     {% endblock %}
18 </div>
19 </body>
20 </html>

```

Save this as *template.html* in the “templates” directory. Do you remember the relationship between the parent and child templates discussed in the last chapter? Read more about it [here](#).

We’ll temporarily “borrow” the majority of the stylesheet from the Flask [tutorial](#). Please copy and paste this.

```

1 body {
2     font-family: sans-serif;
3     background: #eee;
4 }
5
6 a, h1, h2 {
7     color: #377BA8;
8 }
9
10 h1, h2 {
11     font-family: 'Georgia', serif;
12     margin: 0;
13 }
14
15 h1 {
16     border-bottom: 2px solid #eee;
17 }
18
19 h2 {
20     font-size: 1.5em;
21 }
22
23 .page {
24     margin: 2em auto;
25     width: 50em;
26     border: 5px solid #ccc;
27     padding: 0.8em;
28     background: white;
29 }
30

```

```

31 .entries {
32     list-style: none;
33     margin: 0;
34     padding: 0;
35 }
36
37 .entries li {
38     margin: 0.8em 1.2em;
39 }
40
41 .entries li h2 {
42     margin-left: -1em;
43 }
44
45 .add-task {
46     font-size: 0.9em;
47     border-bottom: 1px solid #ccc;
48 }
49 .add-task dl {
50     font-weight: bold;
51 }
52
53 .metanav {
54     text-align: right;
55     font-size: 0.8em;
56     padding: 0.3em;
57     margin-bottom: 1em;
58     background: #fafafa;
59 }
60
61 .flash {
62     background: #CEE5F5;
63     padding: 0.5em;
64 }
65
66 .error {
67     background: #F0D6D6;
68     padding: 0.5em;
69 }
70
71 .datagrid table {
72     border-collapse: collapse;
73     text-align: left;
74     width: 100%;
75 }
76
77 .datagrid {

```

```

78     background: #fff;
79     overflow: hidden;
80     border: 1px solid #000000;
81     border-radius: 3px;
82 }
83
84 .datagrid table td, .datagrid table th {
85     padding: 3px 10px;
86 }
87
88 .datagrid table thead th {
89     background-color: #000000;
90     color: #FFFFFF;
91     font-size: 15px;
92     font-weight: bold;
93 }
94 .datagrid table thead th:first-child {
95     border: none;
96 }
97
98 .datagrid table tbody td {
99     color: #000000;
100    border-left: 1px solid #E1EEF4;
101    font-size: 12px;
102    font-weight: normal;
103 }
104
105 .datagrid table tbody .alt td {
106     background: #E1EEF4;
107     color: #000000;
108 }
109
110 .datagrid table tbody td:first-child {
111     border-left: none;
112 }
113
114 .datagrid table tbody tr:last-child td {
115     border-bottom: none;
116 }
117
118 .button {
119     background-color: #000;
120     display: inline-block;
121     color: #ffffff;
122     font-size: 13px;
123     padding: 3px 12px;
124     margin: 0;

```

```
125 text-decoration:none;
126 position:relative;
127 }
```

Save this as *styles.css* in the “css” directory within the static” directory.

Running our app

Lastly, create a file that we will use to run the application:

```
1 # run.py
2
3
4 from views import app
5 app.run(debug=True)
```

Save this as *run.py* in your project directory. This file is used to run our application instead of running it directly from the controller, like in the blog app from last chapter. This is to help remove unnecessary code from the controller that does not pertain to the *actual* logic that should be found in the controller.

Your project structure should now look like this:

```
1
2 config.py
3 db_create.py
4 flasktaskr.db
5 run.py
6 static
7     css
8         styles.css
9     img
10    js
11 templates
12     login.html
13     template.html
14 views.py
```

Fire up the server:

```
1 $ python run.py
```

Make sure everything works thus far. You’ll only be able to view the login page (but not login), which should be styled, as we have not setup the *tasks.html* page yet.

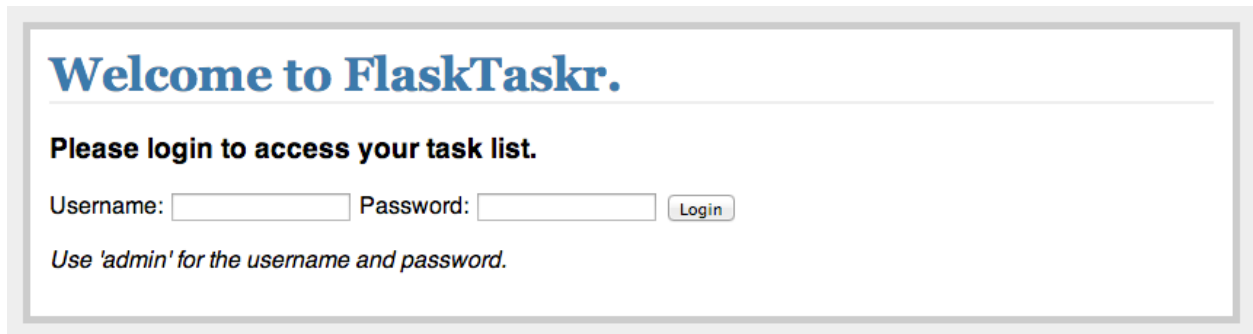


Figure 9.1: flasktaskr

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:

```
1 @app.route('/tasks/')
2 @login_required
3 def tasks():
4     g.db = connect_db()
5     cur = g.db.execute('select name, due_date, priority, task_id from ftasks
6                         where status=1')
7     open_tasks = [dict(name=row[0], due_date=row[1], priority=row[2],
8                         task_id=row[3]) for row in cur.fetchall()]
9     cur = g.db.execute('select name, due_date, priority, task_id from ftasks
10                        where status=0')
11     closed_tasks = [dict(name=row[0], due_date=row[1], priority=row[2],
12                           task_id=row[3]) for row in cur.fetchall()]
13     g.db.close()
14     return render_template('tasks.html', open_tasks=open_tasks,
15                           closed_tasks=closed_tasks)
```

We're querying the database for open and closed tasks, saving the results to two variables, *open_tasks* and *closed_tasks*, and then passing those variables 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:

```
1 # Add new tasks:
2 @app.route('/add/', methods=['POST'])
3 @login_required
4 def new_task():
5     g.db = connect_db()
```

```

6     name = request.form['name']
7     date = request.form['due_date']
8     priority = request.form['priority']
9     if not name or not date or not priority:
10         flash("All fields are required. Please try again.")
11         return redirect(url_for('tasks'))
12     else:
13         g.db.execute('insert into ftasks (name, due_date, priority, status)
values (?, ?, ?, 1)',
14             [request.form['name'], request.form['due_date'],
request.form['priority']])
15         g.db.commit()
16         g.db.close()
17         flash('New entry was successfully posted. Thanks.')
18         return redirect(url_for('tasks'))
19
20 # Mark tasks as complete:
21 @app.route('/complete/<int:task_id>/',)
22 @login_required
23 def complete(task_id):
24     g.db = connect_db()
25     cur = g.db.execute('update ftasks set status = 0 where
task_id='+str(task_id))
26     g.db.commit()
27     g.db.close()
28     flash('The task was marked as complete.')
29     return redirect(url_for('tasks'))
30
31 # Delete Tasks:
32 @app.route('/delete/<int:task_id>/',)
33 @login_required
34 def delete_entry(task_id):
35     g.db = connect_db()
36     cur = g.db.execute('delete from ftasks where task_id='+str(task_id))
37     g.db.commit()
38     g.db.close()
39     flash('The task was deleted.')
40     return redirect(url_for('tasks'))

```

The last two functions pass in a variable parameter, `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.

NOTE: This type of routing is commonly referred to as dynamic routing. Flask makes this not only incredibly powerful - but easy to implement as well. Read more about it

here.

Tasks Template:

[illegible]

```

45 <h2>Open tasks:</h2>
46 <div class="datagrid">
47   <table>
48     <thead>
49       <tr>
50         <th width="300px"><strong>Task Name</strong></th>
51         <th width="100px"><strong>Due Date</strong></th>
52         <th width="50px"><strong>Priority</strong></th>
53         <th><strong>Actions</strong></th>
54       </tr>
55     </thead>
56     {% for o in open_tasks %}
57       <tr>
58         <td width="300px">{{ o.name }}</td>
59         <td width="100px">{{ o.due_date }}</td>
60         <td width="50px">{{ o.priority }}</td>
61         <td>
62           <a href="{% url_for('delete_entry', task_id = o.task_id)
63             }}">Delete</a> -
64           <a href="{% url_for('complete', task_id = o.task_id) }}">Mark as
65             Complete</a>
66         </td>
67       </tr>
68     {% endfor %}
69   </table>
70 </div>
71 <br>
72 <div class="entries">
73   <h2>Closed tasks:</h2>
74   <div class="datagrid">
75     <table>
76       <thead>
77       <tr>
78         <th width="300px"><strong>Task Name</strong></th>
79         <th width="100px"><strong>Due Date</strong></th>
80         <th width="50px"><strong>Priority</strong></th>
81         <th><strong>Actions</strong></th>
82       </tr>
83     </thead>
84     {% for c in closed_tasks %}
85       <tr>
86         <td width="300px">{{ c.name }}</td>
87         <td width="100px">{{ c.due_date }}</td>
88         <td width="50px">{{ c.priority }}</td>
89         <td>

```

```

90         <a href="{ url_for('delete_entry', task_id = c.task_id)
    }}">Delete</a>
91     </td>
92 </tr>
93     {% endfor %}
94 </table>
95 </div>
96 </div>
97 {% endblock %}

```

Save this as *tasks.html* in the “templates” directory.

Although a lot is going on in here, the only things you have not seen before are these statements:

```

1 <a href="{ url_for('delete_entry', task_id = o.task_id) }}">Delete</a>
2 <a href="{ url_for('complete', task_id = o.task_id) }}">Mark as Complete</a>

```

Essentially, we pull the `task_id` from the database dynamically from each row in the database table as the for loop progresses (or iterates). We then assign 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>/',)`.

Make sure to walk through this app line by line. You should understand what each line is doing.

Finally, test out the functionality of the app.

Fire up the server. If you get any errors, be sure to double check your code.

Welcome to FlaskTaskR
[Logout](#)

Add a new task:

Task Name: Due Date (mm/dd/yyyy): Priority:

Open tasks:

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

Closed tasks:

Task Name	Due Date	Priority	Actions
Take a shower	3/26/2012	5	Delete

Figure 9.2: flasktaskr main

Extensions

Now that we have a functional app, let's add some features and extensions so that the application is easier to develop, as we build out additional functionality, and manage (scaling). Further, we will be looking at how best to structure, test, and deploy your app.

Specifically, we will be looking at:

- Database Management via SQLAlchemy,
- User Registration,
- User Logins,
- Database Relationships,
- Managing Sessions,
- Error Handling,
- Unit Testing,
- Functional Testing,
- Styling,
- Blueprints,
- Blueprints: Advanced,
- Conventions and Best Practices,
- Upgrade to PostgreSQL,
- Deployment on Heroku,
- Fabric, and
- Updated Workflow.

Homework

- Please read over the [mainpage](#) of the Flask-SQLAlchemy extension. Compare the code samples to regular SQL. How do the classes/objects compare to the SQL statements used for creating a new table?
- Take a look at all the Flask extensions [here](#). Read them over quickly.

Database Management via SQLAlchemy

As mentioned, 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.^[8]

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.

Setup

Start by installing Flask-SQLAlchemy in root project directory:

```
1 $ pip install Flask-SQLAlchemy==1.0
```

Delete *flasktaskr.db*, and then create a new file called *models.py* in the “app” directory. We're going to recreate the database using SQLAlchemy. As we do this, compare this method to how we created the database before, using vanilla SQL.

```
1 # models.py
2
3
4 from views import db
5
6 class FTasks(db.Model):
7
8     __tablename__ = "ftasks"
9
10    task_id = db.Column(db.Integer, primary_key=True)
11    name = db.Column(db.String, nullable=False)
12    due_date = db.Column(db.Date, nullable=False)
13    priority = db.Column(db.Integer, nullable=False)
14    status = db.Column(db.Integer)
15
16    def __init__(self, name, due_date, priority, status):
17        self.name = name
18        self.due_date = due_date
```

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

```

19         self.priority = priority
20         self.status = status
21
22     def __repr__(self):
23         return '<name %r>' % (self.body)

```

We have one class, `FTasks()`, that defined the `ftasks` table. The variable names are used as the column names. *Any field that has a `primary_key` set to `True` will auto-increment.*

Update the imports and configuration section in `views.py`:

```

1 from flask import Flask, flash, redirect, render_template, request, \
2     session, url_for, g
3 from functools import wraps
4 from flask.ext.sqlalchemy import SQLAlchemy
5
6 app = Flask(__name__)
7 app.config.from_object('config')
8 db = SQLAlchemy(app)

```

Make sure to remove these from `views.py` since we are not using the Python sqlite wrapper to interact with the database anymore:

```

1 import sqlite3
2 def connect_db():
3     return sqlite3.connect(app.config['DATABASE'])

```

Your `views.py` file should now look like this:

```

1 # views.py
2
3
4 from flask import Flask, flash, redirect, render_template, request, \
5     session, url_for, g
6 from functools import wraps
7 from flask.ext.sqlalchemy import SQLAlchemy
8
9 app = Flask(__name__)
10 app.config.from_object('config')
11 db = SQLAlchemy(app)
12
13 from models import FTasks
14
15 def login_required(test):
16     @wraps(test)
17     def wrap(*args, **kwargs):
18         if 'logged_in' in session:
19             return test(*args, **kwargs)

```



```

20         else:
21             flash('You need to login first.')
22             return redirect(url_for('login'))
23     return wrap
24
25 @app.route('/logout/')
26 def logout():
27     session.pop('logged_in', None)
28     flash('You are logged out. Bye. :(')
29     return redirect(url_for('login'))
30
31 @app.route('/', methods=['GET', 'POST'])
32 def login():
33     error = None
34     if request.method == 'POST':
35         if request.form['username'] != app.config['USERNAME'] or
36            request.form['password'] != app.config['PASSWORD']:
37             error = 'Invalid Credentials. Please try again.'
38         else:
39             session['logged_in'] = True
40             return redirect(url_for('tasks'))
41     return render_template('login.html', error=error)
42
43 @app.route('/tasks/')
44 @login_required
45 def tasks():
46     open_tasks = db.session.query(FTasks).filter_by(status='1').order_by(
47         FTasks.due_date.asc())
48     closed_tasks = db.session.query(FTasks).filter_by(status='0').order_by(
49         FTasks.due_date.asc())
50     return render_template('tasks.html', form = AddTask(request.form),
51                            open_tasks=open_tasks, closed_tasks=closed_tasks)
52
53 # Add new tasks:
54 @app.route('/add/', methods=['GET', 'POST'])
55 @login_required
56 def new_task():
57     form = AddTask(request.form, csrf_enabled=False)
58     if form.validate_on_submit():
59         new_task = FTasks(
60             form.name.data,
61             form.due_date.data,
62             form.priority.data,
63             '1'
64         )
65         db.session.add(new_task)
66         db.session.commit()

```

```

66         flash('New entry was successfully posted. Thanks.')
67         return redirect(url_for('tasks'))
68
69 # Mark tasks as complete:
70 @app.route('/complete/<int:task_id>/',)
71 @login_required
72 def complete(task_id):
73     new_id = task_id
74     db.session.query(FTasks).filter_by(task_id=new_id).update({"status": "0"})
75     db.session.commit()
76     flash('The task was marked as complete. Nice.')
77     return redirect(url_for('tasks'))
78
79 # Delete Tasks:
80 @app.route('/delete/<int:task_id>/',)
81 @login_required
82 def delete_entry(task_id):
83     new_id = task_id
84     db.session.query(FTasks).filter_by(task_id=new_id).delete()
85     db.session.commit()
86     flash('The task was deleted. Why not add a new one?')
87     return redirect(url_for('tasks'))

```

Pay attention to the differences in the `new_task()`, `complete()`, and `delete_entry()` functions. How are they structured differently than before when we used straight (or vanilla) SQL instead?

Also, update your `config.py` file:

```

1 # config.py
2
3
4 import os
5
6 # grabs the folder where the script runs
7 basedir = os.path.abspath(os.path.dirname(__file__))
8
9 DATABASE = 'flasktaskr.db'
10 USERNAME = 'admin'
11 PASSWORD = 'admin'
12 SECRET_KEY = 'my_precious'
13
14 # defines the full path for the database
15 DATABASE_PATH = os.path.join(basedir, DATABASE)
16
17 # the database uri
18 SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' + DATABASE_PATH

```

Here we're defining the `SQLALCHEMY_DATABASE_URI` to tell SQLAlchemy where to access the database. Confused about `os.path.join`? Read about it [here](#).

Lastly, update `db_create.py`.

```
1 # db_create.py
2
3
4 from views import db
5 from models import FTasks
6 from datetime import date
7
8 # create the database and the db table
9 db.create_all()
10
11 # insert data
12 db.session.add(FTasks("Finish this tutorial", date(2014, 3, 13), 10, 1))
13 db.session.add(FTasks("Finish Real Python", date(2014, 3, 13), 10, 1))
14
15 # commit the changes
16 db.session.commit()
```

1. We initialize the database schema by calling `db.create_all()`.
2. We then populate it with some data. We use the `FTASKS` object from `*models.py` to specify the data.
3. To apply the inserts to our database we need to commit using `db.session.commit()`

Since we are now using SQLAlchemy, we're modifying the way we do database queries. The code is much cleaner. Take a look Compare it with the actual SQL code from the beginning of the chapter.

Create the database

Save all the files, and run the script:

```
1 $ python db_create.py
```

The `flasktaskr.db` should have been recreated. Open up the file in SQLite Browser to ensure that the table and the above data are present in the `ftasks` table.

Try to run your application. Login. You should see this error-

```
1 NameError: global name 'AddTask' is not defined
```

-since we have not defined this class anywhere.

With that, let's do just that.

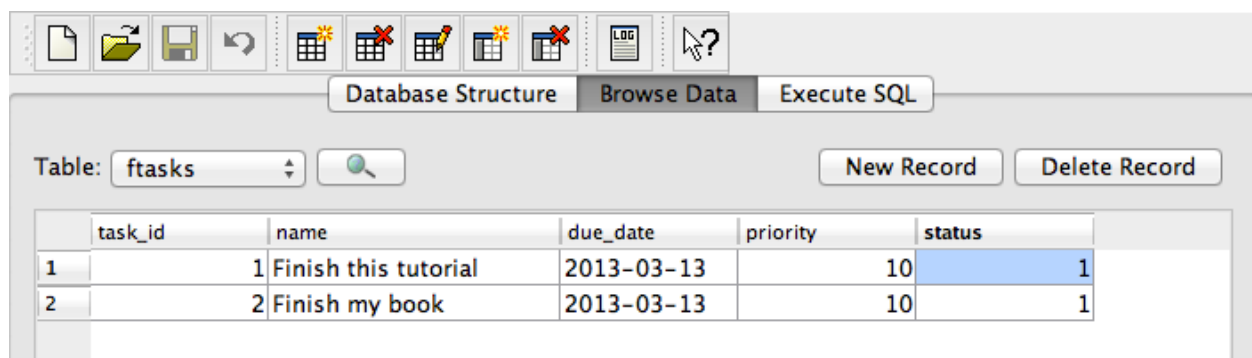


Figure 9.3: flasktaskr sql

User Registration

We're now going to use another powerful Flask extension called WTForms to help with form handling and data validation.²

Add Tasks form

First, install the package. Make sure virtualenv is activated.

```
1 pip install Flask-WTF==0.9.4
```

Now let's create a new file called *forms.py* and add the following code:

```
1 # forms.py
2
3 from flask_wtf import Form
4 from wtforms import TextField, DateField, IntegerField, \
5     SelectField
6 from wtforms.validators import DataRequired
7
8 class AddTask(Form):
9     task_id = IntegerField('Priority')
10    name = TextField('Task Name', validators=[DataRequired()])
11    due_date = DateField('Date Due (mm/dd/yyyy)', validators=[DataRequired()],
12                       format='%m/%d/%Y')
13    priority = SelectField('Priority', validators=[DataRequired()],
14                          choices=[('1', '1'), ('2', '2'), ('3', '3'),
15                                   ('4', '4'), ('5', '5')])
16    status = IntegerField('Status')
```

Notice how we're importing from both Flask-WTF and WTForms. Essentially, Flask-WTF integration with WTForms.

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.

Make sure you update your *views.py* by importing the AddTask() class from *forms.py*:

```
1 from forms import AddTask
```

Fire up your server. You should be able to login now. Ensure that you can still view tasks, add new tasks, mark tasks as completed, and delete tasks.

Let's allow multiple users to access the task manager by setting up a user registration form.

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

Create a new table

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*:

```
1 class User(db.Model):
2
3     __tablename__ = 'users'
4
5     id = db.Column(db.Integer, primary_key=True)
6     name = db.Column(db.String, unique=True, nullable=False)
7     email = db.Column(db.String, unique=True, nullable=False)
8     password = db.Column(db.String, nullable=False)
9
10    def __init__(self, name=None, email=None, password=None):
11        self.name = name
12        self.email = email
13        self.password = password
14
15    def __repr__(self):
16        return '<User %r>' % (self.name)
```

Run *db_create.py* again. Before you do so, commit out the following lines:

```
1 db.session.add(FTasks("Finish this tutorial", date(2014, 3, 13), 10, 1))
2 db.session.add(FTasks("Finish Real Python", date(2014, 3, 13), 10, 1))
```

If you do not do this, it will add that data to the database again.

Open up SQLite Browser. Notice how it ignores (e.g, does not meddle with) the table already created, *ftasks*, and just creates the *users* table:

Configuration

Update the configuration module, *config.py*:

Remove the following lines of code:

```
1 USERNAME = 'admin'
2 PASSWORD = 'admin'
```

We no longer need this configuration since we will use the information from the *users* table in the database instead of this hard-coded data.

We also need to update *forms.py* to cater for both user registration and logging in:

```
1 # forms.py
2
```

Name	Object	Type
▼tasks	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	

Figure 9.4: flask new table

```

3
4 from flask_wtf import Form
5 from wtforms import TextField, DateField, IntegerField, \
6     SelectField, PasswordField
7 from wtforms.validators import DataRequired, Email, EqualTo, Length
8
9 class RegisterForm(Form):
10     name = TextField('Username', validators=[DataRequired(), Length(min=6,
11     max=25)])
12     email = TextField('Email', validators=[DataRequired(), Length(min=6,
13     max=40)])
14     password = PasswordField('Password', validators=[DataRequired(),
15     Length(min=6, max=40)])
16     confirm = PasswordField('Repeat Password', [DataRequired(),
17     EqualTo('password', message='Passwords must match')])
18
19 class LoginForm(Form):
20     name = TextField('Username', validators=[DataRequired()])
21     password = PasswordField('Password', validators=[DataRequired()])
22
23 class AddTask(Form):
24     task_id = IntegerField('Priority')
25     name = TextField('Task Name', validators=[DataRequired()])
26     due_date = DateField('Date Due (mm/dd/yyyy)', validators=[DataRequired()],
27         format='%m/%d/%Y')
28     priority = SelectField('Priority', validators=[DataRequired()])

```

```

25             choices=[('1', '1'), ('2', '2'), ('3', '3'),
26                       ('4', '4'), ('5', '5')])
27     status = IntegerField('Status')

```

Next we need to update the Controller, *views.py*:

1. Update the imports, and add the following code:

```

1  # ---- imports and configuration ---- #
2
3  from flask import Flask, flash, redirect, render_template, request, \
4      session, url_for, g
5  from functools import wraps
6  from flask.ext.sqlalchemy import SQLAlchemy
7  from forms import AddTask, RegisterForm, LoginForm
8
9  app = Flask(__name__)
10 app.config.from_object('config')
11 db = SQLAlchemy(app)
12
13 from models import FTasks, User

```

This allow us to have access to, and use, the RegisterForm() and LoginForm() classes from *forms.py* and the User() class from *models.py*. “

2. Add the new view function, register():

```

1  @app.route('/register/', methods=['GET', 'POST'])
2  def register():
3      error = None
4      form = RegisterForm(request.form, csrf_enabled=False)
5      if form.validate_on_submit():
6          new_user = User(
7              form.name.data,
8              form.email.data,
9              form.password.data,
10             )
11         db.session.add(new_user)
12         db.session.commit()
13         flash('Thanks for registering. Please login.')
14         return redirect(url_for('login'))
15     return render_template('register.html', form=form, error=error)

```

Here, the user information obtained from the *register.html* template (which we still need to create) 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:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Welcome to FlaskTaskr.</h1>
4     <h3>Please register to access the task list.</h3>
5     <form method="POST" action="">
6         <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{ form.email.label
7         }}: {{ form.email }}</p>
8         <p>{{ form.password.label }}: {{ form.password }}&nbsp;&nbsp;&nbsp;{{
9         form.confirm.label }}: {{ form.confirm }}</p>
10        <p><input type="submit" value="Register"></p>
11    </form>
12    <p><em>Already registered?</em> Click <a href="/">here</a> to login.</p>
13 {% endblock %}
```

Save this as *register.html* under your “templates” directory.

Now let’s add a registration link to the bottom of the *login.html* page:

```
1 <br>
2 <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.

User Login

The next step for allowing multiple users to login is to change the `login()` function within the controllers as well as the login template.

Controller

Replace the current `login()` function with:

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     if request.method=='POST':
5         u = User.query.filter_by(name=request.form['name'],
6                                 password=request.form['password']).first()
7         if u is None:
8             error = 'Invalid username or password.'
9         else:
10            session['logged_in'] = True
11            flash('You are logged in. Go Crazy.')
12            return redirect(url_for('tasks'))
13    return render_template("login.html",
14                           form = LoginForm(request.form),
15                           error = error)
```

This code is not too much different from 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:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Welcome to FlaskTaskr.</h1>
4     <h3>Please login to access your task list.</h3>
5     <form method="post" action="/">
6         <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{
7             form.password.label }}: {{ form.password }}&nbsp;&nbsp;&nbsp;<input type="submit"
8             value="Submit"></p>
9     </form>
10    <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
11 {% endblock %}
```

Test it out. Try logging in with the same user you registered. If done correctly, you should be logged in and then redirected to *tasks.html*. Check out the server logs:

```
1 127.0.0.1 - - [18/Feb/2014 21:02:19] "POST / HTTP/1.1" 302 -  
2 127.0.0.1 - - [18/Feb/2014 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.

Database Relationships

To complete the conversion to SQLAlchemy we need to update both the tasks and views templates.

First, let's update the database to add two new fields: "posted_date" and "user_id" to the *ftasks* table. The *user_id* field also needs to link back to the User table.

Database relationships

We briefly touched on the subject of relationally linking tables in the chapter on SQL, but essentially relational databases are designed to connect tables together using unique fields. This is why they are so powerful. 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:

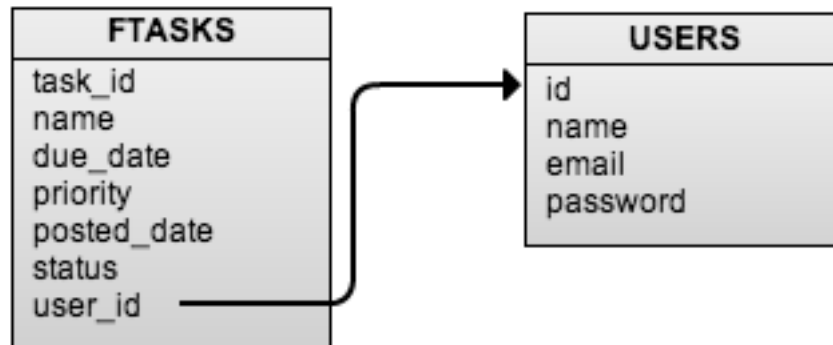


Figure 9.5: sql relationships

Let's look at how to alter the tables (commonly referred to as a migration) to create such relationships within *models.py*.

Add the following field to the "ftasks" table-

```
1 user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

-and this field to the "users" table:

```
1 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 corresponding "id" field of the "users" table. Foreign keys are essential for creating relationships between tables in order to correlate information.

SEE ALSO: Need help with foreign keys? Take a look at the W3 [documentation](#).

Further, in a relational database there are three basic relationships:

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:

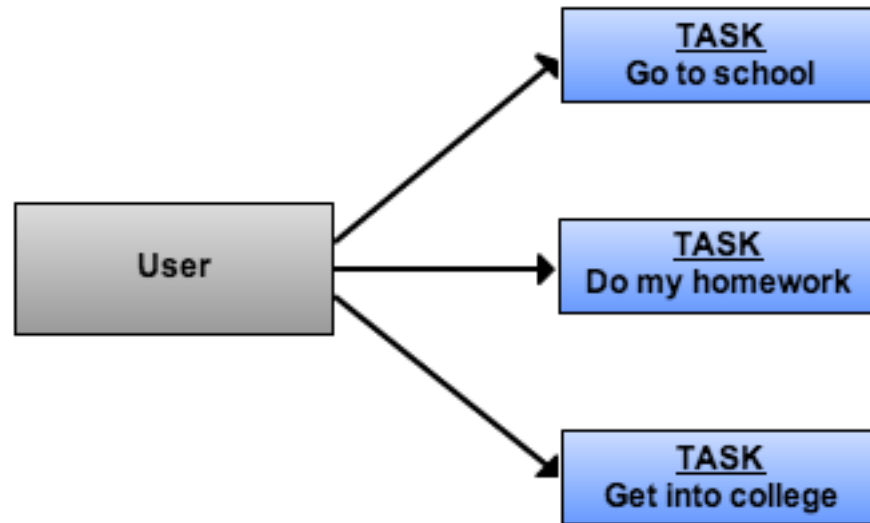


Figure 9.6: erd schema

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.

The `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:

```
1 # models.py
2
3
4 from views import db
5
6 class FTasks(db.Model):
7
8     __tablename__ = "ftasks"
9
```

```

10 task_id = db.Column(db.Integer, primary_key=True)
11 name = db.Column(db.String, nullable=False)
12 due_date = db.Column(db.Date, nullable=False)
13 priority = db.Column(db.Integer, nullable=False)
14 posted_date = db.Column(db.Date, nullable=False)
15 status = db.Column(db.Integer)
16 user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
17
18 def __init__(self, name, due_date, priority, posted_date, status, user_id):
19     self.name = name
20     self.due_date = due_date
21     self.priority = priority
22     self.posted_date = posted_date
23     self.status = status
24     self.user_id = user_id
25
26 def __repr__(self):
27     return '<name %r>' % (self.body)
28
29 class User(db.Model):
30
31     __tablename__ = 'users'
32
33     id = db.Column(db.Integer, primary_key=True)
34     name = db.Column(db.String, unique=True, nullable=False)
35     email = db.Column(db.String, unique=True, nullable=False)
36     password = db.Column(db.String, nullable=False)
37     tasks = db.relationship('FTasks', backref='poster')
38
39
40 def __init__(self, name=None, email=None, password=None):
41     self.name = name
42     self.email = email
43     self.password = password
44
45 def __repr__(self):
46     return '<User %r>' % (self.name)

```

The above code will work only 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:

```

1 # db_migrate.py
2
3
4 from views import db
5 from datetime import datetime

```

```

6 from config import DATABASE_PATH
7 import sqlite3
8
9 with sqlite3.connect(DATABASE_PATH) as connection:
10
11     # get a cursor object used to execute SQL commands
12     c = connection.cursor()
13
14     # temporarily change the name of ftasks table
15     c.execute("""ALTER TABLE ftasks RENAME TO old_ftasks""")
16
17     # recreate a new ftasks table with updated schema
18     db.create_all()
19
20     # retrieve data from old_ftasks table
21     c.execute("""SELECT name, due_date, priority,
22                 status FROM old_ftasks ORDER BY task_id ASC""")
23
24     # save all rows as a list of tuples; set posted_date to now and user_id to 1
25     data = [(row[0], row[1], row[2], row[3],
26              datetime.now(), 1) for row in c.fetchall()]
27
28     # insert data to ftasks table
29     c.executemany("""INSERT INTO ftasks (name, due_date, priority, status,
30                                     posted_date, user_id) VALUES (?, ?, ?, ?, ?, ?)""", data)
31
32     # delete old_ftasks table
33     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.

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:

```

1 posted_date = DateField('Posted Date (mm/dd/yyyy)', validators=[DataRequired()],
    format='%m/%d/%Y')

```

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	

Figure 9.7: updated schema

Controller

We also need to update our *views.py* for adding new tasks. Within the *new_task* function, change the following:

```

1 new_task = FTasks(
2     form.name.data,
3     form.due_date.data,
4     form.priority.data,
5     '1'
6 )

```

to:

```

1 new_task = FTasks(
2     form.name.data,
3     form.due_date.data,
4     form.priority.data,
5     form.posted_date.data,
6     '1',
7     '1'
8 )

```

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 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. However, what happens if there is more than one user? Later, in a 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:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Welcome to FlaskTaskr</h1>
4     <br>
5     <a href="/logout">Logout</a>
6     <div class="add-task">
7         <h3>Add a new task:</h3>
8         <form action="{% url_for('new_task') %}" method="post">
9             <p>{{ form.name.label }}: {{ form.name }}<br>{{ form.due_date.label }}: {{ form.due_date }}&nbsp;{{ form.posted_date.label }}: {{ form.posted_date }}&nbsp;{{ form.priority.label }}: {{ form.priority }}</p>
10            <p><input type="submit" value="Submit"></p>
11        </form>
12    </div>
13    <div class="entries">
14        <br>
15        <br>
16        <h2>Open tasks:</h2>
17        <div class="datagrid">
18            <table>
19                <thead>
20                    <tr>
21                        <th width="200px"><strong>Task Name</strong></th>
22                        <th width="75px"><strong>Due Date</strong></th>
23                        <th width="100px"><strong>Posted Date</strong></th>
24                        <th width="50px"><strong>Priority</strong></th>
25                        <th width="90px"><strong>Posted By</strong></th>
26                        <th><strong>Actions</strong></th>
27                    </tr>
28                </thead>
29                {% for o in open_tasks %}
30                    <tr>
31                        <td width="200px">{{ o.name }}</td>
32                        <td width="75px">{{ o.due_date }}</td>
33                        <td width="100px">{{ o.posted_date }}</td>
34                        <td width="50px">{{ o.priority }}</td>
35                        <td width="90px">{{ o.poster.name }}</td>
36                        <td>
37                            <a href="{% url_for('delete_entry', task_id = o.task_id) %}">Delete</a> -
38                            <a href="{% url_for('complete', task_id = o.task_id) %}">Mark as
```

```

39     Complete</a>
40         </td>
41     </tr>
42     {% endfor %}
43 </table>
44 </div>
45 <br>
46 <br>
47 <div class="entries">
48     <h2>Closed tasks:</h2>
49     <div class="datagrid">
50         <table>
51             <thead>
52                 <tr>
53                     <th width="200px"><strong>Task Name</strong></th>
54                     <th width="75px"><strong>Due Date</strong></th>
55                     <th width="100px"><strong>Posted Date</strong></th>
56                     <th width="50px"><strong>Priority</strong></th>
57                     <th width="90px"><strong>Posted By</strong></th>
58                     <th><strong>Actions</strong></th>
59                 </tr>
60             </thead>
61             {% for c in closed_tasks %}
62                 <tr>
63                     <td width="200px">{{ c.name }}</td>
64                     <td width="75px">{{ c.due_date }}</td>
65                     <td width="100px">{{ c.posted_date }}</td>
66                     <td width="50px">{{ c.priority }}</td>
67                     <td width="90px">{{ c.poster.name }}</td>
68                     <td>
69                         <a href="{ url_for('delete_entry', task_id = c.task_id)
70                         }}">Delete</a>
71                     </td>
72                 </tr>
73             {% endfor %}
74         </table>
75     </div>
76 </div>

```

The changes are fairly straightforward. Can you find them? Take a look at this file along with *forms.py* to see how the drop-down list is implemented.

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*, as expected.

Let's correct that.

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

Figure 9.8: one user

Managing Sessions

Do you remember the relationship we established between the two tables in the last lesson?

```
1 user_id = Column(Integer, ForeignKey('users.id'))
2 tasks = relationship('FTASKS', backref = 'poster')
```

Well, with that simple relationship, 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`:

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     error = None
4     if request.method=='POST':
5         u = User.query.filter_by(name=request.form['name'],
6                                   password=request.form['password']).first()
7         if u is None:
8             error = 'Invalid username or password.'
9         else:
10            session['logged_in'] = True
11            session['user_id'] = u.id
12            flash('You are logged in. Go Crazy.')
13            return redirect(url_for('tasks'))
14
15    return render_template("login.html",
16                           form = LoginForm(request.form),
17                           error = error)
```

Next, when we post a new task, we need to grab that user id and add it to the SQLAlchemy ORM query - so update the `new_task` function:

```
1 @app.route('/add/', methods=['GET', 'POST'])
2 @login_required
3 def new_task():
4     form = AddTask(request.form, csrf_enabled=False)
5     if form.validate_on_submit():
6         new_task = FTasks(
7             form.name.data,
8             form.due_date.data,
9             form.priority.data,
10            form.posted_date.data,
11            '1',
12            session['user_id']
13        )
14        db.session.add(new_task)
15        db.session.commit()
```

```
16     flash('New entry was successfully posted. Thanks.')
17     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:

```
1 @app.route('/logout/')
2 def logout():
3     session.pop('logged_in', None)
4     session.pop('user_id', None)
5     flash('You are logged out. Bye. :(')
6     return redirect(url_for('login'))
```

Now open up *tasks.html*. In each of the two for loops, note these statements:

```
1 <td width="90px">{{ o.poster.name }}</td>
2 ...
3 <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 (which we will be doing later - yay!). 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.

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 - and nothing does. Literally. Obviously this is very confusing for end users. Thus, we need to add in an error message in order to provide good feedback to our users. 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:

```
1 class RegisterForm(Form):
2     name = TextField('Username', validators=[DataRequired(), Length(min=6,
3     max=25)])
4     email = TextField('Email', validators=[DataRequired(), Length(min=6,
5     max=40)])
6     password = PasswordField('Password', validators=[DataRequired(),
7     Length(min=6, max=40)])
8     confirm = PasswordField('Repeat Password', [DataRequired(),
9     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:

```
1 def flash_errors(form):
2     for field, errors in form.errors.items():
3         for error in errors:
4             flash(u"Error in the %s field - %s" % (
5                 getattr(form, field).label.text, error), 'error')
```

Then add an else statement to each view that has form validation in it:

```
1 if form.validate_on_submit():
2     ###Do Something###
3     .....
4 else:
5     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 the Controller, *views.py*:

```
1 from sqlalchemy.exc import IntegrityError
```

2. Then update the `register()` function:

```
1 @app.route('/register/', methods=['GET', 'POST'])
2 def register():
3     error = None
4     form = RegisterForm(request.form, csrf_enabled=False)
5     if form.validate_on_submit():
6         new_user = User(
7             form.name.data,
8             form.email.data,
9             form.password.data,
10            )
11         try:
12             db.session.add(new_user)
13             db.session.commit()
14             flash('Thanks for registering. Please login.')
15             return redirect(url_for('login'))
16         except IntegrityError:
17             error = 'Oh no! That username and/or email already exist.
18             Please try again.'
19             return render_template('register.html', form=form, error=error)
20         else:
21             flash_errors(form)
22             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 stops 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:

```
1 app.run(debug=True)
```

Just change `debug` to `False` to disable it. Do that now. You can always turn it back on if you encounter errors while developing.

Custom Error Pages

Now go back and comment out the `try/except` blocks within the `register()` function that you just set up:

```
1 @app.route('/register/', methods=['GET', 'POST'])
2 def register():
3     error = None
4     form = RegisterForm(request.form, csrf_enabled=False)
5     if form.validate_on_submit():
6         new_user = User(
7             form.name.data,
8             form.email.data,
9             form.password.data,
10            )
11         # try:
12         #     db.session.add(new_user)
13         #     db.session.commit()
14         #     flash('Thanks for registering. Please login.')
15         #     return redirect(url_for('login'))
16         # except IntegrityError:
17         #     error = 'Oh no! That username and/or email already exist. Please
18         try again.'
19         #     return render_template('register.html', form=form, error=error)
20     else:
21         flash_errors(form)
22         return render_template('register.html', form=form, error=error)
```

Register a user with a duplicate name.

You should see an Internal Server Error, which is also known as a 500 error. This is a fairly common error. Another common error is the annoying 404 Page Not Found. To see that error, simply navigate to a route we haven't set up, like <http://localhost:5000/flask>. Boom! 404.

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 using the `app.errorhandler(code)` decorator. You set these up just like any other view:

```
1 @app.errorhandler(500)
2 def internal_error(error):
3     db.session.rollback()
4     return render_template('500.html'), 500
5
6 @app.errorhandler(404)
7 def internal_error(error):
8     return render_template('404.html'), 404
```

Then just set up a couple of templates:

Save the following as *404.html*:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Sorry ...</h1>
4     <p>There's nothing here!</p>
5     <p><a href="{{url_for('login')}}">Back</a></p>
6 {% endblock %}
```

Save the following as *500.html*:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Something's wrong!</h1>
4     <p>Fortunately we are on the job, and you can just return to the login
5     page!</p>
6     <p><a href="{{url_for('login')}}">Back</a></p>
7 {% endblock %}
```

Let's see our error handlers in action. Run the server.

Try to register a duplicate username/email. You should be redirected to the *500.html* template. Then try this route that does not exist <http://localhost:5000/flask>.

Easy, right?

Set them up for other common errors like 403 and 410 as well. Don't forget to to uncomment the try/except pair from the `register()` function before moving on.

Unit Testing

Conducting unit tests on your application is an absolute necessity, especially as your app grows in size and complexity. Tests help ensure that as the complexity of our application grows the various moving parts continue to work together in a harmonious fashion.

Tests help reveal:

1. When code isn't working,
2. What code is broken, and
3. Why we wrote this code in the first place.

Every time we go to add a feature to our application, fix a bug or change some code we should make sure our code is adequately covered by tests and that the tests all pass after we're done.

In many cases, you actually define the tests before we write any code. This is called Test Driven, or Test First, Development. 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 creativity, so be careful with being overly attached to the notion of writing every single test case first. Give yourself permission to explore.

Although there is an official Flask extension called Flask-Testing to perform tests, I recommend using the pre-installed unit test package that comes with Python, aptly named unittest. ³ It's just as easy to use.

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:

```
1 import unittest
2
3 class TestCase(unittest.TestCase):
4
5     # place your test functions here
6
7 if __name__ == '__main__':
8     unittest.main()
```

Let's create a base unit test script:

```
1 # test.py
```

³<http://south.aeracode.org/>

```

2
3
4 import os
5 import unittest
6
7 from views import app, db
8 from models import User
9 from config import basedir
10
11 TEST_DB = 'test.db'
12
13 class Users(unittest.TestCase):
14
15     # this is a special method that is executed prior to each test
16     def setUp(self):
17         app.config['TESTING'] = True
18         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
19             os.path.join(basedir, TEST_DB)
20         self.app = app.test_client()
21         db.create_all()
22
23     # this is a special method that is executed after each test
24     def tearDown(self):
25         db.drop_all()
26
27     # each test should start with 'test'
28     def test_user_setup(self):
29         new_user = User("mherman", "michael@mherman.org", "michaelherman")
30         db.session.add(new_user)
31         db.session.commit()
32
33 if __name__ == "__main__":
34     unittest.main()

```

Save it as *test.py* in the root directory and run it:

```

1 $ python test.py
2 .
3 -----
4 Ran 1 test in 0.223s
5
6 OK

```

It passed!

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, etc.).
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 a second time.

```
1 IntegrityError: (IntegrityError) column email is not unique u'INSERT INTO users
   (name, email, password) VALUES (?, ?, ?)' ('mherman', 'michael@mherman.org',
   'michaelherman')
2
3 -----
4 Ran 1 test in 0.047s
5
6 FAILED (errors=1)
```

What happened?

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`).

Delete `test.db` before moving on.

Assert

Each test should have an `assert()` method to either verify an expected result or a condition, or indicate that an exception is raised.

Let's quickly look at an example of how `assert` works. Update `test.py` with the following code:

```
1 # test.py
2
3
4 import os
5 import unittest
6
7 from views import app, db
8 from models import User
9 from config import basedir
10
```

```

11 TEST_DB = 'test.db'
12
13 class Users(unittest.TestCase):
14
15     # this is a special method that is executed prior to each test
16     def setUp(self):
17         app.config['TESTING'] = True
18         app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + \
19             os.path.join(basedir, TEST_DB)
20         self.app = app.test_client()
21         db.create_all()
22
23     # this is a special method that is executed after to each test
24     def tearDown(self):
25         db.drop_all()
26
27     # each test should start with 'test'
28     def test_users_can_register(self):
29         new_user = User("mherman", "michael@mherman.org", "michaelherman")
30         db.session.add(new_user)
31         db.session.commit()
32         test = db.session.query(User).all()
33         for t in test:
34             t.name
35             assert t.name == "mherman"
36
37 if __name__ == "__main__":
38     unittest.main()

```

In this example, we're testing the same thing: whether a new user is successfully added to the database. We then pull all the data from the database, `test = db.session.query(User).all()`, 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. It should pass.

Then change the assert statement to -

```

1 assert t.name != "mherman"

```

Now you can see what an assertion error looks like:

```

1 =====
2 FAIL: test_user_setup (__main__.AddUser)
3 -----
4 Traceback (most recent call last):
5   File "test.py", line 35, in test_user_setup
6     assert t.name != "mherman"

```

```
7 AssertionError
8
9 -----
10 Ran 1 test in 0.641s
11
12 FAILED (failures=1)
```

Change the assert statement back to `assert t.name == "mherman"`.

Test again to make sure it works.

One test done. Many to go. We'll look at the remaining tests in another chapter. For now, though, let's take a break from both testing and Flask and look at the basics of HTML and CSS.

Chapter 10

Interlude: Introduction to HTML and CSS

Let's take a quick break from Flask to cover HTML and CSS ...

This is a two part tutorial covering HTML, CSS, JavaScript, and jQuery, where we will be building a basic todo list. In this first part we will be addressing HTML and CSS.

NOTE: This is a beginner tutorial. If you already have a basic understanding of HTML and CSS, please feel free to skip.

Websites are made up of many things, but HTML (Hyper Text Markup Language) and CSS (Cascading Style Sheets) are two of the most important components. Together, they are the building blocks for every single webpage on the Internet.

Think of a car. It, too, is made up of many attributes. Doors. Windows. Tires. Seats. In the world of HTML, these are the `elements` of a webpage. Meanwhile, each of the car's attributes are usually different. Perhaps they differ by size. Or color. Or wear and tear. These attributes are used to define how the `elements` look. Back in the world of webpages, CSS is used to define the look and feel of a webpage.

Now let's turn to an actual web page ..

HTML

HTML gives a web pages structure, allowing you to view it from a web browser.

Start by adding some basic structure:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4   </head>
5   <body>
6   </body>
7 </html>
```

Copy and paste this basic webpage structure into your text editor. Save this file as *index.html*.

This structure is commonly referred to as a boilerplate template. Such templates are used to speed up development so you don't have to code the features common to every single webpage each time you create a new page. Most boilerplates include more features (or boilerplate code), but let's start with the basics.

What's going on?

1. The first line, `<!DOCTYPE html>` is the document type declaration, which tells the browser the version of HTML the page is using (HTML5, in our case). Without this, browsers can get confused, especially older versions of Internet Explorer.
2. `<html>` is the first tag and it informs the browser that all code between the opening and closing, `</html>`, tags is HTML.
3. The `<head>` tag contains links to CSS stylesheets and Javascript files that we wish to use in our web page, as well as meta information used by search engines for classification. In the above HTML, I used the `<title>` tag to give the web page a title.
4. All code that falls within the `<body>` tags are part of the main content of the page, which will appear in the browser to the end user.

This is how a standard HTML page, following HTML5 standards, is structured.

Let's add four tags:

1. title `<title>`
2. heading `<h1>`
3. break `
`
4. paragraph `<p>`

```
1 <!DOCTYPE html>
2 <html>
```



```

3  <head>
4    <title>My Todo List</title>
5  </head>
6  <body>
7    <h1>My Todo List</h1>
8    <p>Get yourself organized!</p>
9    <br>
10 </body>
11 </html>

```

Elements, Tags, and Attributes

1. Tags form the structure of your page. They surround and apply *meaning* to content. There usually is an opening tag and then a closing tag, like - <h1></h1>, a heading. Some tags, like the
 (line break) tag do not require a closing tag.
2. Elements represent the tags as well as whatever falls between the opening and closing tags, like - <title>My Todo List</title>
3. Attributes (sometimes referred to as selectors) are key-value pairs used to select the tag for either applying Javascripts or CSS styles. Selectors in most cases are either ids or [classes](#).

Mozilla has an excellent reference guide [here](#) for all HTML elements.

Additional Tags

Let's add some more tags.

```

1  <!doctype html>
2  <html>
3    <head>
4      <title>My Todo List</title>
5    </head>
6    <body>
7
8      <h1>My Todo List</h1>
9      <p>Get yourself organized!</p>
10     <br>
11     <form>
12       <input type="text" placeholder="Enter a todo...">
13       <br>
14     </form>
15     <button>Submit!</button>
16     <br>
17
18   </body>
19 </html>

```

Essentially, we added - 1. A form (`<form>`), with one input, for entering a todo. 2. A button (`<button>`) that's used for submitting the entered todo.

Check it out in your browser. Kind of bland, to put it nicely. Fortunately, we can quickly change that with CSS!

On to CSS ..

CSS

While HTML provides, structure, CSS is used for styling, making webpages look nice. From the size of the text to the background colors to the positioning of HTML elements, CSS gives you control over almost every visual aspect of a page.

CSS and HTML work in tandem. CSS styles (or rules) are applied directly to HTML elements, as you will soon see.

NOTE: There are three ways that you can assign styles to HTML tags. Inline. Internal. Or External. Inline styles are placed directly in the tag; these should be avoided, though, as it's best practice to keep HTML and CSS styles separated (don't mix structure with presentation!). Internal styles fall within the head of a website. Again, these should be avoided as well due to reasons mentioned before. Read more about this [here](#).

First, we need to "link" our HTML page and CSS stylesheet. To do so, add the following code to the <head> section of the HTML page just below the title:

```
1 <link rel="stylesheet" type="text/css"
  href="http://netdna.bootstrapcdn.com/bootswatch/3.0.3/flatly/bootstrap.min.css">
2 <link rel="stylesheet" type="text/css" href="main.css">
```

Your code should now look like this:

```
1 <!doctype html>
2 <html>
3   <head>
4     <link rel="stylesheet" type="text/css"
      href="http://netdna.bootstrapcdn.com/bootswatch/3.0.3/flatly/bootstrap.min.css">
5     <link rel="stylesheet" type="text/css" href="main.css">
6     <title>My Todo List</title>
7   </head>
8   <body>
9
10    <div class="container">
11      <h1>My Todo List</h1>
12      <p>Get yourself organized!</p>
13      <br>
14      <form>
15        <input type="text" placeholder="Enter a todo...">
16        <br>
17      </form>
18      <button>Submit!</button>
19      <br>
20    </div>
21
22  </body>
```

23 </html>

Save the file. We are using two stylesheets. The first is a [bootstrap](#) stylesheet, while the second is a custom stylesheet, which we will create in a few moments. For more information on Bootstrap, please see [this](#) blog post.

Check the page out in your browser. See the difference? Yes, it's subtle - but the font is different along with the style of the inputs.

Now create a *main.css* file and save it in the same folder as your *index.html* file.

Add the following CSS to the file:

```
1 .container {  
2   max-width: 500px;  
3   padding-top: 50px;  
4 }
```

Save. Refresh *index.html* in your browser.

What's going on?

Look back at the CSS file.

1. We have the *.container selector*, which is associated with the selector in our HTML document, followed by curly braces.
2. Inside the curly braces, we have *properties*, which are descriptive words, like font-weight, font-size, or background color.
3. *Values* are then assigned to each property, which are preceded by a colon and followed by a semi-colon. <http://cssvalues.com/> is an excellent resource for finding the acceptable values given a CSS property.

Putting it all together

First, add the following selectors to the HTML:

```
1 <!doctype html>  
2 <html>  
3   <head>  
4     <link rel="stylesheet" type="text/css"  
       href="http://netdna.bootstrapcdn.com/bootswatch/3.0.3/flatly/bootstrap.min.css">  
5     <link rel="stylesheet" type="text/css" href="main.css">  
6     <title>My Todo List</title>  
7   </head>  
8   <body>  
9
```

```

10 <div class="container">
11   <h1>My Todo List</h1>
12   <p class="lead">Get yourself organized!</p>
13   <br>
14   <form id="my-form" role="form">
15     <input id="my-input" class="form-control" type="text" placeholder="Enter
16     a todo...">
17     <br>
18   </form>
19   <button class="btn btn-primary btn-md">Submit!</button>
20   <br>
21 </div>
22 </body>
23 </html>

```

NOTE: Do you see the selectors? Look for the new ids and **classes**. The ids will all be used for Javascript, while the **classes** are all bootstrap styles. If you're curious, check out the bootstrap [website](#) to see more info about these styles.

Save. Refresh your browser.

What do you think? Good. Bad. Ugly? Make any additional changes that you'd like.

Chrome Developer Tools

Using Chrome Developer Tools, we can test temporary changes to either HTML or CSS directly from the browser. This can save a lot of time, plus you can make edits to any website, not just your own.

Open up the HTML page we worked on. Right Click on the heading. Select “Inspect Element”. Notice the styles on the right side of the Developer Tools pane associated with the heading. You can change them directly from that pane. Try adding the following style:

```
1 color: red;
```

This should change the color of the heading to red. Check out the live results in your browser.

You can also edit your HTML in real time. With Dev Tools open, right click the paragraph text in the left pane and select “Edit as HTML” Add another paragraph.

WARNING: Be careful as these changes are temporary. Watch what happens when you refresh the page. Poof!

Again, this is a great way to test temporary HTML and CSS changes live in your browser. You can also find bugs and/or learn how to imitate a desired HTML, CSS, Javascript effect from a different webpage.

Make sure both your .html and .css files are saved. In the second tutorial we’ll add user interactivity with Javascript and jQuery so that we can actually add and remove todo items.

Back to Flask ...

Homework

- If you want some extra practice, go through the Codecademy series on [HTML and CSS](#).
- Want even more practice, try [this](#) fun game for learning CSS selectors.

Chapter 11

Flask: FlaskTaskr (continued...)

Templates and Styling

Now that we're done with creating the basic app, let's update the styles. Thus far, we've been using the CSS styles from the main Flask tutorial. Let's add our own styles. We'll start with using [Bootstrap](#) to add a basic design template, then we'll edit the our CSS file to make style changes to the template.

Bootstrap is a front-end framework that makes your app look good right out of the box.¹ 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](#) the associated files and place them in your project directory:

```
1
2 static
3     css
4         bootstrap.min.css
5     js
6         bootstrap.min.js
7         styles.css
```

Or you can just link directly to the styles in your *template.html* file via a public content delivery network (CDN), which is a repository of commonly used files.

Either method is fine. Let's use the latter method. Add the following files to your *template.html*:

```
1 <!-- styles -->
2 <link
3     href="http://netdna.bootstrapcdn.com/bootstrap/3.1.0/css/bootstrap.min.css"
4     rel="stylesheet">
```

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

```

4 <!-- scripts -->
5 <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
6 <script
  src="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/js/bootstrap.min.js"></script>

```

Your template should now look like this:

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7     <meta name="description" content="">
8     <meta name="author" content="">
9     <title>Welcome to FlaskTaskr!!!</title>
10    <!-- styles -->
11    <link
12      href="http://netdna.bootstrapcdn.com/bootstrap/3.1.0/css/bootstrap.min.css"
13      rel="stylesheet">
14    <link rel="stylesheet" href="{ url_for('static', filename='css/styles.css')
15      }>">
16  </head>
17  <body>
18    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
19      <div class="container">
20        <div class="navbar-header">
21          <button type="button" class="navbar-toggle" data-toggle="collapse"
22            data-target=".navbar-collapse">
23            <span class="sr-only">Toggle navigation</span>
24            <span class="icon-bar"></span>
25            <span class="icon-bar"></span>
26            <span class="icon-bar"></span>
27          </button>
28          <a class="navbar-brand" href="/">FlaskTaskr</a>
29        </div>
30        <div class="collapse navbar-collapse">
31          <ul class="nav navbar-nav">
32            {% if not session.logged_in %}
33              <li><a href="/register">Signup</a></li>
34            {% else %}
35              <li><a href="/logout">Logout</a></li>
36            {% endif %}
37          </ul>
38        </div><!--/.nav-collapse -->
39      </div>
40    </div>

```



```

37 <div class="container">
38   <div class="content">
39     {% for message in get_flashed_messages() %}
40     <div class=flash>{{ message }}</div>
41     <br>
42     {% endfor %}
43     {% if error %}
44     <p class=error><strong>Error:</strong> {{ error }}
45     {% endif %}
46     {% block content %}
47     {% endblock %}
48   </div>
49   <div class="footer">
50     <hr>
51     <p>&copy; <a href="http://www.realpython.com">Real Python</a></p>
52   </div>
53 </div><!-- /.container -->
54 <!-- scripts -->
55 <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
56 <script
57   src="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/js/bootstrap.min.js"></script>
58 </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. Be sure to check out the bootstrap [documentation](#) for more information.

Take a look at your app. See the difference. Now, let's update the sub-templates:

1. *login.html*:

```

1  {% extends "template.html" %}
2  {% block content %}
3  <h1>Welcome to FlaskTaskr.</h1>
4  <form class="form-signin" role="form" method="post" action="">
5    <div class="lead">Please sign in to access your task list</div>
6    {{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&{{ form.password.label
7    }}: {{ form.password }}&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&
8    <button class="btn btn-lg btn-primary" type="submit">Sign in</button>
9    <br>
10   <p><em>Need an account? </em><a href="/register">Signup!!</a></p>
11 </form>
12 {% endblock %}

```

2. *register.html*:

```

1 {% extends "template.html" %}
2 {% block content %}
3 <h1>Welcome to FlaskTaskr.</h1>
4 <form class="form-signin" role="form" method="post" action="">
5   <div class="lead">Please register to start a task list</div>
6   <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{ form.email.label
7     }}: {{ form.email }}</p>
8   <p>{{ form.password.label }}: {{ form.password }}&nbsp;&nbsp;&nbsp;{{
9     form.confirm.label }}: {{ form.confirm }}</p>
10   <button class="btn btn-lg btn-primary" type="submit">Sign in</button>
11   <br>
12   <br>
13   <p><em>Already registered?</em> Click <a href="/">here</a> to login.</p>
</form>
{% endblock %}

```

3. tasks.html:

```

1 {% extends "template.html" %}
2 {% block content %}
3 <h1>Welcome to FlaskTaskr.</h1>
4 <div class="add-task">
5   <div class="lead">Add a new task:</div>
6   <form action="{ url_for('new_task') }" method="post" role="form">
7     <p>
8       {{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{ form.due_date.label
9         }}: {{ form.due_date }}&nbsp;&nbsp;&nbsp;{{ form.posted_date.label }}: {{
10         form.posted_date }}&nbsp;&nbsp;&nbsp;{{ form.priority.label }}: {{ form.priority
11         }}&nbsp;&nbsp;&nbsp;
12       <button class="btn btn-md btn-primary"
13         type="submit">Submit</button></p>
14     </form>
15 </div>
16 <div class="entries">
17   <br>
18   <br>
19   <div class="lead">Open tasks</div>
20   <br>
21   <table class="table table-bordered">
22     <thead>
23       <tr>
24         <th>Task Name</th>
25         <th>Due Date</th>
26         <th>Posted Date</th>
27         <th>Priority</th>
28         <th>Posted By</th>
29         <th>Actions</th>

```

```

26     </tr>
27 </thead>
28 <tbody>
29 {% for o in open_tasks %}
30     <tr>
31         <td>{{ o.name }}</td>
32         <td>{{ o.due_date }}</td>
33         <td>{{ o.posted_date }}</td>
34         <td>{{ o.priority }}</td>
35         <td>{{ o.poster.name }}</td>
36         <td>
37             <a href="{ { url_for('delete_entry', task_id = o.task_id)
38             }}">Delete</a> -
39             <a href="{ { url_for('complete', task_id = o.task_id) }}">Mark as
40             Complete</a>
41         </td>
42     </tr>
43 {% endfor %}
44 </tbody>
45 </table>
46 </div>
47 <br>
48 <br>
49 <div class="entries">
50     <div class="lead">Closed tasks</div>
51     <br>
52     <table class="table table-bordered">
53         <thead>
54             <tr>
55                 <th>Task Name</th>
56                 <th>Due Date</th>
57                 <th>Posted Date</th>
58                 <th>Priority</th>
59                 <th>Posted By</th>
60                 <th>Actions</th>
61             </tr>
62         </thead>
63         <tbody>
64             {% for c in closed_tasks %}
65                 <tr>
66                     <td>{{ c.name }}</td>
67                     <td>{{ c.due_date }}</td>
68                     <td>{{ c.posted_date }}</td>
69                     <td>{{ c.priority }}</td>
70                     <td>{{ c.poster.name }}</td>
71                     <td>
72                         <a href="{ { url_for('delete_entry', task_id = c.task_id)

```

```

71     }}">Delete</a>
72         </td>
73     </tr>
74     {% endfor %}
75 </tbody>
76 </table>
77 </div>
78 {% endblock %}

```

4. Update the styles in *styles.css*:

```

1 body {
2     padding: 60px 0px;
3     font-family: sans-serif;
4     background: #eee;
5 }
6
7 .flash {
8     background: #CEE5F5;
9     padding: 0.5em;
10 }
11
12 .error {
13     background: #F0D6D6;
14     padding: 0.5em;
15 }
16
17 .table {
18     background-color: white;
19 }
20
21 thead {
22     background-color: black;
23     color: white
24 }

```

Continue to make as many changes as you'd like. Make it unique.

See what you can do on your own.

Blueprints

Flask has a built-in feature called Blueprints that allows us to break our application into components.² This is nice feature especially for larger applications because it significantly increases code maintainability and re-usability through encapsulating code, templates and static media.

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.

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.

Let's add Blueprints to our FlaskTaskr application. We'll break it down to two components: - **Users**: this will handle user login, logout, and registration - **Tasks**: this will handle operations for viewing, deleting, and marking tasks as complete

Ready?

1. Let's start by identifying the existing directory structure and files for our application:

```
1  .
2  config.py
3  db_create.py
4  db_migrate.py
5  flasktaskr.db
6  forms.py
7  models.py
8  run.py
9  static
10     css
11         styles.css
12     img
13     js
14  templates
15     404.html
16     500.html
17     login.html
18     register.html
19     tasks.html
20     template.html
21  test.db
22  test.py
23  views.py
```

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

2. After breaking our app up using Blueprints, our application should have the following structure:

```
1  app
2
3      __init__.py
4      models.py
5      static
6          css
7              styles.css
8          img
9          js
10     tasks
11         __init__.py
12         forms.py
13         static
14         templates
15             tasks
16                 tasks.html
17         views.py
18     templates
19         404.html
20         500.html
21         template.html
22     users
23         __init__.py
24         forms.py
25         static
26         templates
27             users
28                 login.html
29                 register.html
30                 template.html
31         views.py
32     views.py
33 config.py
34 db_create.py
35 db_migrate.py
36 flasktaskr.db
37 run.py
38 test.db
39 test.py
```

Study this new structure. You'll notice that both the "Users" and "Tasks" components have their own views, forms, templates, and static files. On the other hand, the *models.py* file remains at the project root directory. In our case, the users and tasks table are bound by a database relationship so it makes sense to define them in a single file.

Let's walk through each step to convert our application.

Convert to Blueprints

1. Create the new folders, “tasks” and “users” with a new directory called “app”.
2. Create two empty `_init_.py` files within the new directories, which indicate to the Python interpreter that such directories should be treated as Packages.
3. Add “static” and “templates” directories within both the “tasks” and “users” directories.
4. Add another empty `_init_.py` file as well as the following existing folders to the new “app” directory:
 - “static”
 - “templates”
5. Move `models.py` and `views.py` to the “app” directory
6. Within your project root, you should still have `congif.py`, `db_create.py`, `db_migrate.py`, `flask-taskr.py`, `run.py`, `test.db`, and `test.py`.

Next, let's create our individual components, User and Tasks.

Users Component

1. Views

```
1  # /app/users/views.py
2
3
4  from app import db
5  from flask import Blueprint, flash, redirect, render_template, request,
    session, url_for
6  from app.views import login_required, flash_errors
7  from forms import RegisterForm, LoginForm
8  from app.models import User
9  from sqlalchemy.exc import IntegrityError
10
11  mod = Blueprint('users', __name__, url_prefix='/users',
12                template_folder='templates', static_folder='static')
13
14  @mod.route('/logout/')
15  def logout():
16      session.pop('logged_in', None)
17      session.pop('user_id', None)
18      flash('You are logged out. Bye. :(')
19      return redirect(url_for('.login'))
20
21  @mod.route('/', methods=['GET', 'POST'])
```

```

22 def login():
23     error = None
24     if request.method=='POST':
25         u = User.query.filter_by(name=request.form['name'],
26                                 password=request.form['password']).first()
27         if u is None:
28             error = 'Invalid username or password.'
29         else:
30             session['logged_in'] = True
31             session['user_id'] = u.id
32             flash('You are logged in. Go Crazy.')
33             return redirect(url_for('tasks.tasks'))
34
35     return render_template("users/login.html",
36                           form = LoginForm(request.form),
37                           error = error)
38
39 @mod.route('/register/', methods=['GET', 'POST'])
40 def register():
41     error = None
42     form = RegisterForm(request.form, csrf_enabled=False)
43     if form.validate_on_submit():
44         new_user = User(
45             form.name.data,
46             form.email.data,
47             form.password.data,
48         )
49         try:
50             db.session.add(new_user)
51             db.session.commit()
52             flash('Thanks for registering. Please login.')
53             return redirect(url_for('.login'))
54         except IntegrityError:
55             error = 'Oh no! That username and/or email already exist.
56             Please try again.'
57         else:
58             flash_errors(form)
59     return render_template('/users/register.html', form=form, error=error)

```

What's going on here?

1. First, we defined our Users Blueprint along with the custom templates and static folders:

```

1 mod = Blueprint('users', __name__, url_prefix='/users',
    template_folder='templates', static_folder='static')

```


2. We also assigned the Blueprint to be accessible at `url_prefix='/users'`, which means the Users component is accessible at the <http://127.0.0.1:5000/users> url.
3. We then use the `mod` variable in place of `app` within the decorators - i.e., `@mod.route('/logout/')`
4. For the `url_for`, we passed in new arguments in two instances above:
 - `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.
5. For the `render_template`, we specified 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

```

1 # /app/users/forms.py
2
3 from flask_wtf import Form
4 from wtforms import TextField, PasswordField
5 from wtforms.validators import DataRequired, EqualTo, Length
6
7 class RegisterForm(Form):
8     name = TextField('Username', validators=[DataRequired(), Length(min=6,
9     max=25)])
10    email = TextField('Email', validators=[DataRequired(), Length(min=6,
11    max=40)])
12    password = PasswordField('Password', validators=[DataRequired(),
13    Length(min=6, max=40)])
14    confirm = PasswordField('Repeat Password', [DataRequired(),
15    EqualTo('password', message='Passwords must match')])
16
17 class LoginForm(Form):
18     name = TextField('Username', validators=[DataRequired()])
19     password = PasswordField('Password', validators=[DataRequired()])

```

3. /app/users/templates/users/template.html:

```

1 <!-- add a template specific to the users component (if desired) -->

```

4. /app/users/templates/users/login.html:

```

1 {% extends "template.html" %}
2 {% block content %}
3 <h1>Welcome to FlaskTaskr.</h1>
4 <form class="form-signin" role="form" method="post" action="">
5   <div class="lead">Please sign in to access your task list</div>

```

```

6      {{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{ form.password.label
      }}: {{ form.password }}&nbsp;&nbsp;&nbsp;
7      <button class="btn btn-lg btn-primary" type="submit">Sign in</button>
8      <br>
9      <br>
10     <p><em>Need an account? </em><a href="{{ url_for('users.register')
      }}">Signup!!</a></p>
11 </form>
12 {% endblock %}

```

5. /app/users/templates/users/register.html:

```

1  {% extends "template.html" %}
2  {% block content %}
3  <h1>Welcome to FlaskTaskr.</h1>
4  <form class="form-signin" role="form" method="post" action="">
5      <div class="lead">Please register to start a task list</div>
6      <p>{{ form.name.label }}: {{ form.name }}&nbsp;&nbsp;&nbsp;{{ form.email.label
      }}: {{ form.email }}</p>
7      <p>{{ form.password.label }}: {{ form.password }}&nbsp;&nbsp;&nbsp;{{
      form.confirm.label }}: {{ form.confirm }}</p>
8      <button class="btn btn-lg btn-primary" type="submit">Sign in</button>
9      <br>
10     <br>
11     <p><em>Already registered?</em> Click <a href="{{ url_for('users.login')
      }}">here</a> to login.</p>
12 </form>
13 {% endblock %}

```

Tasks Component

1. Views

```

1  # /app/tasks/views.py
2
3
4  from app import db
5  from flask import Blueprint, flash, redirect, render_template, request,
      session, url_for
6  from app.views import login_required, flash_errors
7  from forms import AddTask
8  from app.models import FTasks
9
10 mod = Blueprint('tasks', __name__, url_prefix='/tasks',
11                 template_folder='templates', static_folder='static')
12

```

```

13 @mod.route('/tasks/')
14 @login_required
15 def tasks():
16     open_tasks = db.session.query(FTasks).filter_by(status='1').order_by(
17         FTasks.due_date.asc())
18     closed_tasks = db.session.query(FTasks).filter_by(status='0').order_by(
19         FTasks.due_date.asc())
20     return render_template('tasks/tasks.html', form = AddTask(request.form),
21                           open_tasks=open_tasks, closed_tasks=closed_tasks)
22
23 @mod.route('/add/', methods=['GET', 'POST'])
24 @login_required
25 def new_task():
26     form = AddTask(request.form, csrf_enabled=False)
27     if form.validate_on_submit():
28         new_task = FTasks(
29             form.name.data,
30             form.due_date.data,
31             form.priority.data,
32             form.posted_date.data,
33             '1',
34             session['user_id']
35         )
36         db.session.add(new_task)
37         db.session.commit()
38         flash('New entry was successfully posted. Thanks.')
39     else:
40         flash_errors(form)
41     return redirect(url_for('.tasks'))
42
43 @mod.route('/complete/<int:task_id>/',)
44 @login_required
45 def complete(task_id):
46     new_id = task_id
47     db.session.query(FTasks).filter_by(task_id=new_id).update({"status": "0"})
48     db.session.commit()
49     flash('The task was marked as complete. Nice.')
50     return redirect(url_for('.tasks'))
51
52 @mod.route('/delete/<int:task_id>/',)
53 @login_required
54 def delete_entry(task_id):
55     new_id = task_id
56     db.session.query(FTasks).filter_by(task_id=new_id).delete()
57     db.session.commit()
58     flash('The task was deleted. Why not add a new one?')
59     return redirect(url_for('.tasks'))

```

2. Forms

```
1 # /app/tasks/forms.py
2
3
4 from flask_wtf import Form
5 from wtforms import TextField, DateField, IntegerField, \
6     SelectField
7 from wtforms.validators import DataRequired
8
9
10 class AddTask(Form):
11     task_id = IntegerField('Priority')
12     name = TextField('Task Name', validators=[DataRequired()])
13     due_date = DateField('Date Due (mm/dd/yyyy)',
14                          validators=[DataRequired()],
15                          format='%m/%d/%Y')
16     priority = SelectField('Priority', validators=[DataRequired()],
17                           choices=[('1', '1'), ('2', '2'), ('3', '3'),
18                                     ('4', '4'), ('5', '5')])
19     status = IntegerField('Status')
20     posted_date = DateField('Posted Date (mm/dd/yyyy)',
21                             validators=[DataRequired()], format='%m/%d/%Y')
```

3. /app/tasks/templates/tasks.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7     <meta name="description" content="">
8     <meta name="author" content="">
9     <title>Welcome to FlaskTaskr!!!</title>
10    <!-- styles -->
11    <link
12      href="http://netdna.bootstrapcdn.com/bootstrap/3.1.0/css/bootstrap.min.css"
13      rel="stylesheet">
14    <link rel="stylesheet" href="{ url_for('static',
15      filename='css/styles.css') }}">
16  </head>
17  <body>
18    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
19      <div class="container">
20        <div class="navbar-header">
21          <button type="button" class="navbar-toggle"
22            data-toggle="collapse" data-target=".navbar-collapse">
```

```

19         <span class="sr-only">Toggle navigation</span>
20         <span class="icon-bar"></span>
21         <span class="icon-bar"></span>
22         <span class="icon-bar"></span>
23     </button>
24     <a class="navbar-brand" href="/">FlaskTaskr</a>
25 </div>
26 <div class="collapse navbar-collapse">
27     <ul class="nav navbar-nav">
28         {% if not session.logged_in %}
29         <li><a href="{{url_for('users.register')}}">Signup</a></li>
30         {% else %}
31         <li><a href="{{url_for('users.login')}}">Logout</a></li>
32         {% endif %}
33     </ul>
34 </div><!--/.nav-collapse -->
35 </div>
36 <div class="container">
37     <div class="content">
38         {% for message in get_flashed_messages() %}
39         <div class=flash>{{ message }}</div>
40         <br>
41         {% endfor %}
42         {% if error %}
43         <p class=error><strong>Error:</strong> {{ error }}
44         {% endif %}
45         {% block content %}
46         {% endblock %}
47     </div>
48     <div class="footer">
49         <hr>
50         <p>&copy; <a href="http://www.realpython.com">Real Python</a></p>
51     </div>
52 </div><!-- /.container -->
53 <!-- scripts -->
54 <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>
55 <script
56     src="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/js/bootstrap.min.js"></script>
57 </body>
58 </html>

```

Main App

1. /app/init.py:

```

1 # __init__.py
2
3
4 from flask import Flask
5 from flask.ext.sqlalchemy import SQLAlchemy
6
7 app = Flask(__name__)
8 app.config.from_object('config')
9 db = SQLAlchemy(app)
10
11 if not app.debug:
12     import os
13     import logging
14
15     from logging import Formatter, FileHandler
16     from config import basedir
17
18     file_handler = FileHandler(os.path.join(basedir, 'error.log'))
19     file_handler.setFormatter(Formatter('%(asctime)s %(levelname)s:
20     %(message)s '[in %(pathname)s:%(lineno)d]'))
21     app.logger.setLevel(logging.INFO)
22     file_handler.setLevel(logging.INFO)
23     app.logger.addHandler(file_handler)
24     app.logger.info('errors')
25
26 from app import views, models
27 from app.users.views import mod as usersMod
28 from app.tasks.views import mod as tasksMod
29
30 app.register_blueprint(usersMod)
31 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:

```

1 # views.py
2
3
4 from app import app, db
5 from flask import flash, redirect, render_template, session, url_for
6 from functools import wraps
7

```

```

8 def flash_errors(form):
9     for field, errors in form.errors.items():
10         for error in errors:
11             flash(u"Error in the %s field - %s" % (
12                 getattr(form, field).label.text,error), 'error')
13
14 def login_required(test):
15     @wraps(test)
16     def wrap(*args, **kwargs):
17         if 'logged_in' in session:
18             return test(*args, **kwargs)
19         else:
20             flash('You need to login first.')
21             return redirect(url_for('users.login'))
22     return wrap
23
24 @app.errorhandler(500)
25 def internal_error(error):
26     db.session.rollback()
27     return render_template('500.html'), 500
28
29 @app.errorhandler(404)
30 def internal_error(error):
31     return render_template('404.html'), 404
32
33 @app.route('/', defaults={'page': 'index'})
34 def index(page):
35     return(redirect(url_for('users.login')))

```

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:

```

1 @app.route('/', defaults={'page': 'index'})
2 def index(page):
3     return(redirect(url_for('tasks.tasks')))

```

This view redirects the homepage to the tasks view of the Tasks blueprint.

3. /app/templates/404.html:

```

1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Sorry ...</h1>
4     <p>There's nothing here!</p>
5     <p><a href="{url_for('users.login')}">Back</a></p>
6 {% endblock %}

```

4. /app/templates/500.html:

```
1 {% extends "template.html" %}
2 {% block content %}
3     <h1>Something's wrong!</h1>
4     <p>Fortunately we are on the job, and you can just return to the login
      page!</p>
5     <p><a href="{{url_for('login')}}">Back</a></p>
6 {% 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 function 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.

Test Coverage

The best way to reduce bugs and ensure working code is to have a comprehensive test suite in place. Coverage testing is a great way to achieve this as it analyzes your code base and returns a report showing the parts not covered by a test. Keep in mind that even if 100% of your code is covered by tests, there still could be [issues](#) due to flaws in how you structured your tests.

To implement coverage testing, we'll use a tool called [coverage](#).

Install:

```
$ pip install coverage==3.7.1
```

To run coverage, use the following command:

```
$ coverage run tests.py
```

To get a command line coverage report:

```
$ coverage report -m
```

To print a fancier HTML report:

```
$ coverage html
```

Check the report in the newly created “htmlcov” directory by opening *index.html* in your browser.

You should see:

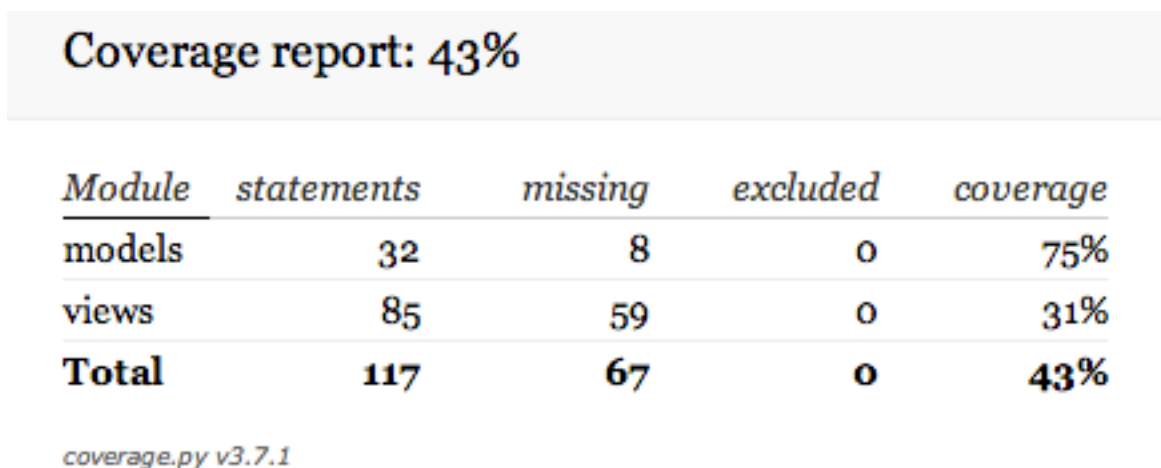


Figure 11.1: coverage report

If you click on one of the modules, you'll see the actual results, line by line. Lines highlighted in red are currently not covered by a test.

More Tests

If you remember, our app currently has this functionality:

1. Users sign in and out from the landing page.
2. New users can register on a registration page.
3. Once signed in, users can add new tasks. Each task consists of a name, due date, priority, status, and an auto-incremented ID.
4. Users can view all uncompleted tasks from the same screen.
5. Users can also delete tasks and mark tasks as completed. If a user deletes a task, it will also be deleted from the database.

Split the test file into two files, `test_tasks.py` and `test_users.py`. Now write all the tests you can to cover all the user functionality. Look to my test files for help. Run the reports again - `coverage run test_tasks.py` and `coverage run test_users.py`, respectively. What's your coverage percentage at?

Keep in mind that even if you do get your coverage percentage up to 100%, you still may not be testing everything because of the inherent [flaws](#) in testing in general.

Try to approach testing and the coverage reports as a guide to see what isn't being tested, and they also help you gain understanding of your code base in general.

Homework

- Although not specifically about Flask, watch [this](#) excellent video on testing Python code.

Logging

Let's face it, your code will never be 100% correct, and you will never be able to write tests to cover *everything* that could happen. Thus, it's vital that you set up a means of capturing all errors so that you can spot trends, setup additional error handlers, and, of course, fix errors that occur.

It's very easy to setup logging at the server level or within the Flask application itself.

There are also numerous third party libraries that can also manage the logging of errors on your behalf. But let's create our own.

Here's a basic logger that uses the logging library:

```
1 if not app.debug:
2     import os
3     import logging
4
5     from logging import Formatter, FileHandler
6     from config import basedir
7
8     file_handler = FileHandler(os.path.join(basedir, 'error.log'))
9     file_handler.setFormatter(Formatter('%(asctime)s %(levelname)s: %(message)s '
10 '[in %(pathname)s:%(lineno)d]'))
11     app.logger.setLevel(logging.INFO)
12     file_handler.setLevel(logging.INFO)
13     app.logger.addHandler(file_handler)
14     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 being the least severe, 5 being the most):

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 (like `ERROR` or `CRITICAL`) so only the most important errors are sent to your email - otherwise, you may just stop viewing them altogether.

In `__init__.py`, add the above logging code just before the code `from app 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. Take a look at the error message. Make sense?

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.³

Deploying an app to Heroku is ridiculously easy:

1. Signup for [Heroku](#)
2. Login and download the [Heroku Toolbelt](#) applicable to your operating system
3. Once installed, open your command-line and run the following command:

```
1 $ heroku login
```

Then follow the prompts:

```
1 Enter your Heroku credentials.  
2 Email: michael@realpython.com  
3 Password (typing will be hidden):  
4 Could not find an existing public key.  
5 Would you like to generate one? [Yn]  
6 Generating new SSH public key.  
7 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`. Keep in mind that 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:

```
1 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 Heroku).

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`:

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

```

1 # run.py
2
3
4 import os
5
6 from app import app
7
8 port = int(os.environ.get('PORT', 5000))
9 app.run(host='0.0.0.0', port=port, debug=False)

```

8. Anytime you PUSH to Heroku, you have to create a separate local Git repository within that project's root directory. That said, it can be hard working with a directory that contains a repository within another directory that contains a repository. I recommend moving the “Flask-Taskr” directory out of “realpython”.

Now create a new repo:

```

1 $ git init
2 $ git add .
3 $ git commit -am "initial"

```

9. Create your app on Heroku:

```

1 $ heroku create

```

10. Deploy your code to Heroku:

```

1 $ git push heroku master
2 $ heroku ps:scale web=1

```

11. Check to make sure your app is running:

```

1 $ heroku ps

```

12. View the app in your browser:

```

1 $ heroku open

```

13. If you see errors, open the Heroku log to view all errors and output:

```

1 $ heroku logs

```

That's it. Make sure that you also PUSH your local repository to Github.

You can see my app at <http://flasktaskr.herokuapp.com>. Cheers!

Fabric

In the last lesson we manually uploaded our code to Heroku. When you only need to deploy code to Heroku and GitHub, you can get away with doing this manually. However, if you are working with multiple servers where a number of developers are sending code each day, you will want to automate this process. We can use [Fabric](#) for such automation.

Setup

As always, start by installing with Pip:

```
1 pip install fabric
```

Fabric is controlled by a file called a fabfile, *fabfile.py*. You define all of the actions (or commands) that Fabric takes in this file. Create the file, placing it in your root directory.

The file itself takes a number of commands. Here's a brief list of some of the most common: - run runs a command on a remote server - local runs a local command - put uploads a local file to the remote server - cd changes the directory on the serverside - get downloads a file from the remote server - prompt prompts a user with text and returns the user input

Preparing

Add the following code to *fabfile.py*:

```
1 from fabric.api import local
2
3 # prepare for deployment
4
5 def test():
6     local("python test_tasks.py -v && python test_users.py -v")
7
8 def commit():
9     message = raw_input("Enter a git commit message: ")
10    local("git add . && git commit -am '{}'.format(message)")
11
12 def push():
13    local("git push origin master")
14
15 def prepare():
16    test()
17    commit()
18    push()
```

Essentially, we import the `local` function from Fabric, then run the basic shell commands for testing and PUSHing to GitHub as you've seen before.

Test this out:

```
1 $ fab prepare
```

If all goes well, this should run the tests, commit the code to your local repo, and then deploy to GitHub.

Testing

What happens if your tests fail? Wouldn't you want to abort the process? Probably.

Update your `test()` function to:

```
1 def test():
2     with settings(warn_only=True):
3         result = local("python test_tasks.py -v && python test_users.py -v",
4             capture=True)
5         if result.failed and not confirm("Tests failed. Continue?"):
6             abort("Aborted at user request.")
```

Also update the imports:

```
1 from fabric.api import local, settings, abort
2 from fabric.contrib.console import confirm
```

Here, if a test fails, then the user is asked to confirm whether or not the script should continue running.

Deploying

Finally, let's deploy:

```
1 def pull():
2     local("git pull origin master")
3
4 def heroku():
5     local("git push heroku master")
6
7 def heroku_test():
8     local("heroku run python test_tasks.py -v && heroku run python test_users.py -v")
9
10 def deploy():
11     pull()
12     test()
13     commit()
14     heroku()
```

```
15 heroku_test()
```

Now if you run the `deploy()` function, you PULL the latest code from GitHub, test the code, commit it to your local repo, PUSH to Heroku, and then test on Heroku. This should all look familiar.

NOTE: This is a relatively basic fabfile. If are working with a team of developers and PUSH code to multiple servers, then your fabfile will be much larger and more complex.

The last thing we need to add is a quick rollback.

```
1 def rollback():
2     local("heroku rollback")
```

If you do run into an error on Heroku, you want to immediately load a prior commit to get it working. You can do this quickly now by running the command:

```
1 $ fab rollback
```

Keep in mind that this is just a temporary fix. After you rollback, make sure to fix the issue locally and then PUSH the updated code to Heroku.

Building a REST API

Finally, let's look at how to design a RESTful API in Flask using FlaskTaskr. We'll look more at APIs in an upcoming chapter. For now we'll just define the basics, then build the actual API.

What is an API?

Put simply, an API is collection of functions that other programs can access or manipulate data from. Each function has an associated endpoint (also called a resource). One can make changes to a resource via the HTTP methods/verbs:

1. GET - view a resource
2. POST - create a new resource
3. PUT - update a resource
4. DELETE - delete a resource

Basic REST Design Practices

URLs (endpoints) are used for identifying a specific resource, while the HTTP verbs define the actions one can take on those resources. Each resource should only have two URLs. The first is for a collection, while the second is for a specific element in that collection.

For example, in FlaskTaskr, the endpoint `/tasks/` could be for the collection, while `/tasks/<id>/` could be for a specific task from the collection.

	GET	POST	PUT	DELETE
<code>/tasks/</code>	View all tasks	Add new task	Update all tasks	Delete all tasks
<code>/tasks/<id>/</code>	View specific task	N/A	Update specific task	Delete specific task

In our API, we are only going to be working with the GET request since it is read only.

Workflow for creating an API via Flask

1. Source data
2. Set up persistence layer
3. Add Flask
4. Set up/Install Flask-SQLAlchemy
5. Create URLs
6. Create Query Strings (optional)
7. Test, Test, Test

We already have the first four steps down, so let's start with creating our actual endpoints. Also, as of right now, we're not going to have the ability to add specific operations with query strings. Plus, we could further test our code with a unit test to assert that the expected returned JSON is the actual returned JSON; however, this is covered indirectly under our regular unit tests. If anything, you'd really just want to assert that hitting those two endpoints returns a 200 response - e.g., `self.assertEqual(response.status_code, 200)`.

With that, let's set up our endpoints.

Set up Endpoints

From the design above, the two URLs we want our app to support are `/tasks/` and `/tasks/<id>`. Let's start with the former.

Add the following code to the main *views.py*:

```
1 @app.route('/api/tasks/', methods=['GET'])
2 def tasks():
3     if request.method == 'GET':
4         results = db.session.query(FTasks).limit(10).offset(0).all()
5         json_results = []
6         for result in results:
7             data = {
8                 'task_id': result.task_id,
9                 'task name': result.name,
10                'due date': str(result.due_date),
11                'priority': result.priority,
12                'posted date': str(result.posted_date),
13                'status': result.status,
14                'user id': result.user_id
15            }
16            json_results.append(data)
17
18     return jsonify(items=json_results)
```

And make sure to update the imports to include `request`, `jsonify`, and `FTasks`:

```
1 from app import app, db
2 from flask import flash, redirect, render_template, session, url_for, request,
   jsonify
3 from functools import wraps
4 from app.models import FTasks
```

What's going on?

1. We map the URL `/api/tasks/` to the `tasks()` function so once that URL is requested via a GET request, we use query the database via SQLAlchemy to grab the first 10 records from the `ftasks` table.
2. Next we create a dictionary out of each return record from the database.
3. Finally, since our API supports JSON, we are passing the dictionary to the `jsonify()` function to render a JSON response back to the browser.

Navigate to <http://localhost:5000/api/tasks/> to view the returned JSON after you've fire up the server.

Now, let's add the next endpoint for `/tasks/<id>`:

```
1 @app.route('/api/task/<int:task_id>')
2 def task(task_id):
3     if request.method == 'GET':
4         result = db.session.query(FTasks).filter_by(task_id=task_id).first()
5
6         json_result = {
7             'task_id' : result.task_id,
8             'task name': result.name,
9             'due date': str(result.due_date),
10            'priority': result.priority,
11            'posted date': str(result.posted_date),
12            'status': result.status,
13            'user id': result.user_id
14        }
15
16     return jsonify(items=json_result)
```

What's going on?

This is very similar to our last endpoint. The difference is that we are using a dynamic route to grab a query and render by a specific `task_id`.

Test this out. Navigate to <http://localhost:5000/api/task/2>.

Boilerplate Template and Workflow

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 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

1. Create a new working directory. Keep this outside of your “realpython” directory. In fact, a directory will be created for you in a minute, so you may just want to start working from your “desktop” or “documents” directory.
2. Download (or clone) the boilerplate template from Github:

```
1 $ git clone https://github.com/mjhea0/flask-boilerplate.git
2 $ cd flask-boilerplate
```

This creates a new folder called “flask-boilerplate”.

Project structure:

```
1
2 Procfile
3 Procfile.dev
4 README.md
5 app.py
6 config.py
7 error.log
8 forms.py
9 models.py
10 requirements.txt
11 static
12     css
13         bootstrap-3.0.0.min.css
14         bootstrap-theme-3.0.0.css
15         bootstrap-theme-3.0.0.min.css
16         font-awesome-3.2.1.min.css
17         layout.forms.css
18         layout.main.css
19         main.css
20         main.quickfix.css
21         main.responsive.css
22     font
```

```

23     FontAwesome.otf
24     fontawesome-webfont.eot
25     fontawesome-webfont.svg
26     fontawesome-webfont.ttf
27     fontawesome-webfont.woff
28     ico
29         apple-touch-icon-114-precomposed.png
30         apple-touch-icon-144-precomposed.png
31         apple-touch-icon-57-precomposed.png
32         apple-touch-icon-72-precomposed.png
33         favicon.png
34     img
35     js
36         libs
37             bootstrap-3.0.0.min.js
38             jquery-1.10.2.min.js
39             modernizr-2.6.2.min.js
40             respond-1.3.0.min.js
41         plugins.js
42         script.js
43     templates
44         errors
45             404.html
46             500.html
47         forms
48             forgot.html
49             login.html
50             register.html
51         layouts
52             form.html
53             main.html
54         pages
55             placeholder.about.html
56             placeholder.home.html

```

3. Install the various libraries and dependencies:

```

1 $ pip install -r requirements.txt

```

You can also view the dependencies by running the command `pip freeze`:

```

Fabric==1.8.2
Flask==0.10.1   Flask-SQLAlchemy==1.0   Flask-WTF==0.9.4   Jinja2==2.7.2
MarkupSafe==0.18   SQLAlchemy==0.9.3   WTForms==1.0.5   Werkzeug==0.9.4
coverage==3.7.1   ecdsa==0.10   itsdangerous==0.23   paramiko==1.12.2
pycrypto==2.6.1   wsgiref==0.1.2

```

Deploy to Heroku

Before you do this delete the hidden “.git” directory. Follow the steps in the last lesson. Or view the Github readme [here](#)

Development workflow

Now that you have your skeleton app up, it's time to start developing locally.

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

NOTE: Remember: the fabfile performs this exact workflow (steps 1 through 5) for you automatically

That's it. You now have a skeleton app to work with to build your own applications. Cheers!

NOTE: 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 done with Flask. Let's move on to a high-level web framework: **web2py**.

Chapter 12

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.

NOTE Take note of the concept “Don’t Repeat Yourself” (or DRY). Always avoid reinventing the wheel. This is exactly what web frameworks excel at. The majority of the low-level tasks (listed above) are common to 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 automating testing as well.

The majority of web frameworks can be classified as either a full (high-level), or micro (low-level), 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 s/he wants. Beginning developers should first

focus on demystifying much of the *magic*, (commonly referred to as automation), to help understand the differences between the various web frameworks and avoid later confusion.

Keep in mind that while there are plenty of upsides to using web frameworks, most notably rapid development, there is also a huge downside: lock-in. Put simply, by choosing a specific framework, you lock your application into that framework's philosophy and become dependent on the framework's developers to maintain and update the code base.

In general, problems are more likely to arise with high-level frameworks due to the automation and features they provide; you have to do things *their* way. However, with low-level frameworks, you have to write more code up front to make up for the missing features, which slows the development process in the beginning. There's no right answer, but you do not want to have to change a mature application's framework; this is a daunting task to say the least. Choose wisely.

Popular Frameworks

1. **web2py**, **Django**, and **Turbogears** are all full frameworks, which offer a number of pre-installed utilities and automate many tasks beneath the hood. 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 are 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/libraries 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!

Chapter 13

web2py: QuickStart

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.



Figure 13.1: web2py

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 University¹), 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

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

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.

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

We'll be developing from the command line, so follow the full installation guide below.

Full Install

1. Create a directory called "web2py". Download the source code from the web2py [website](#) and place it into this new directory. Unzip the file. Rename the new directory from the zip file to "start". Navigate into that directory. *Both apps within this chapter will be developed within this same instance of web2py.*

2. Install and activate your virtualenv.

3. Back on your command line launch web2py:

```
1 $ python web2py.py
```

4. 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:

```
1 $ deactivate
```

5. To run web2py again, activate your virtualenv and launch web2py.

NOTE 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. We'll be addressing such libraries as we go along.

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

Hello World

1. Activate your virtualenv. Fire up the server.

```
1 $ source bin/activate
2 $ 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.

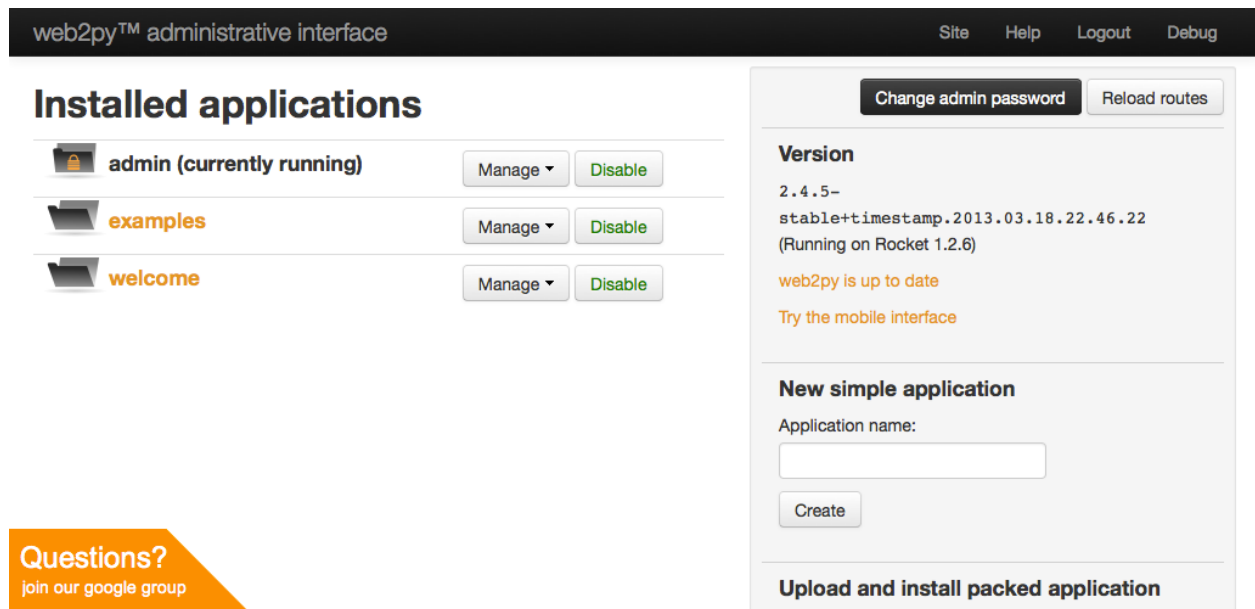


Figure 13.2: web2py sites

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:

NOTE You’ll soon find out that web2py has a default for pretty much everything (but it 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 as much (or at all) on defaults. Thus, I urge you to practice. Break things. Learn the default behaviors. Make them better. Make them your own.

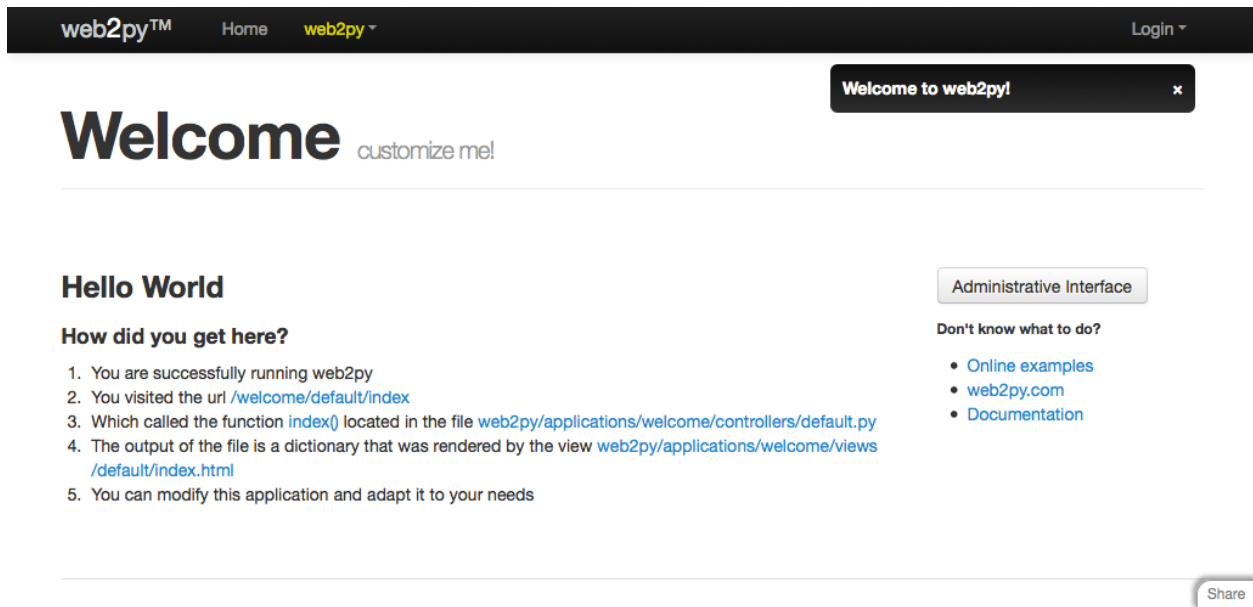


Figure 13.3: web2py welcome

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**. Replace the `index()` function with the following code:

```
1 def index():
2     return dict(message="Hello from web2py!")
```

7. Save the file, then hit the Back button to return to the **Edit** page.
8. Now let's update the view. Edit the *default/index.html* file, replacing the existing code with:

```
1 <html>
2   <head>
3     <title>Hello App!</title>
4   </head>
```



```
5 <body>
6   <br/>
7   <h1>{{=message}}</h1>
8 </body>
9 </html>
```

9. Save the file, then 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 output within the view when 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:

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`

Try this out. When done, hit CTRL-C from your terminal to stop the server.

Views ?

download layouts











Edit	 <code>__init__.py</code>
Edit	 <code>appadmin.html</code> extends layout.html
Edit	 <code>default/index.html</code>
Edit	 <code>default/user.html</code> extends layout.html
Edit	 <code>generic.html</code> extends layout.html
Edit	 <code>generic.ics</code> 
Edit	 <code>generic.json</code>
Edit	 <code>generic.jsonp</code>
Edit	 <code>generic.load</code>

Figure 13.4: web2py generic view

Deploying on PythonAnywhere

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



Figure 13.5: pythonanywhere

Simply invite a friend or colleague to join your session, and you have a 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”, then “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 web2py 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. shared hosting?
- Learn more about other Python PaaS options: <http://www.slideshare.net/appsembler/pycon-talk-deploy-python-apps-in-5-min-with-a-paas>

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

seconds2minutes App

Let's create a new app that converts, well, seconds to minutes. Activate your virtualenv, start the server, set a password, enter the Admin Interface, and create a new app called "seconds2minutes".

NOTE: 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 will then be displayed for users to enter the number of seconds they want converted over to minutes. Meanwhile, the second function, `convert()`, will take the number of seconds, converting 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:

```
1 def index():
2     form=FORM('# of seconds: ',
3         INPUT(_name='seconds', requires=IS_NOT_EMPTY()),
4         INPUT(_type='submit')).process()
5     if form.accepted:
6         redirect(URL('convert',args=form.vars.seconds))
7     return dict(form=form)
8
9 def convert():
10     seconds = request.args(0,cast=int)
11     return dict(seconds=seconds, minutes=seconds/60, new_seconds=seconds%60)
```

2. Edit the `default/index.html` view, replacing the default code with:

```
1 <center>
2 <h1>seconds2minutes</h1>
3 <h3>Please enter the number of seconds you would like converted to
   minutes.</h3>
4 <p>{{=form}}</p>
5 </center>
```

3. Create a new view called `default/convert.html`, replacing the default code with:

```
1 <center>
2 <h1>seconds2minutes</h1>
3 <p>{{=seconds}} seconds is {{=minutes}} minutes and {{=new_seconds}}
   seconds.</p>
4 <br/>
5 <p><a href="/seconds2minutes/default/index">Try again?</a><p>
6 </center>
```

4. 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.

5. Change the form validator to `requires=IS_INT_IN_RANGE(0,1000000)`. The new `index()` function looks like this:

```
1 def index():
2     form=FORM('# of seconds: ',
3               INPUT(_type='integer', _name='seconds',
4                     requires=IS_INT_IN_RANGE(0,1000000)),
5               INPUT(_type='submit')).process()
6     if form.accepted:
7         redirect(URL('convert',args=form.vars.seconds))
8     return dict(form=form)
```

Test it out. You should get an error, unless you enter an integer between 0 and 999,999:

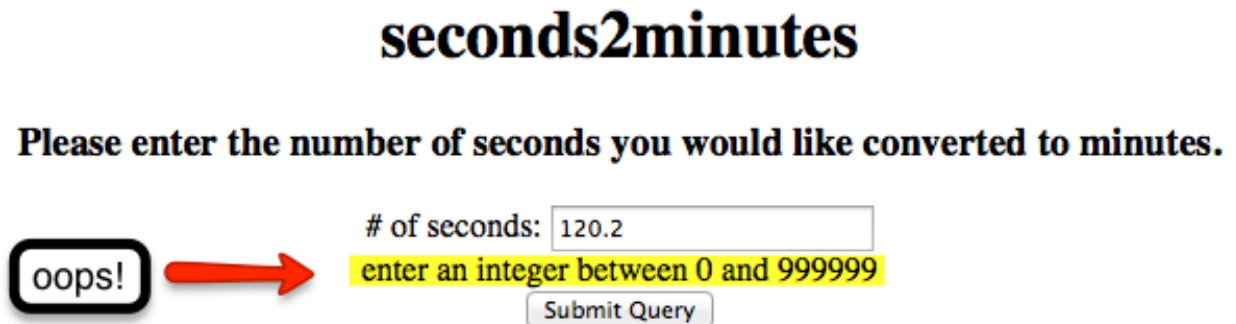


Figure 13.6: web2py error

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.

Chapter 14

Interlude: APIs

Introduction

This chapter focuses on client-side programming. 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
. *Or simply review the chapter in this course on HTML and CSS.*

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

For example:

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

Finally, to fully explore this topic, client-side programming, 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, some of which we will be covering in this Chapter.

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, create a new folder within the "realpython" directory called "client-side". Don't forget to create and activate a new virtualenv.

Install requests:

```
1 $ pip install requests
```

Code:

```
1 # Retrieving a web page
2
3
4 import requests
5
6 # retrieve the web page
7 r = requests.get("http://www.python.org/")
8
9 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:

```
1 # Downloading a web page
2
3
4 import requests
5
6 r = requests.get("http://www.python.org/")
7
8 # write the content to test_request.html
9 with open("test_requests.html", "wb") as code:
10     code.write(r.content)
```

Save the file as *clientb.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 Sublime. Now it's much easier to examine the actual HTML.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

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.

NOTE: “wb” stands for write binary, which downloads the raw bytes of the file. In other words, 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:

```
1 # Submitting to a web form
2
3
4 import requests
5
6 url = 'http://httpbin.org/post'
7 data = {'fname': 'Michael', 'lname': 'Herman'}
8
9 # submit post request
10 r = requests.post(url, data=data)
11
12 # display the response to screen
13 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 `http://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. However, the codes in **bold** above are the most common.

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:

```
1 print r.content
```

Save this as a new file called *clientd.py*. Run the file.

You should see the data you sent within the response:

```
1 "form": {  
2     "lname": "Herman",  
3     "fname": "Michael"  
4 },
```

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:

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, of course), 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 Chrome Developer Tools:

1. Open Chrome.
2. Right click anywhere on the screen and within the window, click “Inspect Element”.

3. You are now looking at the main Developer Tools pane:

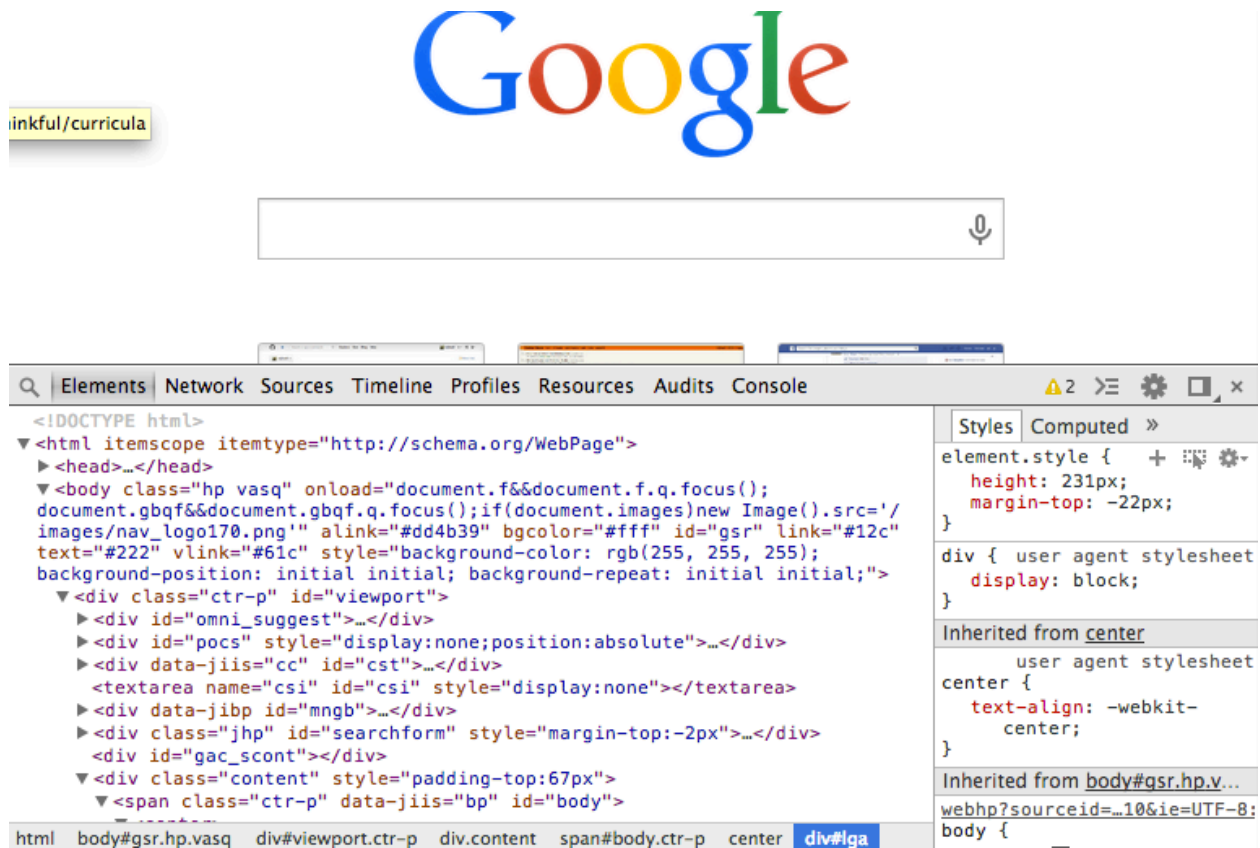


Figure 14.1: chrome dev tools

5. Click the Network panel.
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? You basically sent a request asking the server to display information for you.
7. Now login. If you're already logged in, go ahead and log out. Enter your login credentials. Pay close attention to the console.
8. Did you see the POST request? 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 depend 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. **Also, it's not really important to understand the abstract principles of RESTful design. Simply being able to recognize at a highlevel what it is and the associated four HTTP methods is sufficient.**

Summary

In summary, APIs:

- Facilitate the exchange of information,
- Speak a common language, and
- Can be access 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.

Before moving on, let's look at a fun example. <http://placekitten.com> is a REST API that returns a picture of a kitten given a width an height - <http://placekitten.com/>. You can test it out right in your browser; just navigate to these URLs:

- <http://placekitten.com/200/300>
- <http://placekitten.com/300/450>
- <http://placekitten.com/700/600>

Pretty cool, right?

Homework

- Read [this](#) article providing a high-level overview of standards associated with RESTful APIs.
- if you're still struggling with understanding what REST is and why we use it, [here's](#) a great video. After about 8:30 minutes it starts to get pretty technical, but before that it's pretty accessible to everyone.

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.

Please note: Although, JSON continues to push XML out of the picture in terms of REST, XML is still widely used and can be easier to parse.

With that, let's look at an example of an XML file:

```
1 <?xml version="1.0"?>
2 <CARS>
3   <CAR>
4     <MAKE>Ford</MAKE>
5     <MODEL>Focus</MODEL>
6     <COST>15000</COST>
7   </CAR>
8   <CAR>
9     <MAKE>Honda</MAKE>
10    <MODEL>Civic</MODEL>
11    <COST>20000</COST>
12  </CAR>
13  <CAR>
14    <MAKE>Toyota</MAKE>
15    <MODEL>Camry</MODEL>
16    <COST>25000</COST>
17  </CAR>
18  <CAR>
19    <MAKE>Honda</MAKE>
20    <MODEL>Accord</MODEL>
21    <COST>22000</COST>
22  </CAR>
23 </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.

Code:

```

1 # XML Parsing 1
2
3
4 from xml.etree import ElementTree as et
5
6 # parses the file
7 doc = et.parse("cars.xml")
8
9 # outputs the first MODEL in the file
10 print doc.find("CAR/MODEL").text

```

Output:

```

1 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:

```

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

```

The above line is equivalent to:

```

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

```

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

```

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

```

The output should be:

```

1 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:

```

1 # XML Parsing 2
2
3
4 from xml.etree import ElementTree as et
5

```

```

6 doc = et.parse("cars.xml")
7
8 # outputs the make, model and cost of each car to the screen
9 for element in doc.findall("CAR"):
10     print (element.find("MAKE").text + " " +
11           element.find("MODEL").text +
12           ", $" + element.find("COST").text)

```

You should get the following results:

```

1 Ford Focus, $15000
2 Honda Civic, $20000
3 Toyota Camry, $25000
4 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:

```

1 # XML Parsing 3
2
3
4 from xml.etree import ElementTree as et
5 import requests
6
7 # retrieve an xml document from a web server
8 xml = requests.get("http://www.w3schools.com/xml/cd_catalog.xml")
9
10 with open("test.xml", "wb") as code:
11     code.write(xml.content)
12
13 doc = et.parse("test.xml")
14
15 # outputs the album, artist and year of each CD to the screen
16 for element in doc.findall("CD"):
17     print "Album: ", element.find("TITLE").text
18     print "Artist: ", element.find("ARTIST").text
19     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.

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 Python dictionary.

Let's look at a quick example:

```
1 {  
2   "CARS": [  
3     {  
4       "MAKE": "Ford",  
5       "MODEL": "Focus",  
6       "COST": "15000"  
7     },  
8     {  
9       "MAKE": "Honda",  
10      "MODEL": "Civic",  
11      "COST": "20000"  
12    },  
13    {  
14      "MAKE": "Toyota",  
15      "MODEL": "Camry",  
16      "COST": "25000"  
17    },  
18    {  
19      "MAKE": "Honda",  
20      "MODEL": "Accord",  
21      "COST": "22000"  
22    }  
23  ]  
24 }
```

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.

Code:

```
1 # JSON Parsing 1
2
3
4 import json
5
6 # decodes the json file
7 output = json.load(open('cars.json'))
8
9 # display output to screen
10 print output
```

Output:

```
1 [{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're having a hard time, try changing the print statement to:

```
1 print json.dumps(output, indent=4, sort_keys=True)
```

Your output should now look like this:

```
1 [
2     {
3         "CAR": [
4             {
5                 "COST": "15000",
6                 "MAKE": "Ford",
7                 "MODEL": "Focus"
8             },
9             {
10                "COST": "20000",
11                "MAKE": "Honda",
12                "MODEL": "Civic"
13            },
14            {
15                "COST": "25000",
16                "MAKE": "Toyota",
17                "MODEL": "Camry"
18            },
19        ]
20     }
```

```

19         {
20             "COST": "22000",
21             "MAKE": "Honda",
22             "MODEL": "Accord"
23         }
24     ]
25 }
26 ]

```

Much easier to read, right?

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:

```

1 # JSON Parsing 2
2
3 import json
4
5 # decodes the json file
6 output = json.load(open('cars.json'))
7
8 # display output to screen
9 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: `Focus`.

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

Code:

```

1 # POST JSON Payload
2
3
4 import json
5 import requests
6
7 url = "http://httpbin.org/post"
8 payload = {"colors": [
9     {"color": "red", "hex": "#f00"},
10    {"color": "green", "hex": "#0f0"},
11    {"color": "blue", "hex": "#00f"},

```

```

12     {"color": "cyan", "hex": "#0ff"},
13     {"color": "magenta", "hex": "#f0f"},
14     {"color": "yellow", "hex": "#ff0"},
15     {"color": "black", "hex": "#000"}
16     ]]
17 headers = {"content-type": "application/json"}
18
19 # post data to a web server
20 response = requests.post(url, data=json.dumps(payload), headers=headers)
21
22 # output response to screen
23 print response.status_code

```

Output:

```

1 200 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 have 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.

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 client library (remember what a client library is/does?) for this next example: `pip install gdata`²

Code:

```
1 # Basic web services example
2
3
4 # import the client libraries
5 import gdata.youtube
6 import gdata.youtube.service
7
8 # YouTubeService() used to generate the object so that we can communicate with
   the YouTube API
9 youtube_service = gdata.youtube.service.YouTubeService()
10
11 # prompt the user to enter the Youtube User ID
12 playlist = raw_input("Please enter the user ID: ")
13
14 # setup the actual API call
15 url = "http://gdata.youtube.com/feeds/api/users/"
16 playlist_url = url + playlist + "/playlists"
17
18 # retrieve Youtube playlist
19 video_feed = youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
20
21 print "\nPlaylists for " + str.format(playlist) + ":\n"
22
23 # display each playlist to screen
24 for p in video_feed.entry:
25     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-

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

1. We started by importing the required libraries, which are for the Youtube Python client library;
2. We then established communication with Youtube; prompted the user for a user ID, and then made the API call to request the data.
3. 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:

```
1 # Basic web services example
2
3
4 # import the client libraries
5 import gdata.youtube
6 import gdata.youtube.service
7
8 # YouTubeService() used to generate the object so that we can communicate with
   the YouTube API
9 youtube_service = gdata.youtube.service.YouTubeService()
10
11 # prompt the user to enter the Youtube User ID
12 playlist = raw_input("Please enter the user ID: ")
13
14 # setup the actual API call
15 url = "http://gdata.youtube.com/feeds/api/users/"
16 playlist_url = url + playlist + "/playlists"
17
18 # retrieve Youtube playlist
19 video_feed = youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
20
21 print video_feed
22
23 # print "\nPlaylists for " + str.format(playlist) + ":\n"
24
25 # # display each playlist to screen
26 # for p in video_feed.entry:
27 #     print p.title.text
```

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:

```
1 # iterate through the feed as you would with any other
2 for playlist_video_entry in playlist_video_feed.entry:
3     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:

```
1 for p in video_feed.entry:
2     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" algorithm/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. Cheers!

```
1 # Basic web services example
2
3 # import the client libraries
4 import gdata.youtube
5 import gdata.youtube.service
6
7 # YouTubeService() used to generate the object so that we can communicate with
   the YouTube API
8 youtube_service = gdata.youtube.service.YouTubeService()
9
10 # prompt the user to enter the Youtube User ID
11 user_id = raw_input("Please enter the user ID: ")
12
13 # setup the actual API call
14 url = "http://gdata.youtube.com/feeds/api/users/"
15 playlist_url = url + user_id + "/playlists"
16
```

```

17 # retrieve Youtube playlist and video list
18 playlist_feed = youtube_service.GetYouTubePlaylistVideoFeed(playlist_url)
19 print "\nPlaylists for " + str.format(user_id) + ":\n"
20
21 # display each playlist to screen
22 for playlist in playlist_feed.entry:
23     print playlist.title.text
24     playlistid = playlist.id.text.split('/')[1]
25     video_feed = youtube_service.GetYouTubePlaylistVideoFeed(playlist_id =
26     playlistid)
27     for video in video_feed.entry:
28         print "\t"+video.title.text

```

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

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>


```

1 # Twitter Web Services
2
3
4 import tweepy
5
6 consumer_key    = "<get_your_own>"
7 consumer_secret = "<get_your_own>"
8 access_token    = "<get_your_own>"
9 access_secret   = "<get_your_own>"
10
11 auth = tweepy.auth.OAuthHandler(consumer_key, consumer_secret)
12 auth.set_access_token(access_token, access_secret)
13 api = tweepy.API(auth)
14
15 tweets = api.search(q='#python')
16
17 # display results to screen
18 for t in tweets:
19     print t.created_at, t.text, "\n"

```

NOTE Add your keys and tokens in the above code before running.

If done correctly, this should output the tweets and the dates and times they were created:

```

1 2014-01-26 16:26:31 RT @arialdomartini: #pymacs is a crazy tool that allows the
   use of #python as external language for programming and extending #emacs
   http...:/
2
3 2014-01-26 16:22:43 The Rise And Fall of Languages in 2013
   http://t.co/KN4gaJASkn via @dr_dobbs #Python #Perl #C++
4
5 2014-01-26 16:21:28 #pymacs is a crazy tool that allows the use of #python as
   external language for programming and extending #emacs http://t.co/m2oU1ApDJp
6
7 2014-01-26 16:20:06 RT @PyCero91: Nuevo #MOOC muy interesante de programación
   con #Python de la Universidad de #Rice http://t.co/OXjiT38Wk0
8
9 2014-01-26 16:19:50 nice to start to email list only early bird sales for snakes
   on a train! http://t.co/A7bemXStDt #pycon #python
10
11 2014-01-26 16:15:31 RT @sarakhkendrew: this is super #python &gt; RT @davidwhogg:
   In which @jakevdp extolls #hackAAS and tells us about tooltips
   http://t.co/...s47SkX
12
13 2014-01-26 16:14:15 this is super #python &gt; RT @davidwhogg: In which @jakevdp
   extolls #hackAAS and tells us about tooltips http://t.co/s47SkXw9wa

```

```

14
15 2014-01-26 16:12:00 Most of this stuff doesn't bother me. Means that I should
    use Python more pycoders: What I Hate About Python http://t.co/3J4yScMLpE
    #python
16
17 2014-01-26 16:06:15 analog - analog - Log Analysis Utility pypi:
    http://t.co/aDakEdKQ4j www: https://t.co/1rRCffeH90 #python
18
19 2014-01-26 16:05:52 RT @pycoders: Parallelism in one line http://t.co/XuVjYXEiJ3
    #python
20
21 2014-01-26 16:00:04 RT @a_bhi_9: A 100 step for loop being iterated 100000000
    times for 4 different possibilities. I am pulling my 7 year old to the limit
    I ...be
22
23 2014-01-26 16:00:04 RT @a_bhi_9: Timing analysis of #iterators shows NoneType
    iterators perform way more better than list #generators. #Python #analysis
24
25 2014-01-26 15:54:07 #python #table #dive http://t.co/4wHxDgJCE5 Dive Into Python
26
27 2014-01-26 15:51:03 RT @takenji_ebooks: I think I'm going to write in #Python 3
    as much as possible rather than 2 from now on.
28
29 2014-01-26 15:49:04 RT @pycoders: What I Hate About Python
    http://t.co/URI1gh0sqA #python

```

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. Have fun!

Please Note:

Make sure you change your keys and token variables back to-

```

1 consumer_key      = "<get_your_own>"
2 consumer_secret   = "<get_your_own>"
3 access_token      = "<get_your_own>"
4 access_secret     = "<get_your_own>"

```

-before you PUSH to Github. *You do not want anyone else but you to know those keys.*

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 (requests) describes the type of information you can obtain from the API, and the second part details how to obtain (responses) said information.

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

1. Use the following URL to call (or invoke) the API: `http://maps.googleapis.com/maps/api/directions/output?`
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:

```
1 origin=Central+Park
2 destination=Times+Square
3 sensor=false
4 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` - and then call the API directly from your browser:

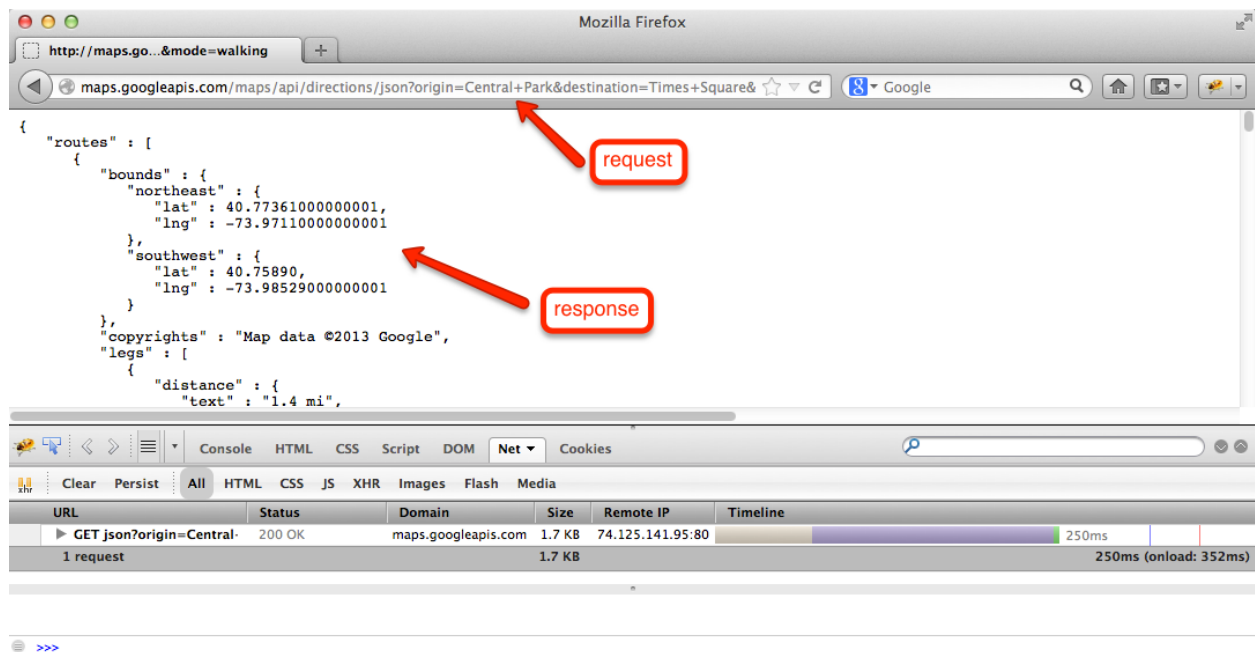


Figure 14.2: driving api

```

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>
>>>

```

Figure 14.3: directions

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:

All right. Let's breakdown the for loops:

```

1 for route in output['routes']:
2     for leg in route['legs']:
3         for step in leg['steps']:
4             print step['html_instructions']

```

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

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

```

1 for route in output['routes']:
2     for leg in route['legs']:
3         print leg['start_address']
4         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.

```

{
  "routes" : [
    {
      "bounds" : {
        "northeast" : {
          "lat" : 40.773610000000001,
          "lng" : -73.971100000000001
        },
        "southwest" : {
          "lat" : 40.75890,
          "lng" : -73.985290000000001
        }
      },
      "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" : -73.985290000000001
          },
          "start_address" : "Central Park, 14 East 60th Street, New York, NY 10022, USA",
          "start_location" : {
            "lat" : 40.773610000000001,
            "lng" : -73.971100000000001
          },
          "steps" : [
            {
              "distance" : {
                "text" : "0.2 mi",
                "value" : 328
              },
              "duration" : {
                "text" : "4 mins",
                "value" : 241
              },
              "end_location" : {
                "lat" : 40.770790000000001,
                "lng" : -73.97220
              },
              "html_instructions" : "Head \u003cb\u003esouth\u003c/b\u003e on \u003cb\u003eThe Mall\u003c/b\u003e",

```

Figure 14.4: json parsing

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" on the right side of the page. Immediately, you'll see the URL (or endpoint):

[http://api.rottentomatoes.com/api/public/v1.0/lists/movies/in_theaters.json?apikey=\[your_api_key\]](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 (or web services) 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 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:

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. *Again, make sure not to include the key within your file when you PUSH to Github.*

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](#).

⁴<http://south.aeracode.org/>

Register Your New Application

- **Name of your application (you can change it later)**
- **Application URL**

Figure 14.5: rotten tomatoes api

Now comes the fun part: building the program to actually GET the data, parsing the relevant data, and then dumping it into a database. We'll do this in iterations.

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

Code:

```
1 # Create a SQLite3 database and table
2
3
4 import sqlite3
5
6 with sqlite3.connect("movies.db") as connection:
7     c = connection.cursor()
8
9     # create a table
10    c.execute("""CREATE TABLE new_movies
11                (title TEXT, year INT, rating text,
12                 release text, runtime INT, critics_review INT,
13                 audience_review INT)""")
```

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

```
1 # GET data from Rotten Tomatoes, parse, and write to database
2
3
4 import json
5 import requests
6 import sqlite3
7
8 YOUR_OWN_KEY = 'GET_YOUR_OWN_KEY'
9 url = requests.get("http://api.rottentomatoes.com/api/public/v1.0/" +
10                    "lists/movies/in_theaters.json?apikey=%s" % (YOUR_OWN_KEY,))
```

```

11
12 # convert data from feed to binary
13 binary = url.content
14
15 # decode the json feed
16 output = json.loads(binary)
17
18 # grab the list of movies
19 movies = output["movies"]
20
21 with sqlite3.connect("movies.db") as connection:
22     c = connection.cursor()
23
24     # iterate through each movie and write to the database
25     for movie in movies:
26         c.execute("INSERT INTO new_movies VALUES(?, ?, ?, ?, ?, ?, ?)",
27                 (movie["title"], movie["year"], movie["mpaa_rating"],
28                 movie["release_dates"]["theater"], movie["runtime"],
29                 movie["ratings"]["critics_score"],
30                 movie["ratings"]["audience_score"]))
31
32     # retrieve data
33     c.execute("SELECT * FROM new_movies ORDER BY title ASC")
34
35     # fetchall() retrieves all records from the query
36     rows = c.fetchall()
37
38     # output the rows to the screen, row by row
39     for r in rows:
40         print r[0], r[1], r[2], r[3], r[4], r[5], r[6]

```

Make sure you add your API key into the value for the variable `YOUR_OWN_KEY`.

What happened?

1. We grabbed the endpoint URL with a GET request.
2. Converted the data to binary.
3. Decoded the JSON feed.
4. Then used a for loop to write the data to the database.
5. Finally we grabbed the data from the database and outputted it.

Nice, right?

Were you able to follow this code? Go through it a few more times. See if you can grab data from a different endpoint. *Practice!*

Chapter 15

web2py: QuickStart (continued...)

Sentiment Analysis

Continue to use the same web2py instance as before for the apps in this chapter as well.

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 a sense of how Twitter users feel about a particular topic. Companies use sentiment analysis to gain a deeper understanding about marketing campaigns, product lines, and the company itself.

NOTE: Sentiment analysis works best when it's conducted on a popular 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.¹ 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. Please follow the steps [here](#) to install. Make sure to add the cURL package when you get to the “Choosing Packages” step. Or scroll down to step 5 and use Hurl instead.

3. Unix users, and Windows users with Cygwin installed, test out the API in the terminal:

```
1 $ curl -d "text=great" http://text-processing.com/api/sentiment/  
2 {"probability": {"neg": 0.35968353095023886, "neutral":  
3   0.29896828324578045, "pos": 0.64031646904976114}, "label": "pos"}  
4  
5 $ curl -d "text=i hate apples" http://text-processing.com/api/sentiment/  
6 {"probability": {"neg": 0.65605365432549356, "neutral": 0.3611947857779943,  
7   "pos": 0.34394634567450649}, "label": "neg"}  
8  
9 $ curl -d "text=i usually like ice cream but this place is terrible"  
10 http://text-processing.com/api/sentiment/  
11 {"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"}
```

4. 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?

5. 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?

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>



http://text-processing.com/api/sentiment/

POST ☐ follow redirects

+ add param + set post body

text "i greatly dislike perl" ✕

☒ no auth ☐ HTTP basic

+ add header

Send

Figure 15.1: hurl

Requests

1. Alright, 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:

```
1 import requests
2
3 url = 'http://text-processing.com/api/sentiment/'
4 data = {'text': 'great'}
5 r = requests.post(url, data=data)
6 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.*

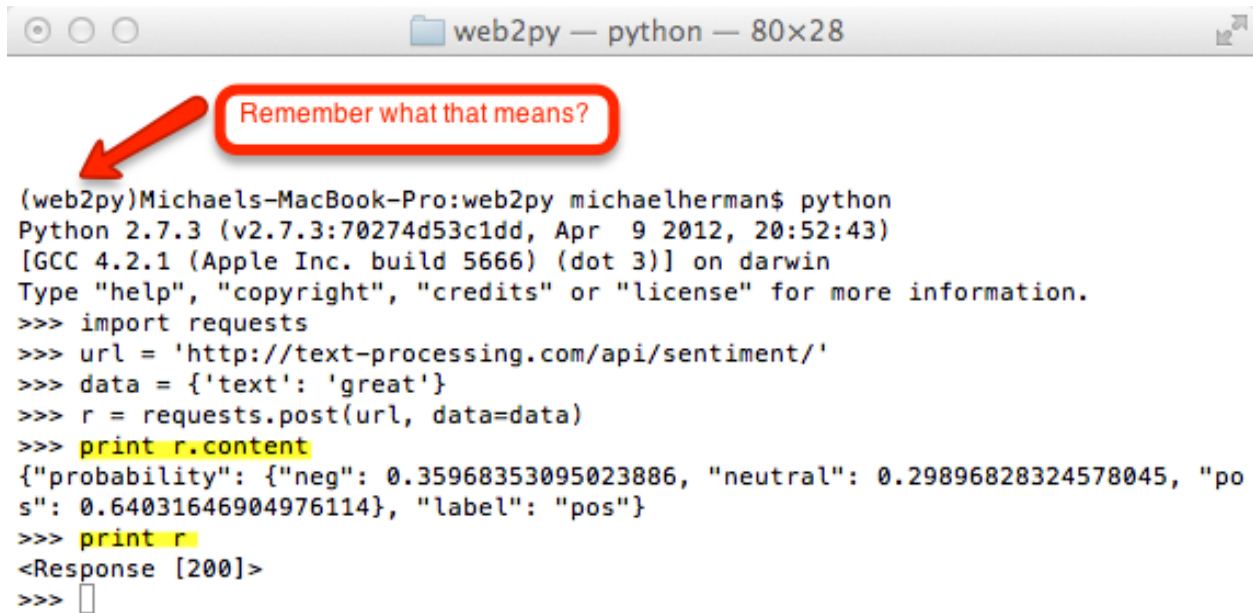
Remember the command `pip install requests`?

Now, test out the above code in your Shell:

Now, let's build the app for easily testing sentiment analysis. It's called Pulse.

Pulse

1. You know the drill: Activate your 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`.



```
(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]>
>>> 
```

Figure 15.2: requests pulse

3. Replace the code in the default controller with:

```
1 import requests
2
3 def index():
4     form=FORM(
5         TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
6         INPUT(_type='submit')).process()
7     if form.accepted:
8         redirect(URL('pulser',args=form.vars.pulse))
9     return dict(form=form)
10
11 def pulser():
12     text = request.args(0)
13     text = text.split('_')
14     text = ' '.join(text)
15     url = 'http://text-processing.com/api/sentiment/'
16     data = {'text': text}
17     r = requests.post(url, data=data)
18     return dict(text=text, r=r.content)
```

Now let's create some basic views.

4. *default/index.html*:

```
1 {{extend 'layout.html'}}
2 <center>
3 <br/>
```

```

4 <br/>
5 <h1>check a pulse</h1>
6 <h4>Just another Sentiment Analysis tool.</h4>
7 <br/>
8 <p>{{=form}}</p>
9 </center>

```

5. *default/pulser.html*:

```

1 {{extend 'layout.html'}}
2 <center>
3 <p>{{=text}}</p>
4 <br/>
5 <p>{{=r}}</p>
6 <br/>
7 <p><a href="/pulse/default/index">Another Pulse?</a><p>
8 </center>

```

6. Alright, test this out. Compare the results to the results using either cURL or Hurl to make sure all is set up correctly. If you did everything right you should have received an error that the requests module is not installed. Kill the server, and then install requests from the terminal:

```

1 $ pip install requests

```

Test it again. Did it work this time?

7. Now, let's finish cleaning up *pulser.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.

8. Update *default.py*:

```

1 import requests
2 import json
3
4 def index():
5     form=FORM(
6         TEXTAREA(_name='pulse', requires=IS_NOT_EMPTY()),
7         INPUT(_type='submit')).process()
8     if form.accepted:
9         redirect(URL('pulser',args=form.vars.pulse))
10    return dict(form=form)
11
12 def pulser():
13     text = request.args(0)
14     text = text.split('_')

```

```

15     text = ' '.join(text)
16
17     url = 'http://text-processing.com/api/sentiment/'
18     data = {'text': text}
19
20     r = requests.post(url, data=data)
21
22     binary = r.content
23     output = json.loads(binary)
24     label = output["label"]
25
26     return dict(text=text, label=label)

```

9. Update *default/pulser.html*:

```

1  {{extend 'layout.html'}}
2  <center>
3  <br/>
4  <br/>
5  <h1>your pulse</h1>
6  <h4>{{=text}}</h4>
7  <p>is</p>
8  <h4>{{=label}}</h4>
9  <br/>
10 <p><a href="/pulse/default/index">Another Pulse?</a><p>
11 </center>

```

10. Make it pretty. Update *layout.html*:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>check your pulse</title>
5  <meta charset="utf-8" />
6  <style type="text/css">
7  body {font-family: Arial, Helvetica, sans-serif; font-size:x-large;}
8  </style>
9  {{
10 middle_columns = {0:'span12',1:'span9',2:'span6'}
11 }}
12 {{block head}}{{end}}
13 </head>
14 <body>
15 <div class="{{=middle_columns}}">
16     {{block center}}
17     {{include}}
18     {{end}}

```

```
19 </div>
20 </body>
21 </html>
```

11. Test it out. Is it pretty? No. It's functional:

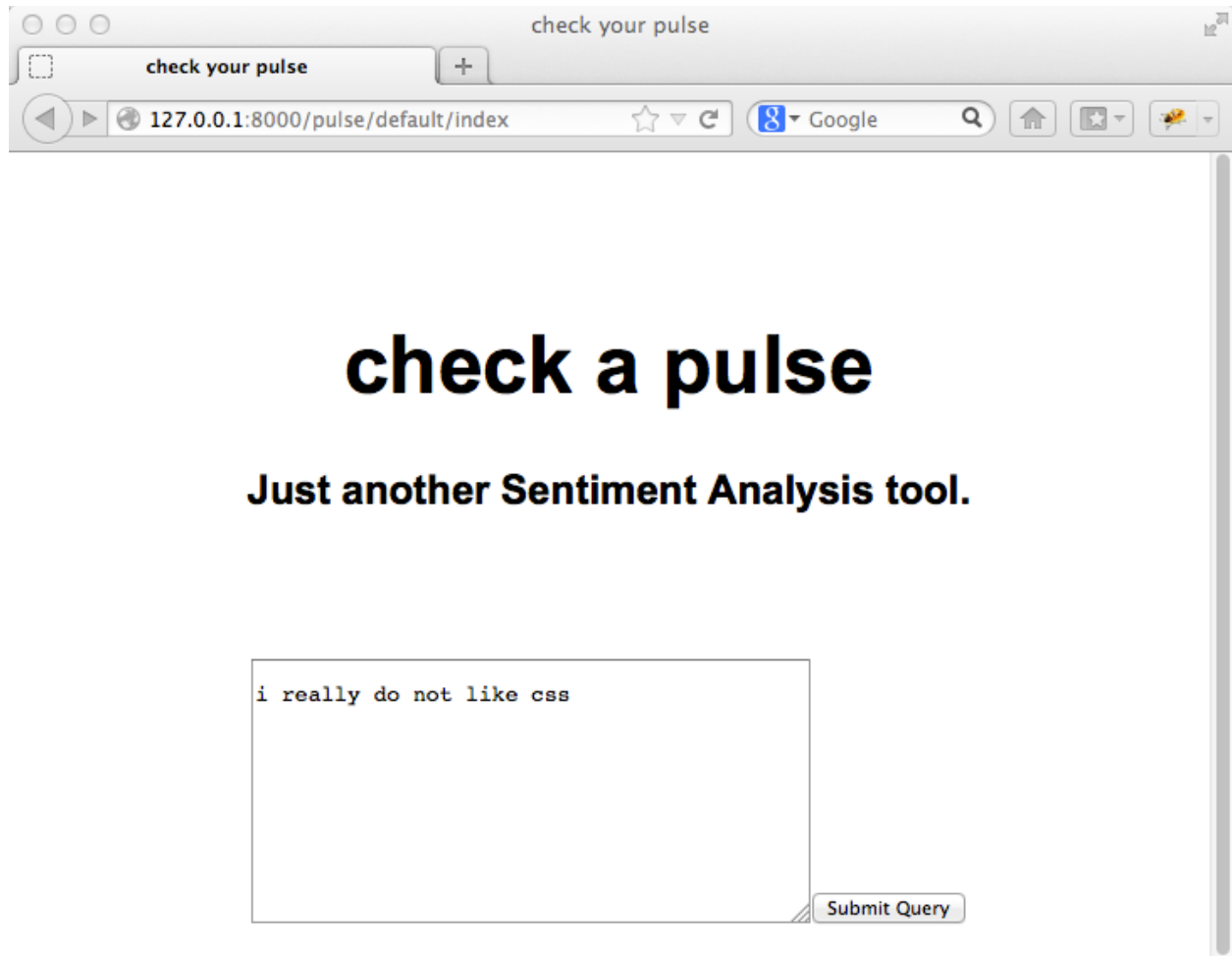


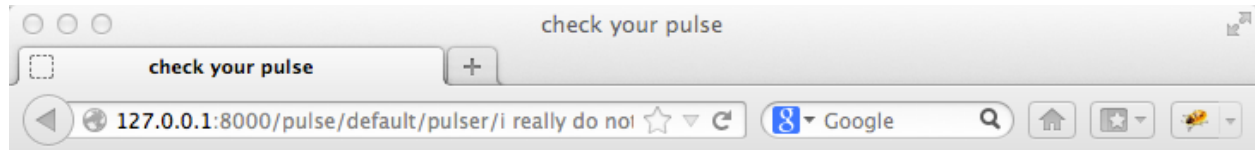
Figure 15.3: sentiment app 1

12. Make yours pretty.

13. Better?

Next steps

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 over time. Right now, the algorithm is static. Try entering the term “i like milk”. It’s negative, right?



your pulse

i really do not like css

is

neg

[Another Pulse?](#)

Figure 15.4: sentiment app 2

check a pulse

Just another Sentiment Analysis tool.

yay|

Submit

Figure 15.5: sentiment app 3

your pulse

yay

is

neutral

Another Pulse?

Figure 15.6: sentiment app 4

```
{"probability": {"neg": 0.50114184747628709, "neutral": 0.34259733533730058,  
"pos": 0.49885815252371291}, "label": "neg"}
```

Why is that?

Test out each word:

```
"i": {"probability": {"neg": 0.54885268027242828, "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.

Sentiment Analysis Expanded

Let's take Pulse to the next level by adding jQuery and AJAX. Don't worry if you've never worked with either jQuery or AJAX before, as web2py automates much of this. We'll also be covering both in a latter chapter. If you are interested in going through a quick primer on jQuery, check out [this](#) tutorial. If you do go through the tutorial, keep in mind that you will not have to *actually* code any Javascript or jQuery in web2py for this tutorial.

Let's begin.

NOTE: If you don't know what Javascript, jQuery, or AJAX is, take a look at [this](#) Stack-Overflow article. Still confused? Google it. We will cover Javascript and jQuery in a latter chapter. Feel free to jump ahead.

web2py and AJAX

1. web2py defines a function called `ajax()` built on top of jQuery, which essentially takes three arguments, a url, a list of ids, and a target id.

```
1 {{extend 'layout.html'}}
2 <h1>AJAX Test</h1>
3 <form>
4     <input type="number" id="key" name="key">
5     <input type="button" value="submit"
6         onclick="ajax('{{URL('data')}}', ['key'], 'target')">
7 </form>
8 <br>
9 <div id="target"></div>
```

Add this code to a new view in your Pulse app, which will be used just for testing. Call the view `test/index.html`.

This is essentially a regular form, but you can see the `ajax()` function within the button input - `onclick="ajax('{{URL('data')}}', ['key'], 'target')"`. The url will call a function within our controller (which we have yet to define), the data is grabbed from the input box, then the final results will be appended to `<div id="target"></div>`. Essentially, on the button click, we grab the value from the input box, send it to the `data()` function for *something* to happen, then added to the page between the `<div>` tag with the selector `id=target`.

Make sense? Let's get our controller setup. Create a new one called `test.py`.

2. Now update the following code to the new controller:

```
1 def index():
2     return dict()
3
4 def data():
5     return (int(request.vars.key)+10)
```

So when the input data is sent to the `data()` function, we are simply adding 10 to the number and then returning it.

3. Now let's update the parent/base template *index.html*:

```
1 <!DOCTYPE html>
2 <head>
3   <title>AJAX Test</title>
4   <script src="{%=URL('static','js/modernizr.custom.js')%}"></script>
5   <!-- include stylesheets -->
6   {{
7     response.files.insert(0,URL('static','css/web2py.css'))
8     response.files.insert(1,URL('static','css/bootstrap.min.css'))
9     response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
10    response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
11  }}
12  {{include 'web2py_ajax.html'}}
13  {{
14    middle_columns = {0:'span12',1:'span9',2:'span6'}
15  }}
16  <noscript><link href="{%=URL('static','css/web2py_bootstrap_nojs.css')%}" rel="stylesheet" type="text/css"
17    /></noscript>
18  {{block head}}{{end}}
19 </head>
20 <body>
21   <div class="container">
22     <section id="main" class="main row">
23       <div class="{%=middle_columns%}">
24         {{block center}}
25         {{include}}
26         {{end}}
27       </div>
28     </section><!--/main-->
29   </div> <!-- /container -->
30   <script src="{%=URL('static','js/bootstrap.min.js')%}"></script>
31   <script src="{%=URL('static','js/web2py_bootstrap.js')%}"></script>
32 </body>
</html>
```

4. Test it out. Make sure to enter an integer. You should see this (if you entered 15, of course):

Did you notice that when you click the button the page does not refresh? This is what makes AJAX, well, AJAX: To the end user, the process is seamless. Web apps send data from the client to the server, which is then sent back to the client without interfering with the behavior of the page.

Let's apply this to our Pulse app.

AJAX Test

25

Figure 15.7: sentiment app 5

Adding AJAX to Pulse

1. Update the controller:

```
1 import requests
2
3 def index():
4     return dict()
5
6 def pulse():
7     session.m=[]
8     if request.vars.sentiment:
9         text = request.vars.sentiment
10        text = text.split('_')
11        text = ' '.join(text)
12        url = 'http://text-processing.com/api/sentiment/'
13        data = {'text': text}
14        r = requests.post(url, data=data)
15        session.m.append(r.content)
16    session.m.sort()
17    return text, TABLE(*[TR(v) for v in session.m]).xml()
```

Here, we're simply grabbing the text from the input, running the analysis, appending the results to a list, and then returning the original text along with the results.

2. Update the view, *default/index.html*:

```
1 {{extend 'layout.html'}}
2 <h1>check your pulse</h1>
3 <form>
4   <input type="text" id="sentiment" name="sentiment">
5   <input type="button" value="submit"
6     onclick="ajax('{{URL('pulse')}}', ['sentiment'], 'target')">
7 </form>
8 <br>
9 <div id="target"></div>
```

Nothing new here.

3. Test it out. You should see something like:

check your pulse

web2py

```
{"probability": {"neg": 0.50955199890675162, "neutral": 0.59508906140405482, "pos": 0.49044800109324838}, "label": "neutral"}
```

Figure 15.8: sentiment app 6

Additional Features

Now, let's expand our app so that we can enter two text inputs for comparison.

1. Update the controller:

```
1 import requests
2
3 def index():
4     return dict()
5
6 def pulse():
7
8     session.m=[]
9     url = 'http://text-processing.com/api/sentiment/'
10
```

```

11     # first item
12     text_first = request.vars.first_item
13     text_first = text_first.split('_')
14     text_first = ' '.join(text_first)
15     data_first = {'text': text_first}
16     r_first = requests.post(url, data=data_first)
17     session.m.append(r_first.content)
18
19     # second item
20     text_second = request.vars.second_item
21     text_second = text_second.split('_')
22     text_second = ' '.join(text_second)
23     data_second = {'text': text_second}
24     r_second = requests.post(url, data=data_second)
25     session.m.append(r_second.content)
26
27     session.m.sort()
28     return text_first, text_second, TABLE(*[TR(v) for v in session.m]).xml()

```

This performs the exact same actions as before, only with two inputs instead of one.

2. Update the view:

```

1  {{extend 'layout.html'}}
2  <h1>pulse</h1>
3  <p>comparisons using sentiment</p>
4  <br>
5  <form>
6      <input type="text" id="first_item" name="first_item" placeholder="enter
7          first item...">
8      <br>
9      <input type="text" id="second_item" name="second_item"
10         placeholder="enter second item...">
11      <br>
12      <input type="button" value="submit"
13         onclick="ajax('{%=URL('pulse')%}', ['first_item', 'second_item'], 'target')">
14  </form>
15
16  <br>
17  <div id="target"></div>

```

Notice how we're passing two items to the pulse URL, ['first_item', 'second_item'].

3. Test.

Now, let's make this look a little nicer.

4. Only display the item that has the higher (more positive) sentiment. To do this update the controller:

pulse

comparisons using sentiment

rubypython

```
{"probability": {"neg": 0.42554563711374715, "neutral": 0.7963320491099668, "pos": 0.57445436288625285}, "label": "neutral"}  
{"probability": {"neg": 0.50471472774640502, "neutral": 0.59508906140405482, "pos": 0.49528527225359503}, "label": "neutral"}
```

Figure 15.9: sentiment app 7

```
1 import requests
2 import json
3
4 def index():
5     return dict()
6
7 def process(my_list):
8
9     # analyze first item
10    binary_first = my_list[1]
11    output_first = json.loads(binary_first)
12    label_first = output_first["label"]
13
14    # analyze second item
15    binary_second = my_list[3]
16    output_second = json.loads(binary_second)
17    label_second = output_second["label"]
18
19    # logic
20    if label_first == "pos":
21        if label_second != "pos":
22            return my_list[0]
23        else:
24            if output_first["probability"]["pos"] >
25                output_second["probability"]["pos"]:
26                return my_list[0]
27            else:
28                return my_list[2]
29    elif label_first == "neg":
30        if label_second != "neg":
```

```

30         return my_list[2]
31     else:
32         if output_first["probability"]["neg"] <
output_second["probability"]["neg"]:
33             return my_list[0]
34         else:
35             return my_list[2]
36     elif label_first == "neutral":
37         if label_second == "pos":
38             return my_list[2]
39         elif label_second == "neg":
40             return my_list[0]
41         else:
42             if output_first["probability"]["pos"] >
output_second["probability"]["pos"]:
43                 return my_list[0]
44             else:
45                 return my_list[2]
46
47 def pulse():
48
49     session.m=[]
50     url = 'http://text-processing.com/api/sentiment/'
51
52     # first item
53     text_first = request.vars.first_item
54     text_first = text_first.split('_')
55     text_first = ' '.join(text_first)
56     session.m.append(text_first)
57     data_first = {'text': text_first}
58     r_first = requests.post(url, data=data_first)
59     session.m.append(r_first.content)
60
61     # second item
62     text_second = request.vars.second_item
63     text_second = text_second.split('_')
64     text_second = ' '.join(text_second)
65     session.m.append(text_second)
66     data_second = {'text': text_second}
67     r_second = requests.post(url, data=data_second)
68     session.m.append(r_second.content)
69
70     winner = process(session.m)
71
72     return "The winner is {}".format(winner)

```

Although there is quite a bit more code here, the logic is simple. Walk through the program,

slowly, stating what each line is doing. Do this out loud. Keep going through it until it makes sense. Also, make sure to test it to ensure it's returning the right values. You can compare the results with the results found [here](#).

Next, let's clean up the output.

1. First update the layout template with a bootstrap stylesheet as well as some custom styles:

```
1 <!DOCTYPE html>
2 <head>
3   <title>AJAX Test</title>
4   <script src="{%=URL('static','js/modernizr.custom.js')%}"></script>
5   <!-- include stylesheets -->
6   {{
7     response.files.insert(0,URL('static','css/web2py.css'))
8     response.files.insert(1,URL('static','css/bootstrap.min.css'))
9     response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
10    response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
11  }}
12  {{include 'web2py_ajax.html'}}
13  {{
14    middle_columns = {0:'span12',1:'span9',2:'span6'}
15  }}
16  <link rel="stylesheet"
17    href="//netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/bootstrap.min.css">
18  <noscript><link href="{%=URL('static',
19    'css/web2py_bootstrap_nojs.css')%}" rel="stylesheet" type="text/css"
20    /></noscript>
21  {{block head}}{{end}}
22 </head>
23 <body>
24   <div class="container">
25     <div class="jumbotron">
26       <div class="{%=middle_columns%}">
27         {{block center}}
28         {{include}}
29         {{end}}
30       </div>
31     </div>
32   </div> <!-- /container -->
33   <script src="{%=URL('static','js/bootstrap.min.js')%}"></script>
34   <script src="{%=URL('static','js/web2py_bootstrap.js')%}"></script>
35 </body>
36 </html>
```

2. Now update the child view:

```

1  {{extend 'layout.html'}}
2  <h1>pulse</h1>
3  <p class="lead">comparisons using sentiment</p>
4  <br>
5  <form role="form">
6      <div class="form-group">
7          <input type="text" id="first_item" class="form-control" name
8              ="first_item" placeholder="enter first item...">
9          <br>
10         <input type="text" id="second_item" class="form-control" name
11             ="second_item" placeholder="enter second item...">
12         <br>
13         <input type="button" class="btn btn-success" value="submit"
14             onclick="ajax('{{URL('pulse')}}', ['first_item', 'second_item'], 'target')">
15     </div>
16 </form>
<br>
<h2 id="target"></h2>

```

We simply added some bootstrap classes to make it look much nicer.

Looking good!

pulse

comparisons using sentiment

The winner is python!

Figure 15.10: sentiment app 8.png

Movie Suggestor

Let's take this to the next level and create a movie suggester. Essentially, we'll pull data from the Rotten Tomatoes API to grab movies that are playing then display the sentiment of each.

Setup

1. Create a new app called "movie_suggest".
2. Replace the code in the controller, *default.py* with:

```
1 import requests
2 import json
3
4 def index():
5     return dict()
6
7 def grab_movies():
8     session.m = []
9     YOUR_OWN_KEY = 'GET_YOUR_OWN_KEY'
10    url = requests.get("http://api.rottentomatoes.com/api/public/v1.0/" +
11        "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
12    binary = url.content
13    output = json.loads(binary)
14    movies = output['movies']
15    for movie in movies:
16        session.m.append(movie["title"])
17    session.m.sort()
18    return TABLE(*[TR(v) for v in session.m]).xml()
```

NOTE: Make sure you add your API key into the value for the variable YOUR_OWN_KEY. Forgot your key? Return to the last chapter to find out how to obtain a new one.

In this script, we're using the Rotten Tomatoes API to grab the movies currently playing.

3. Test this out in your shell to see exactly what's happening:

```
1 >>> import requests
2 >>> import json
3 >>> YOUR_OWN_KEY = 'abha3gx3p42czdrswkejmyxm'
4 >>> url = requests.get("http://api.rottentomatoes.com/api/public/v1.0/" +
5 ...    "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
6 >>> binary = url.content
7 >>> output = json.loads(binary)
```

```

8 >>> movies = output['movies']
9 >>> for movie in movies:
10 ...     movie["title"]
11 ...
12 u'The Lego Movie'
13 u'Pompeii'
14 u'Son Of God'
15 u'Non-Stop'
16 u'3 Days To Kill'
17 u'The Monuments Men'
18 u'RoboCop'
19 u'Ride Along'
20 u'About Last Night'
21 u'Endless Love'
22 u'Anchorman 2: The Legend Continues'
23 u'Lone Survivor'
24 u'American Hustle'
25 u'The Nut Job'
26 u"Winter's Tale"
27 u'The Wind Rises'

```

4. Update the view, *default/index.html*:

```

1 {{extend 'layout.html'}}
2 <h1>suggest-a-movie</h1>
3 <p>use sentiment to find that perfect movie</p>
4 <br>
5 <form>
6     <input type="button" value="Get Movies"
7         onclick="ajax('{{URL('grab_movies')}}',[],'target')">
8 </form>
9 <br>
10 <div id="target"></div>

```

5. Update the parent view, *layout.html*:

```

1 <!DOCTYPE html>
2 <head>
3 <title>suggest-a-movie</title>
4 <script src="{{URL('static','js/modernizr.custom.js')}}"></script>
5 <!-- include stylesheets -->
6 {{
7 response.files.insert(0,URL('static','css/web2py.css'))
8 response.files.insert(1,URL('static','css/bootstrap.min.css'))
9 response.files.insert(2,URL('static','css/bootstrap-responsive.min.css'))
10 response.files.insert(3,URL('static','css/web2py_bootstrap.css'))
11 }}

```

```

12 {{include 'web2py_ajax.html'}}
13 {{
14 middle_columns = {0:'span12',1:'span9',2:'span6'}
15 }}
16 <link rel="stylesheet"
17     href="//netdna.bootstrapcdn.com/bootswatch/3.1.1/yeti/bootstrap.min.css">
18 <noscript><link href="{%=URL('static', 'css/web2py_bootstrap_nojs.css')%}"
19     rel="stylesheet" type="text/css" /></noscript>
20 {{block head}}{{end}}
21 </head>
22 <body>
23 <div class="container">
24     <div class="jumbotron">
25         <div class="{%=middle_columns%}">
26             {{block center}}
27             {{include}}
28             {{end}}
29         </div>
30     </div>
31 </div> <!-- /container -->
32 <script src="{%=URL('static', 'js/bootstrap.min.js')%}"></script>
33 <script src="{%=URL('static', 'js/web2py_bootstrap.js')%}"></script>
34 </body>
35 </html>

```

6. Test it out. You should see something similar to this (depending on which movies are in the theater, of course):

Add Sentiment

Let's now determine the sentiment of each movie.

1. Update the controller:

```

1 import requests
2 import json
3
4 def index():
5     return dict()
6
7 def grab_movies():
8     session.m=[]
9     YOUR_OWN_KEY = 'abha3gx3p42czdrswkejmyxm'
10    url = requests.get("http://api.rottentomatoes.com/api/public/v1.0/" +
11        "lists/movies/in_theaters.json?apikey={}".format(YOUR_OWN_KEY))
12    binary = url.content

```


suggest-a-movie

use sentiment to find that perfect movie

Get Movies

3 Days To Kill
About Last Night
American Hustle
Anchorman 2: The Legend Continues
Endless Love
Lone Survivor
Non-Stop
Pompeii
Ride Along
RoboCop
Son Of God
The Lego Movie
The Monuments Men
The Nut Job
The Wind Rises
Winter's Tale

Figure 15.11: suggest movie

```

13     output = json.loads(binary)
14     movies = output['movies']
15     for movie in movies:
16         session.m.append(pulse(movie["title"]))
17     session.m.sort()
18     return TABLE(*[TR(v) for v in session.m]).xml()
19
20 def pulse(movie):
21     text = movie.replace('_', ' ')
22     url = 'http://text-processing.com/api/sentiment/'
23     data = {'text': text}
24     r = requests.post(url, data=data)
25     binary = r.content
26     output = json.loads(binary)
27     label = output["label"]
28     pos = output["probability"]["pos"]
29     neg = output["probability"]["neg"]
30     neutral = output["probability"]["neutral"]
31     return text, label, pos, neg, neutral

```

Here we are simply taking each name and passing them as an argument into the pulse() function where we are calculating the sentiment.

Update Styles

1. Update the child view with some bootstrap styles:

```

1  {{extend 'layout.html'}}
2  <h1>suggest-a-movie</h1>
3  <p>use sentiment to find that perfect movie</p>
4  <br>
5  <form>
6      <input type="button" class="btn btn-primary" value="Get Movies"
7          onclick="ajax('{%=URL('grab_movies')%}', [], 'target')">
8  </form>
9  <br>
10 <table class="table table-hover">
11     <thead>
12         <tr>
13             <th>Movie Title</th>
14             <th>Label</th>
15             <th>Positive</th>
16             <th>Negative</th>
17             <th>Neutral</th>
18         </tr>
19     </thead>
20     <tbody id="target"></tbody>

```

suggest-a-movie

use sentiment to find that perfect movie

Get Movies

3 Days To Kill	neutral0.3021240014340.6978759985660.96141481597
About Last Night	neutral0.4894781079380.5105218920620.777012481552
American Hustle	neutral0.6184005631970.3815994368030.663263611318
Anchorman 2: The Legend Continues	neutral0.47670093557 0.52329906443 0.800510393573
Endless Love	pos 0.5229853853920.4770146146080.462337287624
Lone Survivor	neutral0.5241479270120.4758520729880.757142151876
Non-Stop	neutral0.4316059211660.5683940788340.7897784246
Pompeii	neutral0.4904480010930.5095519989070.595089061404
Ride Along	neutral0.6104252652 0.3895747348 0.538105042179
RoboCop	neutral0.5026478605770.4973521394230.595089061404
Son Of God	neutral0.5683231581460.4316768418540.881602969824
The Lego Movie	neg 0.4881549487380.5118450512620.40276071531
The Monuments Men	neutral0.5057370719790.4942629280210.681055003825
The Nut Job	neutral0.5820900141920.4179099858080.774520853746
The Wind Rises	neutral0.43051395448 0.56948604552 0.552275748981
Winter's Tale	pos 0.5909380730380.4090619269620.417161505192

Figure 15.12: suggest movie again

suggest-a-movie

use sentiment to find that perfect movie

Get Movies

Movie Title	Label	Positive	Negative	Neutral
3 Days To Kill	neutral	0.302124001434	0.697875998566	0.96141481597
About Last Night	neutral	0.489478107938	0.510521892062	0.777012481552
American Hustle	neutral	0.618400563197	0.381599436803	0.663263611318
Anchorman 2: The Legend Continues	neutral	0.47670093557	0.52329906443	0.800510393573
Endless Love	pos	0.522985385392	0.477014614608	0.462337287624
Lone Survivor	neutral	0.524147927012	0.475852072988	0.757142151876
Non-Stop	neutral	0.431605921166	0.568394078834	0.7897784246
Pompeii	neutral	0.490448001093	0.509551998907	0.595089061404
Ride Along	neutral	0.6104252652	0.3895747348	0.538105042179
RoboCop	neutral	0.502647860577	0.497352139423	0.595089061404
Son Of God	neutral	0.568323158146	0.431676841854	0.881602969824

Figure 15.13: suggest movie final

Think about what else you could do? Perhaps you could highlight movies that are positive. Check the web2py [documentation](#) for help.

Deploy to Heroku

Let's deploy this app to Heroku. Before you do that, you probably want to move this app out of the “realpython” directory. *Remember: a repo within a repo is never any good.* Simply follow the instructions after you move the app found [here](#).

Check out my app at http://obscure-citadel-4389.herokuapp.com/movie_suggest/default/index.

Cheers!

Blog App

Let's recreate the blog app we created in Flask. 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. Please.)

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).² To keep things simple, an ORM is a subset of a DAL. Both are used to map database functions to Python objects. Check out the following [StackOverflow article](#) for more info.

Open up *db.py* and append the following code:

```
1 db.define_table('blog_posts',
2     Field('title', notnull=True),
3     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:

```
1 timestamp: 2014-03-03T21:04:08.758346
2 CREATE TABLE blog_posts(
3     id INTEGER PRIMARY KEY AUTOINCREMENT,
4     title CHAR(512) NOT NULL,
5     post TEXT NOT NULL
6 );
7 success!
```

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

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:

```
1 def index():
2     form=SQLFORM(db.blog_posts)
3     if form.process().accepted:
4         response.flash = "Post accepted - cheers!"
5     elif form.errors:
6         response.flash = "Post not accepted - fix the error(s)."
```

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 within the views.

View

Edit the view *default/index.html* with the following code:

```
1 {{extend 'layout.html'}}
2 <h1>Add a Post</h1>
3 {{=form}}
4 <h1>Current Posts</h1>
5 <br/>
6 <table>
7     <tr><td><h3>Title</h3></td><td><h3>Post</h3></td></tr>
8     {{for p in posts:}}
9     <tr><td>{{=(A(p.title))}}</td><td>{{=(A(p.post))}}</td></tr>
10    {{pass}}
11 </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. Read over it again. 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*:

```
1 @auth.requires_login()
2 def index():
3     form=SQLFORM(db.blog_posts)
4     if form.process().accepted:
5         response.flash = "Post accepted - cheers!"
6     elif form.errors:
7         response.flash = "Post not accepted - fix the error(s)."
8     else:
9         response.flash = "Please fill out the form - thank you!"
10    posts = db().select(db.blog_posts.ALL)
11    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 web2py next chapter.

Chapter 16

web2py: py2manager

Introduction

In the last chapters 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 pattern.

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, notes, and other information associated with a particular project and view other employees' profiles.

Regarding the actual data model ...

1. Each company consists of a company name, email, phone number, and URL.
2. 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.
3. Finally, the 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
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 (or logic), the controller then performs an action, such as querying or modifying the database. Once the data is found or updated, the controller then passes the results back to the views, which are rendered into HTML for the end the user to see (response).

Most modern web frameworks utilize the 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 our application.

Setup

1. Create a new directory called “py2manager” to house your app.
2. Created and activate a new virtualenv.
3. 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.
4. Create a new app from your terminal:

```
1 $ python web2py.py -S py2manager
```

After this project is created, we are left in the Shell. Go ahead and exit it.

5. Within you terminal, navigate to the “Applications” directory, then into the “py2manager” directory. This directory holds all of your application’s files.

Sublime Text

Instead of using the internal web2py IDE for development, like in the previous examples, we’re going to again use the powerful text editor Sublime Text. This will help us keep our project structure organized.

Load the entire project into Sublime by selecting “Project”, then “Add folder to project.” Navigate to your “/py2manager/applications/py2manager” directory. Select the directory. Click open.

You should now have the entire application structure (files and folders) in Sublime. Take a look around. Open the “Models”, “Views”, and “Controllors” 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 them quickly.

Let’s start developing!

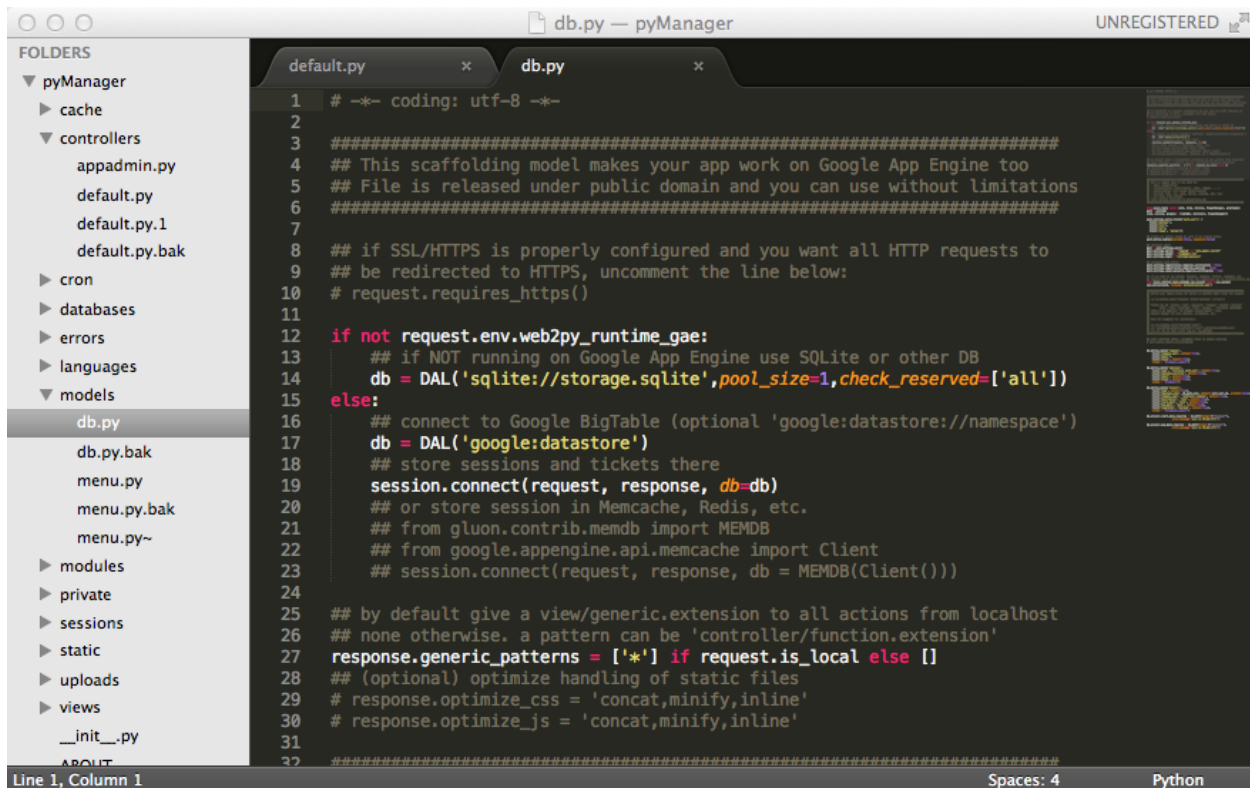


Figure 16.1: sublime

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 a DAL easier to work with than an ORM. If not, learning the syntax is no more difficult than an ORM.

ORM:

```
1 class User(db.Model):
2     __tablename__ = 'users'
3     name = db.Column(db.String, unique=True, nullable=False)
4     email = db.Column(db.String, unique=True, nullable=False)
5     password = db.Column(db.String, nullable=False)
```

DAL:

```
1 db.define_table('users',
2     Field('name', 'string', unique=True, notnull=True),
3     Field('email', 'string', unique=True, notnull=True),
4     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 flavor uses functions. Both are portable among many different relational database engines. Meaning they are database agnostic, so 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 with your database from the web2py Shell. The following is a quick, unrelated example:

1. In your terminal navigate to the project root directory, “/web2py/py2manager”, then run the following command:

```
1 $ python web2py.py --shell=py2manager
```

2. Run the following DAL commands: ²

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

```

1 >>> db = DAL('sqlite://storage.sqlite',pool_size=1,check_reserved=['all'])
2 >>> db.define_table('special_users', Field('name'), Field('email'))
3 <Table special_users (id,name,email)>
4 >>> db.special_users.insert(id=1, name="Alex", email="hey@alex.com")
5 1L
6 >>> db.special_users.bulk_insert([{'name': 'Alan', 'email': 'a@a.com'},
7 ... {'name': 'John', 'email': 'j@j.com'}, {'name': 'Tim',
8   'email': 't@t.com'}])
9 [2L, 3L, 4L]
10 >>> db.commit()
11 >>> for row in db().select(db.special_users.ALL):
12 ...     print row.name
13 ...
14 Alex
15 Alan
16 John
17 Tim
18 >>> for row in db().select(db.special_users.ALL):
19 ...     print row
20 ...
21 <Row {'name': 'Alex', 'email': 'hey@alex.com', 'id': 1}>
22 <Row {'name': 'Alan', 'email': 'a@a.com', 'id': 2}>
23 <Row {'name': 'John', 'email': 'j@j.com', 'id': 3}>
24 <Row {'name': 'Tim', 'email': 't@t.com', 'id': 4}>
25 >>> db.special_users.drop()
>>> exit()

```

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 using for loops before dropping (deleting) the table and exiting the Shell.

web2py Admin

Now, as I mentioned in the last chapter, web2py has a default for everything. These are the default values for each table field.

For example:

```

1 Field(name, 'string', length=None, default=None,
2   required=False, requires='<default>',
3   ondelete='CASCADE', notnull=False, unique=False,
4   uploadfield=True, widget=None, label=None, comment=None,
5   writable=True, readable=True, update=None, authorize=None,
6   autodelete=False, represent=None, compute=None,
7   uploadfolder=os.path.join(request.folder, 'uploads'),
8   uploadseparate=None, uploadfs=None)

```

```
py2manager — bash — 80x27

>>> 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$
```

Figure 16.2: web2py shell

So, our `company_name` field would by default be a string value, it is not required (meaning it is not mandatory to enter a company name), and, finally, it does not have to be a unique value. Keep these defaults in mind when you are creating your database tables.

Let's create the model for our application. Navigate into the “applications” directory, and then to the “py2manager” directory.

1. Create a new file to define your database schema called `db_tasks.py` within the “Models” directory. Add the following code:

```
1 db.define_table('company',
2     Field('company_name', notnull=True, unique=True),
3     Field('email'),
4     Field('phone', notnull=True),
5     Field('url'),
6     format = '%(company_name)s')
7
8 db.company.email.requires=IS_EMAIL()
9 db.company.url.requires=IS_EMPTY_OR(IS_URL())
10
11 db.define_table('project',
12     Field('name', notnull=True),
13     Field('employee_name', db.auth_user, default=auth.user_id),
14     Field('company_name', 'reference company', notnull=True),
15     Field('description', 'text', notnull=True),
16     Field('start_date', 'date', notnull=True),
17     Field('due_date', 'date', notnull=True),
18     Field('completed', 'boolean', notnull=True),
19     format = '%(company_name)s')
20
21 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 company. 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, his/her user information will automatically be added to the database. Save the file.

2. Navigate back to your project root. Fire up web2py in your terminal:

```
1 $ python web2py.py -a 'PUT_YOUR_PASSWORD_HERE' -i 127.0.0.1 -p 8000
```

Make sure to replace ‘PUT_YOUR_PASSWORD_HERE’ with an actual password -
e.g., `python web2py.py -a admin -i 127.0.0.1 -p 8000`

3. Navigate to <http://localhost:8000/> in your browser. Make your way 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:

```
1 timestamp: 2014-03-04T00:44:44.436419
2 CREATE TABLE special_users(
3     id INTEGER PRIMARY KEY AUTOINCREMENT,
4     name CHAR(512),
5     email CHAR(512)
6 );
7 success!
8 DROP TABLE special_users;
9 success!
10 timestamp: 2014-03-04T01:01:27.408170
11 CREATE TABLE auth_user(
12     id INTEGER PRIMARY KEY AUTOINCREMENT,
13     first_name CHAR(128),
14     last_name CHAR(128),
15     email CHAR(512),
16     password CHAR(512),
17     registration_key CHAR(512),
18     reset_password_key CHAR(512),
19     registration_id CHAR(512)
20 );
21 success!
22 timestamp: 2014-03-04T01:01:27.411689
23 CREATE TABLE auth_group(
24     id INTEGER PRIMARY KEY AUTOINCREMENT,
25     role CHAR(512),
26     description TEXT
27 );
28 success!
29 timestamp: 2014-03-04T01:01:27.413847
30 CREATE TABLE auth_membership(
31     id INTEGER PRIMARY KEY AUTOINCREMENT,
32     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
33     group_id INTEGER REFERENCES auth_group (id) ON DELETE CASCADE
34 );
35 success!
36 timestamp: 2014-03-04T01:01:27.416469
37 CREATE TABLE auth_permission(
38     id INTEGER PRIMARY KEY AUTOINCREMENT,
39     group_id INTEGER REFERENCES auth_group (id) ON DELETE CASCADE,
40     name CHAR(512),
41     table_name CHAR(512),
42     record_id INTEGER
```

```

43 );
44 success!
45 timestamp: 2014-03-04T01:01:27.419046
46 CREATE TABLE auth_event(
47     id INTEGER PRIMARY KEY AUTOINCREMENT,
48     time_stamp TIMESTAMP,
49     client_ip CHAR(512),
50     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
51     origin CHAR(512),
52     description TEXT
53 );
54 success!
55 timestamp: 2014-03-04T01:01:27.421675
56 CREATE TABLE auth_cas(
57     id INTEGER PRIMARY KEY AUTOINCREMENT,
58     user_id INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
59     created_on TIMESTAMP,
60     service CHAR(512),
61     ticket CHAR(512),
62     renew CHAR(1)
63 );
64 success!
65 timestamp: 2014-03-04T01:01:27.424647
66 CREATE TABLE company(
67     id INTEGER PRIMARY KEY AUTOINCREMENT,
68     company_name CHAR(512) NOT NULL UNIQUE,
69     email CHAR(512),
70     phone CHAR(512) NOT NULL,
71     url CHAR(512)
72 );
73 success!
74 timestamp: 2014-03-04T01:01:27.427338
75 CREATE TABLE project(
76     id INTEGER PRIMARY KEY AUTOINCREMENT,
77     name CHAR(512) NOT NULL,
78     employee_name INTEGER REFERENCES auth_user (id) ON DELETE CASCADE,
79     company_name INTEGER REFERENCES company (id) ON DELETE CASCADE,
80     description TEXT NOT NULL,
81     start_date DATE NOT NULL,
82     due_date DATE NOT NULL,
83     completed CHAR(1) NOT NULL
84 );
85 success!

```

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. By using the `format` attribute references will not show up by the id - but by the preferred field. You'll see *exactly* what that means in a second.

4. Open <http://localhost:8000/py2manager/default/user/register>, then register yourself as a new user.
5. Next, let's setup a new company and an associated project by navigating to <http://localhost:8000/py2manager/appadmin/index>. Click the relevant tables and add in data for the company and project. Make sure *not* to mark the project as complete.

NOTE: 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](#).

Homework

- Download the [web2py cheatsheet](#). Read it.

URL Routing

Controllers describe the application/business logic and workflow in order to link the user with the application through the request/response cycle. 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, navigate to the login [page](#) within your app and log in with the user that you created. When you clicked the “Login” 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 database via the model. 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:

```
1 def index():  
2     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:

```
1 http://www.yoursite.com/hello/default/index.html
```

This is also the default URL routing method:

```
1 http://www.yousite.com/application_name/controller_name/function_name.html
```

You can customize the URL routing methods by adding a *routes.py* file. If you wanted to remove the `controller_name` from the url, for example, add the following code newly created file:

```
1 routers = dict(  
2     BASE = dict(  
3         default_application='py2manager',  
4     )  
5 )
```

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

Make those changes.

Test this out. Restart the server. Navigate to the login page again: <http://localhost:8000/py2manager/default/user/login>. Well, since we made those changes, we can now access the same page from this url <http://localhost:8000/user/login>.

SEE ALSO: 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*:

```
1 @auth.requires_login()
2 def index():
3     project_form = SQLFORM(db.project).process()
4     projects = db(db.project).select()
5     users = db(db.auth_user).select()
6     companies = db(db.company).select()
7     return locals()
```

Here we are displaying 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. We just need to update the views, organize the *index.html* page, and update the layout and styles.

Initial Views

Views (or templates) describe how the subsequent response, from a request, should be translated to the user using mostly a combination of a templating engine, HTML, Javascript, and CSS.

Some major components of the templates include:

1. Template Engine: Template engines are used for embedding Python code directly 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 as well as `if` statements.

For example:

- Add a basic function to the controller:

```
1 def tester():
2     return locals()
```

- Next, create a new view, “default/tester.html”:

```
1 <html>
2   <body>
3     {{numbers = [1, 2, 3]}}
4     <ul>
5       {{for n in numbers:}}<li>{{=n}}</li>{{pass}}
6     </ul>
7   </body>
8 </html>
```

- Test it out <http://localhost:8000/tester>.

2. Template Composition (inheritence): like most templating languages, the web2py flavor can extend and include a set of sub templates. For example, you could have the base or child template, *index.html*, that extends from a parent template, *default.html*. Meanwhile, *default.html* could include two sub templates, *header.html* and *footer.html*:

For example, add the parent

```
1 {{extend 'layout.html'}}
2 <html>
3   <body>
4     {{numbers = [1, 2, 3]}}
5     <ul>
6       {{for n in numbers:}}<li>{{=n}}</li>{{pass}}
7     </ul>
8   </body>
9 </html>
```

Test it out again [here](#).

3. Javascript/jQuery libraries: As you have seen, web2py includes a number of Javascript and jQuery libraries, many of which are pre-configured. Refer to the web2py [documentation](#) for more information on Javascript, jQuery, and other components of the views.

Let's build the templates for py2manager.

1. *default/index.html*:

```
1  {{extend 'layout.html'}}
2  <h2>Welcome to py2manager</h2>
3  <br/>
4      {{=(project_form)}}
5  <br/>
6  <h3> All Open Projects </h3>
7  <ul>{{for project in projects:}}
8      <li>
9          {{=(project.name)}}
10         </li>
11         {{pass}}
12 </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/index>. Notice how this template extends from *layout.html*.

2. *default/user.html*:

Open up this file. This template was created automatically to make the development process easier and quicker by providing user authentication in the box. If you go back to the *default.py* file, you can see a description of the main functionalities of the user function:

```
1  """
2  exposes:
3  http://.../[app]/default/user/login
4  http://.../[app]/default/user/logout
5  http://.../[app]/default/user/register
6  http://.../[app]/default/user/profile
7  http://.../[app]/default/user/retrieve_password
8  http://.../[app]/default/user/change_password
9  use @auth.requires_login()
10     @auth.requires_membership('group name')
11     @auth.requires_permission('read','table name',record_id)
12  to decorate functions that need access control
13  """
```

Read more about authentication [here](#).

3. Layout:

Let's edit the main layout to replace the generic template. Start with *models/menu.py*. Update the following code:

```
1 response.logo = A(B('py',SPAN(2),'manager'),_class="brand")
2 response.title = "py2manager"
3 response.subtitle = T('just another project manager')
```

Then update the application menu:

```
1 response.menu = [(T('Home'), False, URL('default', 'index'), []),
2 (T('Add Project'), False, URL('default', 'add'), []),
3 (T('Add Company'), False, URL('default', 'company'), []),
4 (T('Employees'), False, URL('default', 'employee'), [])]
5
6 DEVELOPMENT_MENU = False
```

Take a look at your changes:

```
1 #####
2 ## Customize your APP title, subtitle and menus here
3 #####
4
5 response.logo = A(B('py',SPAN(2),'manager'),_class="brand")
6 response.title = "py2manager"
7 response.subtitle = T('just another project manager')
8
9 #####
10 ## this is the main application menu add/remove items as required
11 #####
12
13 response.menu = [(T('Home'), False, URL('default', 'index'), []),
14 (T('Add Project'), False, URL('default', 'add'), []),
15 (T('Add Company'), False, URL('default', 'company'), []),
16 (T('Employees'), False, URL('default', 'employee'), [])]
17
18 DEVELOPMENT_MENU = False
```

Now that we've gone over the Model View Controller architecture, let's shift to focus on the main functionality of the application.

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* and place the following code after `auth = Auth(db)` and before `auth.define_tables()`:

```
1 auth.settings.extra_fields['auth_user']= [  
2     Field('address'),  
3     Field('city'),  
4     Field('zip'),  
5     Field('image', 'upload')]
```

Save the file. Navigate to <http://localhost:8000/index> and login if necessary. 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 them 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

Choose File

No file chosen

[file] ☐ delete

Image:



Save profile

Figure 16.3: profile pic

Add Projects

To clean up the homepage, let's move the form to add new projects to a separate page.

1. Open your *default.py* file, and add a new function:

```
1 @auth.requires_login()
2 def add():
3     project_form = SQLFORM(db.project).process()
4     return dict(project_form=project_form)
```

2. Then update the *index()* function:

```
1 @auth.requires_login()
2 def index():
3     projects = db(db.project).select()
4     users = db(db.auth_user).select()
5     companies = db(db.company).select()
6     return locals()
```

3. Add a new template in the default directory called *add.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new project:</h2>
3 <br/>
4 {{=project_form.custom.begin}}
5 <strong>Project name</strong><br/>{{=project_form.custom.widget.name}}<br/>
6 <strong>Company
   name</strong><br/>{{=project_form.custom.widget.company_name}}<br/>
7 <strong>Description</strong><br/>{{=project_form.custom.widget.description}}<br/>
8 <strong>Start
   Date</strong><br/>{{=project_form.custom.widget.start_date}}<br/>
9 <strong>Due Date</strong><br/>{{=project_form.custom.widget.due_date}}<br/>
10 {{=project_form.custom.submit}}
11 {{=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]`.

4. Remove the form from *index.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Welcome to py2manager</h2>
3 <br>
4 <h3> All Open Projects </h3>
```

```
5 <ul>{{for project in projects:}}  
6   <li>  
7     {{=(project.name)}}  
8   </li>  
9   {{pass}}  
10 </ul>
```

Add Companies

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.

1. *default.py*:

```
1 @auth.requires_login()
2 def company():
3     company_form = SQLFORM(db.company).process()
4     return dict(company_form=company_form)
```

2. Add a new template in the default directory called *company.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new company:</h2>
3 <br/>
4 {{=company_form.custom.begin}}
5 <strong>Company
6     Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>
7 <strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
8 <strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
9 <strong>URL</strong><br/>{{=company_form.custom.widget.url}}<br/>
10 {{=company_form.custom.submit}}
    {{=company_form.custom.end}}
```

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.

1. Update the `index()` function in `default.py`:

```
1 @auth.requires_login()
2 def index():
3     response.flash = T('Welcome!')
4     grid = SQLFORM.grid(db.project)
5     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:

```
1 {{extend 'layout.html'}}
2 <h2>All projects:</h2>
3 <br/>
4 {{=grid}}
```

3. Navigate to <http://127.0.0.1:8000/> 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`:

```
1 db.project.start_date.requires = IS_DATE(format=T('%m-%d-%Y'),
2     error_message='Must be MM-DD-YYYY!')
3
4 db.project.due_date.requires = IS_DATE(format=T('%m-%d-%Y'),
5     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:

```
1 var w2p_ajax_date_format = "{%=T('%Y-%m-%d')%}";  
2 var w2p_ajax_datetime_format = "{%=T('%Y-%m-%d %H:%M:%S')%}";
```

To:

```
1 var w2p_ajax_date_format = "{%=T('%m-%d-%Y')%}";  
2 var w2p_ajax_datetime_format = "{%=T('%m-%d-%Y %H:%M:%S')%}";
```

6. Now let's update the grid in the `index()` function within the controller:

```
1 grid = SQLFORM.grid(db.project, create=False,  
2     fields=[db.project.name, db.project.employee_name,  
3     db.project.company_name, db.project.start_date,  
4     db.project.due_date, db.project.completed],  
5     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 output for additional help.

More Grids

First, let's add a grid to the company view.

1. *default.py*:

```
1 @auth.requires_login()
2 def company():
3     company_form = SQLFORM(db.company).process()
4     grid = SQLFORM.grid(db.company, create=False, deletable=False,
5         editable=False,
6         maxtextlength=50, orderby=db.company.company_name)
7     return locals()
```

2. *company.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Add a new company:</h2>
3 <br/>
4 {{=company_form.custom.begin}}
5 <strong>Company
6     Name</strong><br/>{{=company_form.custom.widget.company_name}}<br/>
7 <strong>Email</strong><br/>{{=company_form.custom.widget.email}}<br/>
8 <strong>Phone</strong><br/>{{=company_form.custom.widget.phone}}<br/>
9 {{=company_form.custom.submit}}
10 {{=company_form.custom.end}}
11 <br/>
12 <br/>
13 <h2>All companies:</h2>
14 <br/>
15 {{=grid}}
```

Next, let's create the employee view:

1. *default.py*:

```
1 @auth.requires_login()
2 def employee():
3     employee_form = SQLFORM(db.auth_user).process()
4     grid = SQLFORM.grid(db.auth_user, create=False,
5         fields=[db.auth_user.first_name, db.auth_user.last_name,
6             db.auth_user.email], deletable=False, editable=False, maxtextlength=50)
7     return locals()
```

2. *employee.html*:


```
1 {{extend 'layout.html'}}
2 <h2>All employees:</h2>
3 <br/>
4 {{=grid}}
```

Test both of the new views in the browser.

Notes

Next, let's add the ability to add notes to each project.

1. Add a new table to the database in *db_task.py*:

```
1 db.define_table('note',
2     Field('post_id', 'reference project', writable=False),
3     Field('post', 'text', notnull=True),
4     Field('created_on', 'datetime', default=request.now, writable=False),
5     Field('created_by', db.auth_user, default=auth.user_id))
6
7 db.note.post_id.readable = db.note.post_id.writable = False
8 db.note.created_on.readable = db.note.created_on.writable = False
9 db.note.created_on.requires = IS_DATE(format=T('%m-%d-%Y'),
10     error_message='Must be MM-DD-YYYY!')
11 db.note.created_by.readable = db.note.created_by.writable = False
```

2. Update the *index()* function and add a *note()* function in the controller:

```
1 @auth.requires_login()
2 def index():
3     response.flash = T('Welcome!')
4     notes = [lambda project:
5         A('Notes', _href=URL("default", "note", args=[project.id]))]
6     grid = SQLFORM.grid(db.project, create=False, links=notes,
7         fields=[db.project.name, db.project.employee_name,
8             db.project.company_name, db.project.start_date, db.project.due_date,
9             db.project.completed], deletable=False, maxtextlength=50)
10     return locals()
11
12 @auth.requires_login()
13 def note():
14     project = db.project(request.args(0))
15     db.note.post_id.default = project.id
16     form = crud.create(db.note) if auth.user else "Login to Post to the
17     Project"
18     allnotes = db(db.note.post_id==project.id).select()
19     return locals()
```

3. Take a look. Add some notes. Now let's add a new view called *default/note.html*:

```
1 {{extend 'layout.html'}}
2 <h2>Project Notes</h2>
3 <br/>
4 <h4>Current Notes</h4>
```

```

5  {{for n in allnotes:}}
6      <ul>
7          <li>{{=db.auth_user[n.created_by].first_name}} on
            {{=n.created_on.strftime("%m/%d/%Y")}}
8          - {{=n.post}}</li>
9      </ul>
10  {{pass}}
11  <h4>Add a note</h4>
12  {{=form}}<br>

```

4. Finally, let's update the `index()` function to add a button for the Notes link:

```

1  @auth.requires_login()
2  def index():
3      response.flash = T('Welcome!')
4      notes = [lambda project: A('Notes', _class="btn",
5          _href=URL("default", "note", args=[project.id]))]
6      grid = SQLFORM.grid(db.project, create=False, links=notes,
7          fields=[db.project.name, db.project.employee_name,
8              db.project.company_name,
9              db.project.start_date, db.project.due_date, db.project.completed],
10             deletable=False, maxtextlength=50)
11  return locals()

```

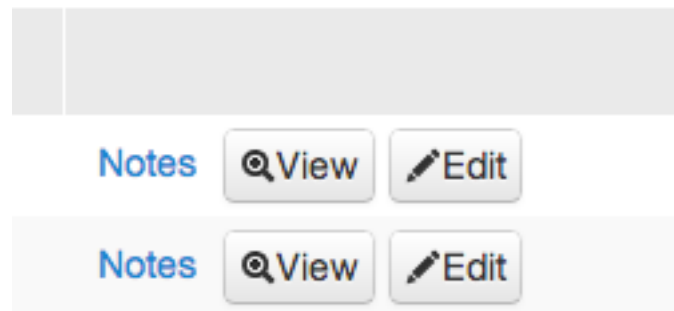


Figure 16.4: grid links

We just added the `btn` class to the `notes` variable.

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 following 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:

```
1 routes_onerror = [  
2     ('*/*', '/py2manager/static/error.html')  
3 ]
```

Then add the `error.html` file to the “static” directory:

```
1 <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.

Homework

- Please read the web2py [documentation](#) regarding error handling.

Final Word

What's this app missing? Would you like to see any additional features? If so, please post your feedback on the Real Python [forum](#). Thanks!

Chapter 17

Interlude: Web Scraping and Crawling

Since data is unfortunately not always accessible through a manageable format via web APIs (i.e., XML or JSON), 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 repeated requests in a short span of time 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: ¹

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.

3. 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.

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

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.

Figure 17.1: ebay

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
```

Figure 17.2: robots.txt

- 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.
- Disallow parameters establish the directories or files - “Disallow: /folder/” or “Disallow: /file.html” - that robots must avoid.
- 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.

WARNING: 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. ²

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

NOTE If you are using Windows there are additional steps and dependencies that you need to install. Please follow this [video](#) for details. Good luck!

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

HackerNews (BaseSpider)

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

Once Scrapy is installed, open your terminal and navigate to your “client-side” directory, and then start a new Scrapy project: `scrapy startproject hackernews` ³

This will create a “hackernews” directory with the following contents:

```
1  hackernews
2      __init__.py
3      items.py
4      pipelines.py
5      settings.py
6      spiders
7      __init__.py
8  scrapy.cfg
```

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 in your text editor and edit it to define the fields that you want extracted. Let's grab the title and url from each posting:

```
1  from scrapy.item import Item, Field
2
3  class HackernewsItem(Item):
4      title = Field()
5      url = Field()
```

Now, let's create the actual spider:

```
1  # spider.py
2
3
4  from scrapy.spider import BaseSpider
5  from scrapy.selector import HtmlXPathSelector
6  from hackernews.items import HackernewsItem
7
8  class MySpider(BaseSpider):
9
10     # name the spider
11     name = "hackernews"
12
13     # allowed domains to scrape
14     allowed_domains = ["news.ycombinator.com/"]
```

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

```

15
16 # urls the spider begins to crawl from
17 start_urls = ["https://news.ycombinator.com/"]
18
19 # parses and returns the scraped data
20 def parse(self, response):
21     hxs = HtmlXPathSelector(response)
22     titles = hxs.select('//td[@class="title"]')
23     items = []
24     for title in titles:
25         item = HackernewsItem()
26         item["title"] = title.select("a/text()").extract()
27         item["url"] = title.select("a/@href").extract()
28         items.append(item)
29     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 or Chrome: <https://news.ycombinator.com/>
2. Right click on the first article link and select “Inspect Element”
3. In the Firebug or Developer Tools console, you can see the HTML that’s used to display the first link:
4. You can see that everything we need, text and url, is located between the `<td class="title">` `</td>` tag:

```

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

```

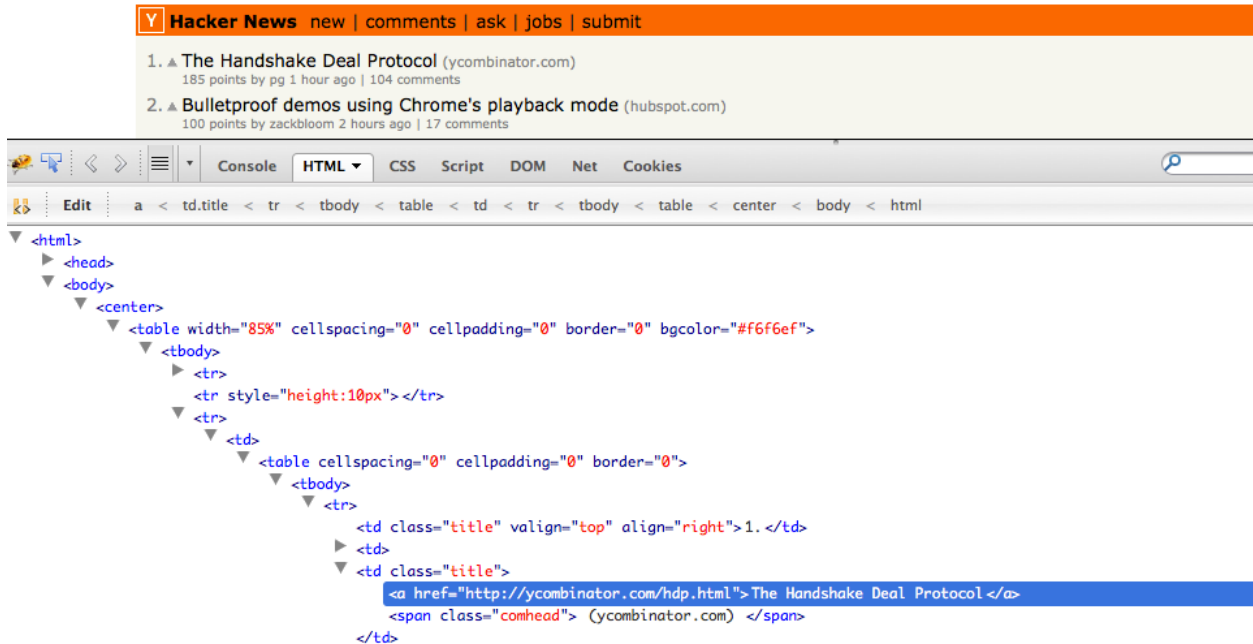


Figure 17.3: hackernews

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:

```
1 <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 the 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`. Assuming there are no errors in the URL, you can now test your XPath expressions.

1. Start by using Firebug or Developer Tools to get an idea of what to test. Based on the 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 `sel.xpath('//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:

```
1 sel.xpath('//td[@class="title"]/a').extract()[0]
```

NOTE: By adding `[0]` to the end, we are just returning the first result.

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

```
1 sel.xpath('//td[@class="title"]/a/text()').extract()[0]
```

and

```
1 sel.xpath('//td[@class="title"]/a/@href').extract()[0]
```

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:

```
1 sel.xpath('//span[@class="yclinks"]/a[3]/@href').extract()
```

- The comment links and URLs:

```
1 sel.xpath('//td[@class="subtext"]/a/@href').extract()[0]
```

and

```
1 sel.xpath('//td[@class="subtext"]/a/text()').extract()[0]
```

See what else you can extract. Play around with this!

NOTE 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:2014_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:

```
1 $ scrapy startproject wikipedia
```

2. Create the *items.py* file:

```
1 from scrapy.item import Item, Field
2
3 class WikipediaItem(Item):
4     title = Field()
5     url = Field()
```

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

```
1 $ cd wikipedia/wikipedia
2 $ scrapy genspider -d basic
```

The results are outputted to the screen:

```
1 class $classname(Spider):
2     name = "$name"
3     allowed_domains = ["$domain"]
4     start_urls = (
5         'http://www.$domain/',
6     )
7
8     def parse(self, response):
9         pass
```

4. Copy and paste the output into your text editor, and then finish coding the scraper:

```

1 # wikibase.py - basespider
2
3
4 from scrapy.spider import BaseSpider
5 from scrapy.selector import HtmlXPathSelector
6
7 from wikipedia.items import WikipediaItem
8
9 class MySpider(BaseSpider):
10     name = "wiki"
11     allowed_domains = ["en.wikipedia.org"]
12     start_urls = [
13         "http://en.wikipedia.org/wiki/Category:2014_films"
14     ]
15
16     def parse(self, response):
17         hxs = HtmlXPathSelector(response)
18         titles = hxs.select('//tr[@style="vertical-align: top;"]//li')
19         items = []
20         for title in titles:
21             item = WikipediaItem()
22             item["title"] = title.select("a/text()").extract()
23             item["url"] = title.select("a/@href").extract()
24             items.append(item)
25         return(items)

```

Save this to your “spiders” directory as *wiki.py*.

Did you notice the XPath?

```

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

```

This is equivalent to:

```

1 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, `//`.

This time, output the data to a JSON file:

```

1 $ 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:

```

1 for title in titles:
2     item = WikipediaItem()
3     url = title.select("a/@href").extract()
4     item["title"] = title.select("a/text()").extract()
5     item["url"] = urljoin("http://en.wikipedia.org", url[1:])
6     items.append(item)
7 return(items)

```

Your script should look like this:

```

1 # wikibase.py - basespider
2
3
4 from scrapy.spider import BaseSpider
5 from scrapy.selector import HtmlXPathSelector
6
7 from wikipedia.items import WikipediaItem
8 from urlparse import urljoin
9
10 class MySpider(BaseSpider):
11     name = "wiki"
12     allowed_domains = ["en.wikipedia.org"]
13     start_urls = [
14         "http://en.wikipedia.org/wiki/Category:2014_films"
15     ]
16
17     def parse(self, response):
18         hxs = HtmlXPathSelector(response)
19         titles = hxs.select('//tr[@style="vertical-align: top;"]//li')
20         items = []
21         for title in titles:
22             item = WikipediaItem()
23             url = title.select("a/@href").extract()
24             item["title"] = title.select("a/text()").extract()
25             item["url"] = urljoin("http://en.wikipedia.org", url[1:])
26             items.append(item)
27         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:

```
1 $ 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*:

```
1 from scrapy.item import Item, Field
2
3 class SocrataItem(Item):
4     text = Field()
5     url = Field()
6     views = Field()
```

2. *socrata_base.py*:

```
1 # socrata_base.py - basespider
2
3
4 from scrapy.spider import BaseSpider
5 from scrapy.selector import HtmlXPathSelector
6
7 from socrata.items import SocrataItem
8
9 class MySpider(BaseSpider):
10     name = "socrata"
11     allowed_domains = ["opendata.socrata.com"]
12     start_urls = [
13         "https://opendata.socrata.com"
14     ]
15
16     def parse(self, response):
17         hxs = HtmlXPathSelector(response)
18         titles = hxs.select('//tr[@itemscope="itemscope"]')
19         items = []
20         for t in titles:
21             item = SocrataItem()
22             item["text"] = t.select("td[2]/div/span/text()").extract()
23             item["url"] = t.select("td[2]/div/a/@href").extract()
24             item["views"] = t.select("td[3]/span/text()").extract()
```

```

25         items.append(item)
26     return(items)

```

3. Release the spider:

```

1 $ cd socrata/socrata
2 $ 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 it. 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 *socrata_base.py*. Save it as *socrata_crawl.py*.
3. Update the imports:

```

1 from scrapy.contrib.spiders import CrawlSpider, Rule
2 from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
3 from scrapy.selector import HtmlXPathSelector
4
5 from socrata.items import SocrataItem

```

4. Add the rules:

```

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

```

5. What else do you have to update? First, the class must inherit from CrawlSpider, not BaseSpider. Anything else?

Final code:

```

1 # socrata_crawl.py - crawls spider
2
3
4 from scrapy.contrib.spiders import CrawlSpider, Rule
5 from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor

```

```

6 from scrapy.selector import HtmlXPathSelector
7
8 from socrata.items import SocrataItem
9
10 class MySpider(CrawlSpider):
11     name = "socrata2"
12     allowed_domains = ["opendata.socrata.com"]
13     start_urls = [
14         "https://opendata.socrata.com"
15     ]
16
17     rules = (Rule
18         (SgmlLinkExtractor(allow=("browse\?utf8=%E2%9C%93&page=\d*", )),
19         callback="parse_items", follow= True),)
18
19     def parse_items(self, response):
20         hxs = HtmlXPathSelector(response)
21         titles = hxs.select('//tr[@itemscope="itemscope"]')
22         items = []
23         for t in titles:
24             item = SocrataItem()
25             item["text"] = t.select("td[2]/a/text()").extract()
26             item["url"] = t.select("td[2]/a/@href").extract()
27             item["views"] = t.select("td[3]/span/text()").extract()
28             items.append(item)
29         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:

```

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

```

NOTE: 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 (Part 1) 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:

```
1 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 remembering.

WARNING 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:

```
1 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 second “socrata” directory from your shell:

```
1 import sqlite3
2
3 conn = sqlite3.connect("project.db")
4 cursor = conn.cursor()
5 cursor.execute("""CREATE TABLE data
6                 (text TEXT, url TEXT, views TEXT)
7                 """)
```

2. Update the *pipelines.py* file:

```

1 import sqlite3
2
3 class SocrataPipeline(object):
4     def __init__(self):
5         self.conn = sqlite3.connect('project.db')
6         self.cur = self.conn.cursor()
7
8     def process_item(self, item, spider):
9         self.cur.execute("insert into data (text, url, views)
10            values(?,?,?)", (item['text'][0], item['url'][0], item['views'][0]))
11         self.conn.commit()
12         return item

```

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

```

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

```

4. Test this out with the BaseSpider first:

```

1 $ scrapy crawl socrata -o project.db

```

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

Ready? Fire away:

```

1 $ scrapy crawl socrata2 -o project.db

```

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. Do a little dance.

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.

For now, go ahead and move on with the scraper running.

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`

NOTE Want some more fun? We need web professional web scrapers. Practice more with Scrapy. Make sure to upload everything to Github. Email us the link to info@realpython.com. We pay well.

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 on a weekly basis, 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.

```
1 # Download stock quotes in CSV
2
3
4 import requests
5 import time
6
7 i = 0
8
9 while (i < 0):
10
11     base_url = 'http://download.finance.yahoo.com/d/quotes.csv'
12
13     # retrieve data from web server
14     data = requests.get(base_url,
15                         params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})
16
17     # write the data to csv
18     with open("stocks.csv", "a") as code:
19         code.write(data.content)
20     i+=1
21
22     # pause for 3 seconds
23     time.sleep(3)
```

Save this file as *clientp.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. Just leave it running in the background.

Let's look at how I got the parameters: `params={'s': 'GOOG', 'f': 'sl1d1t1c1ohgv', '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=sl1d1t1c1ohgv&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.

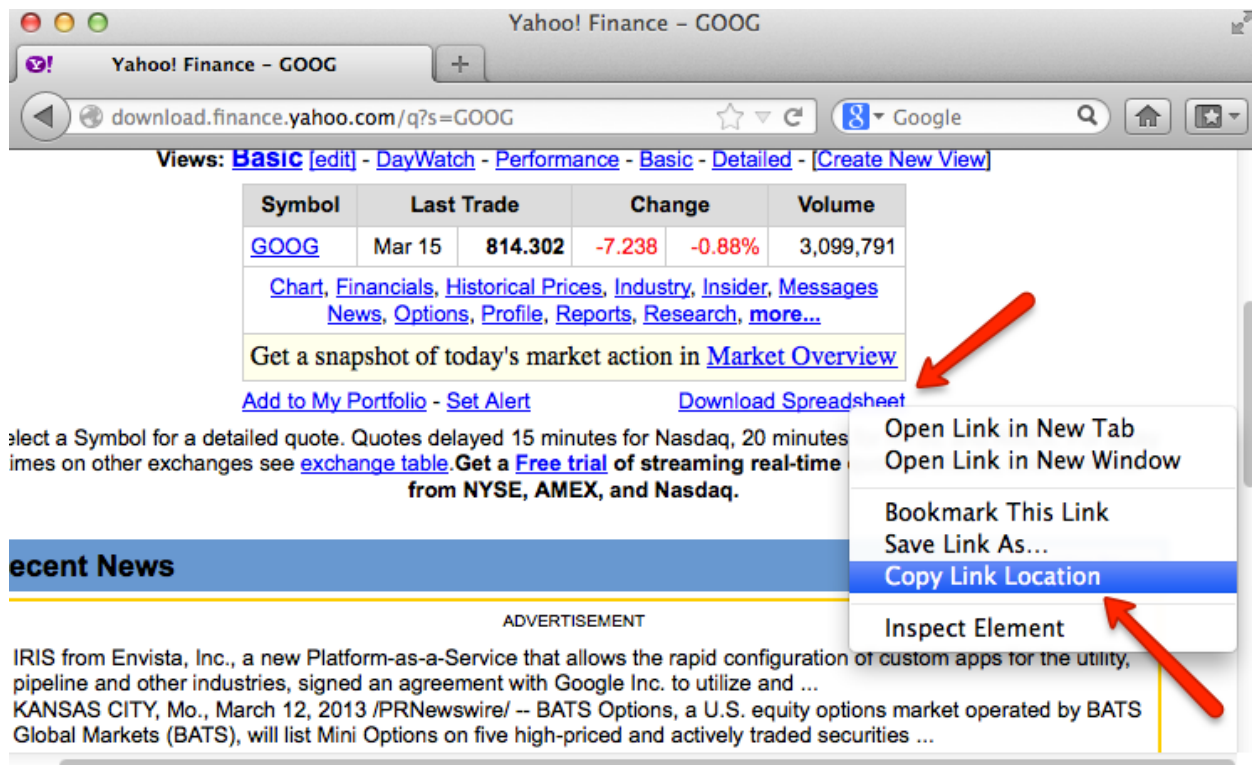


Figure 17.4: yahoo finance

How would you then pull prices for a number of quotes using a loop? Think about this before you look at the answer.

```

1 # Download stock quotes in CSV
2
3
4 import requests
5 import time
6
7 i = 0
8 stock_list = ['GOOG', 'YHOO', 'AOL']
9
10 while (i < 1):
11
12     base_url = 'http://download.finance.yahoo.com/d/quotes.csv'
13
14     # retrieve data from web server
15     for stock in stock_list:
16         data = requests.get(base_url,
17                             params={'s': stock, 'f': 'sl1d1t1c1ohgv', 'e': '.csv'})
18
19     # write the data to csv
20     with open("stocks.csv", "a") as code:

```

```
21         code.write(data.content)
22
23     i+=1
24
25     # pause for 3 seconds
26     time.sleep(3)
```


Chapter 18

web2py: REST Redux

Introduction

Remember the data we scraped from Socrata? No? 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 RESTful 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.

Basic REST

Create a new folder called “web2py-rest”. Then install and activate a virtualenv. Download the source files from web2py. Unzip the file. Add the contents directly to “web2py-rest”. Fire up the server. Then create a new application called “basic_rest”.

To set up a RESTful API follow these steps:

1. Create a new database table in in *db.py*:

```
1 db.define_table('fam',Field('role'),Field('name'))
```

2. Within the web2py admin, enter some dummy data into the newly created table.
3. Add the RESTful functions to the controller: ¹

```
1 @request.restful()
2 def api():
3     response.view = 'generic.'+request.extension
4     def GET(*args,**vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns,args,vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status,parser.error)
11     def POST(table_name,**vars):
12         return db[table_name].validate_and_insert(**vars)
13     def PUT(table_name,record_id,**vars):
14         return db(db[table_name]._id==record_id).update(**vars)
15     def DELETE(table_name,record_id):
16         return db(db[table_name]._id==record_id).delete()
17     return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)
```

These functions expose any field in our database to the outside world. If you want to limit the resources exposed, you’ll need to define various [patterns](#).

For example:

```
1 def GET(*args,**vars):
2     patterns = [
3         "/test[fam]",
4         "/test/{fam.name.startswith}",
5         "/test/{fam.name}/:field",
6     ]
7     parser = db.parse_as_rest(patterns,args,vars)
```

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

```

8 if parser.status == 200:
9     return dict(content=parser.response)
10 else:
11     raise HTTP(parser.status,parser.error)

```

So, these patterns define the following rules:

- `http://127.0.0.1:8000/basic_rest/default/api/test` -> GET all data
- `http://127.0.0.1:8000/basic_rest/default/api/test/t` -> GET a single data point where the “name” starts with “t”
- `http://127.0.0.1:8000/basic_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/basic_rest/default/api/fam.json`
`{“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/basic_rest/default/api/fam/id/1.json`

```

1 {“content”: [{“role”: “Father”, “id”: 1, “name”: “Tom”}]}

```

Explanation: GET data where “id” == 1

NOTE: These results will vary depending upon the sample data that you added.

5. Test out the following requests in the Shell and look at the results in the database:

```

1 >>> import requests
2 >>>
3 >>> payload = {“name” : “john”, “role” : “brother”}
4 >>> r =
      requests.post(“http://127.0.0.1:8000/basic_rest/default/api/fam.json”,
      data=payload)
5 >>> r
6 <Response [200]>
7 >>>
8 >>> r =
      requests.delete(“http://127.0.0.1:8000/basic_rest/default/api/fam/2.json”)
9 >>> r
10 <Response [200]>
11 >>>
12 >>> payload = {“name” : “Jeffrey”}

```

```

13 >>> r =
    requests.put("http://127.0.0.1:8000/basic_rest/default/api/fam/3.json",
    data=payload)
14 >>> r
15 <Response [200]>

```

6. Now in most cases, you do not want just anybody having access to your API like this. Besides, limiting the datapoints as described above, you also want to have user authentication in place.
7. Register a new user at http://127.0.0.1:8000/basic_rest/default/user/register, and then update the function in the controller, adding a login required decorator:

```

1 auth.settings.allow_basic_login = True
2
3 @auth.requires_login()
4 @request.restful()
5 def api():
6     response.view = 'generic.'+request.extension
7     def GET(*args,**vars):
8         patterns = 'auto'
9         parser = db.parse_as_rest(patterns,args,vars)
10        if parser.status == 200:
11            return dict(content=parser.response)
12        else:
13            raise HTTP(parser.status,parser.error)
14    def POST(table_name,**vars):
15        return db[table_name].validate_and_insert(**vars)
16    def PUT(table_name,record_id,**vars):
17        return db(db[table_name]._id==record_id).update(**vars)
18    def DELETE(table_name,record_id):
19        return db(db[table_name]._id==record_id).delete()
20    return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)

```

8. Now you need to be authenticated to make any requests:

```

1 >>> import requests
2 >>> from requests.auth import HTTPBasicAuth
3 >>> payload = {"name" : "Katie", "role" : "Cousin"}
4 >>> auth = HTTPBasicAuth("Michael", "reallyWRONG")
5 >>> r =
    requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json",
    data=payload, auth=auth)
6 >>> r
7 <Response [403]>
8 >>>
9 >>> auth = HTTPBasicAuth("michael@realpython.com", "admin")

```

```
10 >>> r =  
    requests.post("http://127.0.0.1:8000/basic_rest/default/api/fam.json",  
                  data=payload, auth=auth)  
11 >>> r  
12 <Response [200]>
```

9. Test this out some more.

Homework

- Please watch [this](#) short video on REST.

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.

Setup

1. Create a new app called “socrata” within “web2py-rest”
2. Register a new user: <http://127.0.0.1:8000/socrata/default/user/register>
3. Create a new table with the following schema:

```
1 db.define_table('socrata',Field('name'),Field('url'),Field('views'))
```

4. Now we need to extract the data from the *projects.db* data and import it 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*: “../web2py-rest/applications/socrata”
 - You need to rename the “text” field since it’s technically a restricted name. Change this to “name”. Then use the Find and Replace feature in Sublime to remove the word “views” from each data point. The word “views” makes it impossible to filter and sort the data. When we scraped the data, we should have only grabbed the view count, not the word “views”. Mistakes are how you learn, though. Make sure when you do your find, grab the blank space in front of the word as well - e.g, “ views”.
 - 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:

In the future, when you set up your Scrappy Items Pipeline, you can dump the data right to the web2py database. The process is the same as outlined. Again, make sure to only grab the view count, not the word “views”.

API Design

1. First, When designing your RESTful API, you should follow these best practices: ³

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

- 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*:

```

1 @request.restful()
2 def api():
3     response.view = 'generic.json'+request.extension
4     def GET(*args,**vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns,args,vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status,parser.error)
11     def POST(table_name,**vars):
12         return db[table_name].validate_and_insert(**vars)
13     def PUT(table_name,record_id,**vars):
14         return db(db[table_name]._id==record_id).update(**vars)
15     def DELETE(table_name,record_id):
16         return db(db[table_name]._id==record_id).delete()
17     return dict(GET=GET, POST=POST, PUT=PUT, DELETE=DELETE)

```

Test

GET

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"] }

Endpoints and Patterns:

- <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].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/id/[id]/[field_name].json)

Let's look at each one in detail with the Python Shell.

Import the requests library to start:

```
1 >>> import requests
```

1. [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].json)

```
1 >>> r =
    requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/1.json")
2 >>> r
3 <Response [200]>
4 >>> r.content
5 '{"content": [{"name": "Drug and Alcohol Treatments Florida", "views": "6",
    "url":
    "https://opendata.socrata.com/dataset/Drug-and-Alcohol-Treatments-Florida/uzmv-9jrm",
    "id": 100}]]\n'
```

2. [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)

```
1 >>> r =
    requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/100/name.json")
2 >>> r
3 <Response [200]>
4 >>> r.content
5 '{"content": [{"name": "Drug and Alcohol Treatments Florida"}]]\n'
6 >>> r =
    requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/100/views.json")
7 >>> r
8 <Response [200]>
9 >>> r.content
10 '{"content": [{"views": "6"}]]\n'
```

POST

<http://127.0.0.1:8000/socrata/default/api/socrata.json>

```
1 >>> payload = {'name': 'new database', 'url': 'http://new.com', 'views': '22'}
2 >>> r =
    requests.post("http://127.0.0.1:8000/socrata/default/api/socrata.json", payload)
3 >>> r
4 <Response [200]>
```


PUT

http://127.0.0.1:8000/socrata/default/api/socrata/[id].json

```
1 >>> payload = {'name': 'new database'}
2 >>> r = requests.put("http://127.0.0.1:8000/socrata/default/api/socrata/3.json",
3     payload)
4 >>> r
5 <Response [200]>
6 >>> r =
7     requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/3/name.json")
8 >>> r
9 <Response [200]>
10 >>> r.content
11 '{"content": [{"name": "new database"}]}\n'
```

DELETE

http://127.0.0.1:8000/socrata/default/api/socrata/[id].json

```
1 >>> r =
2     requests.delete("http://127.0.0.1:8000/socrata/default/api/socrata/3.json")
3 >>> r
4 <Response [200]>
5 >>> r =
6     requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/3/name.json")
7 >>> r
8 <Response [404]>
9 >>> r.content
10 'no record found'
```

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.

```
1 auth.settings.allow_basic_login = True
2 @auth.requires_login()
```

Authorized:

```
1 >>> r =
2     requests.get("http://127.0.0.1:8000/socrata/default/api/socrata/id/1.json",
3     auth=('michael@realpython.com', 'admin'))
4 >>> r.content
```

3

```
'{"content": [{"name": "2010 Report to Congress on White House Staff", "views":  
  "508,705", "url":  
  "https://opendata.socrata.com/Government/2010-Report-to-Congress-on-White-House-Staff/vedg-  
  "id": 1}]]\n'
```

Chapter 19

Django: Quickstart

Overview

Like web2py, Django is a high-level web framework, used for rapid web development. With a strong community of supporters and some of the largest, 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 back end. 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 one of 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.



Figure 19.1: django logo

Django projects are logically organized around the Model-View-Controller (MVC) architecture. However, Django's MVC flavor is slightly different in that the views act as the controllers. So, projects are

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

actually organized in a Model-Template-Views architecture (MTV):

- **Models** represent your data model/tables, 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 (like the controllers in the 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

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.

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

Installation

1. Create a “django” directory, then create and activate a virtualenv.
2. Now install Django 1.5.5:

```
1 $ pip install Django==1.5.5
```

If you need to check the Django version currently installed within your virtualenv, open the Python shell and run the following commands:

```
1 >>> import django
2 >>> django.VERSION
3 (1, 5, 5, 'final', 0)
```

If you see a different version make sure to uninstall Django, `pip uninstall Django` or if you see an `ImportError`, then try installing Django again. Double-check that your virtualenv is activated before installing.

Hello, World!

As always, let's start with a basic Hello World app. :)

Basic Setup

1. Navigate to the “django” directory. Activate your virtualenv. Start a new Django project:

```
1 $ django-admin.py startproject hello_world_project
```

2. This command created the basic project layout, containing one directory and five files:

```
1
2 hello_world_project
3     __init__.py
4     settings.py
5     urls.py
6     wsgi.py
7     manage.py
```

NOTE: 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 the *manage.py*, *settings.py*, and *urls.py* files:

- **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 project, let's make sure everything is setup correctly by running the built-in development server. Navigate into the first “hello_world_project” directory (Project root) and run the following command:

```
1 $ python manage.py runserver
```

You can specify a different port with the following command (if necessary):

```
1 $ python manage.py runserver 8080
```

You should see something similar to this:

```
1 Validating models...
2
3 0 errors found
4 March 06, 2014 - 17:31:43
5 Django version 1.5.5, using settings 'hello_world.settings'
6 Development server is running at http://127.0.0.1:8000/
7 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!

Figure 19.2: dev server

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. Within Sublime:

- Navigate to Project -> Add folder to Project
- Find the first "hello_world_project" directory then click Open
- Save the project for easy access by navigating to Project -> Save Project and save as *hello-world-project.sublime-project* within your "django" directory.

You should see both directories and all five files on the left pane.

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 Project root directory, and then run the following command:

```
1 $ python manage.py startapp hello_world
```

This will create a new directory called “hello_world”, which includes the following files:

- **models.py:** This file is used to define your data models (entities and relationships) and map them to database table(s).
- **tests.py:** This houses your test code used for testing your application (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 accept requests and return responses back to the user.

Your Project structure should now look like this:

```
1
2 hello_world
3     __init__.py
4     models.py
5     tests.py
6     views.py
7 hello_world_project
8     __init__.py
9     settings.py
10    urls.py
11    wsgi.py
12 manage.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 should have a *separate* function associated with it, distinct from other apps. Django apps are used to encapsulate common functions. Put another way, the project is your final product (your entire web app), comprised of separate functions from each app (each individual feature of your app, like - user authentication, a blog, registration, and so on. By breaking Projects into a series of small applications, you can theoretically reuse an application in a different Django project - so there's no need to reinvent the wheel.

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, *hello_wold*, to the end of the tuple. Your “INSTALLED_APPS” section should look like this-


```

1 INSTALLED_APPS = (
2     'django.contrib.auth',
3     'django.contrib.contenttypes',
4     'django.contrib.sessions',
5     'django.contrib.sites',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'hello_world',
9     # Uncomment the next line to enable the admin:
10    #'django.contrib.admin',
11    # Uncomment the next line to enable admin documentation:
12    # 'django.contrib.admindocs',
13 )

```

3. Open the *views.py* file and add the following code:

```

1 from django.http import HttpResponse
2
3 def index(request):
4     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. Add a *urls.py* file to the the “hello_world” app, then add the following code to link (or map) a URL to our specific home view:

```

1 from django.conf.urls import patterns, include, url
2 from hello_world import views
3
4 urlpatterns = patterns('',
5     url(r'^$', 'views.index', name='index'),
6 )

```

5. With our App view and URLs setup, we now just need to link the Project URLs to the App URLs. To do so, add the following code to the *Projecturls.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'^$', 'hello_world_project.views.home', name='home'),
10    # url(r'^hello_world_project/',
11    include('hello_world_project.foo.urls')),
12
13    # Uncomment the admin/doc line below to enable admin documentation:
14    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
15
16    # Uncomment the next line to enable the admin:
17    # url(r'^admin/', include(admin.site.urls)),
18    url(r'^hello_world/$', 'hello_world.urls'),
19 )

```

SEE ALSO: Please read the official Django documentation on [URLs](#) for further details and examples.

- Let's test it out. Fire up the server (`python manage.py runserver`), and then open your browser to http://localhost:8000/hello_world. It worked! You should see the “Hello, World!” text in the browser.
- 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 Project's `urls.py` file again and add the following line to the end of the `urlpatterns` tuple:

```

1 urlpatterns = patterns('',
2     url(r'^hello_world/', include('hello_world.urls')),
3     url(r'^$', include('hello_world.urls')),
4 )

```

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 with the App's `urls.py` file and assigning them to URLs.

For example:

```
view: python def about(request):    return HttpResponse("Here is the About
Page. Want to return home? <a href='/'>Back Home</a>")

url: url(r'^about', views.about, name='about'),
```

Templates

Django templates are similar to web2py's views, which are used for displaying HTML to the user. You can also embed Python code directly into the templates. Let's modify the example above to take advantage of this feature.

1. Navigate to the root and create a new directory called "templates". Your project structure should now look like this:

```
1
2  hello_world
3      __init__.py
4      models.py
5      tests.py
6      urls.py
7      views.py
8  hello_world_project
9      __init__.py
10     settings.py
11     urls.py
12     wsgi.py
13 manage.py
14 templates
```

2. Next, let's update the *settings.py* file to add the path to our "templates" directory to the `TEMPLATE_DIRS` tuple so that your Django project knows where to find the templates:

```
1 TEMPLATE_DIRS = (
2     os.path.join(PROJECT_ROOT, 'templates'),
3 )
```

Also, make sure to add the following imports to the top of the file as well as the following paths.

```
1 import os
2
3 SETTINGS_DIR = os.path.dirname(__file__)
4 PROJECT_PATH = os.path.join(SETTINGS_DIR, os.pardir)
5 PROJECT_ROOT = os.path.abspath(PROJECT_PATH)
```

The `PROJECT_ROOT` variable contains the absolute path to our Django Project's root. You can make certain of this by adding a `print` statement to the *settings.py* file:

```
1 print PROJECT_ROOT
```

Then run the file:

```
1 $ python settings.py
2 /Users/michaelherman/Documents/repos/realpython/book2-exercises/django/hello_world_project
```

3. Update the *views.py* file:

```
1 from django.template import Context, loader
2 from datetime import datetime
3 from django.http import HttpResponse
4
5 def index(request):
6     return HttpResponse('<html><body>Hello, World!</body></html>')
7
8 def about(request):
9     return HttpResponse("Here is the About Page. Want to return home? <a
10 href='/'>Back Home</a>")
11
12 def better_hello(request):
13     t = loader.get_template('betterhello.html')
14     c = Context({'current_time': datetime.now(),})
15     return HttpResponse(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, again, is similar to the web2py syntax we saw earlier). 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:

```
1 <html>
2 <head><title>A Better Hello!</title></head>
3 <body>
4     <p>Hello, World! This template was rendered on {{current_time}}.</p>
5 </body>
6 </html>
```

5. Finally, we need to add the view to the *urls.py* file:

```
1 url(r'^better/$', views.better, name='better'),
```

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 March 6, 2014, 6:09 p.m..”

Here we have a placeholder in our view, `{{current_time}}`, which is replaced with the current date and time from the views: `{'current_time': datetime.now(),}`. If you see an error, double check your code. One of the most common errors is that the `TEMPLATE_DIRS` path is incorrect in your `settings.py` file. Try adding that `print` statement to double check that path like I did earlier.

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 re-fresh <http://localhost:8000/better>, it should display the U.S. Pacific Time.

Change the time zone so that the time is correct based on your location.

Workflow

Before moving on, let's take a look at the basic workflow used for creating a Django Project and Application:

Creating a Project:

1. Run `python django-admin.py startproject <name_of_the_project>` to create a new Project. This will create the Project in a new directory.

Creating an application:

1. Run, `python manage.py startapp <name_of_the_app>`.
2. Add the name of the app to the `INSTALLED_APPS` tuple within `settings.py` file so that Django knows that the new app exists.
3. Link the application urls to the main urls within the Project's `urls.py` file.
4. Add the views to the application's `views.py` file.
5. Add a `urls.py` file within the new application's directory to map the views to specific URLs.
6. Create a “templates” directory, update the paths in the “settings.py” file, and finally add any templates.

Homework

- The above workflow is used in a number of Django tutorials and projects. Keep in mind, that there are a number of workflows you could follow/employ resulting in a number of project structures.

- Within the exercises folder on Github, find the “hello_world” project. This project was built using a different workflow resulting in a slightly different structure. See if you can locate the differences. Try to build a basic project using the structure employed in this app.

Chapter 20

Interlude: Introduction to Javascript and jQuery

Before moving to the next Django project, let's finish the frontend tutorial by looking at part two: **JavaScript and jQuery**.

Along with HTML and CSS (which, again, provide structure and beauty, respectively) Javascript and jQuery are used to make webpages interactive.

Start by adding a *main.js* file and include the follow code:

```
1 $(function() {  
2     console.log("whee!")  
3 });
```

Then add the following files to your *index.html* just before the closing `</body>` tag:

```
1 <script src="http://code.jquery.com/jquery-1.10.2.min.js"></script>  
2 <script  
3     src="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></script>  
4 <script src="main.js"></script>
```

Here we are just including jQuery, JavaScript, and our custom JavaScript file, *main.js*.

Open the “index.html” file in your web browser. In the JavaScript file there is a `console.log`. This is a debugging tool that allows you to post a message to the browser’s JavaScript console - e.g. Firebug (Firefox) or Developer Tools (Chrome / Safari). If you’ve never used this before, Google “accessing the js console in Chrome” to learn how to pull up the JavaScript console.

Open your console. You should see the text “whee!”.

Now, insert a word into the input box and click Submit. Nothing happens. We need to somehow grab that inputted word and do *something* with it.

Handling the Event

The process of grabbing the inputted word from the form when a user hits Submit is commonly referred to as an event handler. In this case, the event is the actual button click. We will use jQuery to “handle” that event.

NOTE Please note that jQuery is JavaScript, just a set of libraries developed in JavaScript. It is possible to perform all the same functionality jQuery provides in vanilla JavaScript; it just takes a lot more code.

Update *main.js*:

```
1 $(function() {  
2  
3     console.log("whee!")  
4  
5     // event handler  
6     $("#btn-click").click(function() {  
7         if ($('#input').val() !== '') {  
8             var input = $("#input").val()  
9             console.log(input)  
10        }  
11    });  
12  
13 });
```

Add an id (id="btn-click") to “index.html” within the <button> tags:

Before:

```
1 <button class="btn btn-primary btn-md">Submit!</button>
```

After:

```
1 <button id="btn-click" class="btn btn-primary btn-md">Submit!</button>
```

What’s going on?

1. \$("#btn-click").click(function() { is the event. This initiates the process, running the code in the remainder of the function. In other words, the remaining JavaScript/jQuery will not run until there is a button click.
2. var input = \$("#input").val() sets a variable called “input”, while .val() fetches the value from the form input.
3. id="btn-click" is used to tie the HTML to the JavaScript. This id is referenced in the initial event within the JavaScript file - "#btn-click".

4. `console.log(input)` displays the word to the end user via the JavaScript console.

NOTE The `$()` in `$("#btn-click").click()` is a jQuery constructor. Basically, it's used to tell the browser that the code within the parenthesis is jQuery.

Open “index.html” in your browser. Make sure you have your JavaScript console open. Enter a word in the input box and click the button. This should display the word in the console:

Append the text to the DOM

Next, instead of using a `console.log` to display the inputted word to the user, let's add it to the Document Object Model (DOM). Wait. What's the DOM? Quite simply, the DOM is a structural representation of the HTML document. Using JavaScript, you can add, remove, or modify the contents of the DOM, which changes how the page looks to the end user.

main.js

Open up your JavaScript file and add this line of code:

```
1 $('ol').append('<li><a href="">x</a>' + input + '</li>');
```

Updated file:

```
1 $(function() {  
2  
3     console.log("whee!")  
4  
5     // event handler  
6     $("#btn-click").click(function() {  
7         if ($('#input').val() !== '') {  
8             // grab the value from the input box after the button click  
9             var input = $("input").val()  
10            // display value within the browser's JS console  
11            console.log(input)  
12            // add the value to the DOM  
13            $('ol').append('<li><a href="">x</a> - ' + input + '</li>');  
14        }  
15        $('#input').val('');  
16    });  
17  
18 });
```

What's going on?

`$('#ol').append('x' + input + '');` adds the value of the input variable to the DOM between the `<ol class="results">` and ``. Further, we're adding the value from the input, `input`, variable plus some HTML, `x - `. Also, the `$('#input').val('');` clears the input box.

Test this out in your browser.

Before moving on, we need to make one last update to “main.js” to remove todos from the DOM once complete.

Remove text from the DOM

Add the following code to *main.js*

```
1 $(document).on('click', 'a', function (e) {  
2     e.preventDefault();  
3     $(this).parent().remove();  
4 });
```

The final code:

```
1 $(function() {  
2  
3     console.log("whee!")  
4  
5     // event handler  
6     $("#btn-click").click(function() {  
7         if ($('#input').val() !== '') {  
8             // grab the value from the input box after the button click  
9             var input = $("#input").val()  
10            // display value within the browser's JS console  
11            console.log(input)  
12            // add the value to the DOM  
13            $('ol').append('<li><a href="">x</a> - ' + input + '</li>');  
14        }  
15        $('#input').val('');  
16    });  
17  
18 });  
19  
20 $(document).on('click', 'a', function (e) {  
21     e.preventDefault();  
22     $(this).parent().remove();  
23 });
```

Here, on the event, the click of the link, we're removing that specific todo from the DOM. `e.preventDefault()` cancels the default action of the click, which is to follow the link. Try removing this to see what happens. `this` refers to the current object, `a`, and we're removing the parent element, ``.

Test this out.

Homework

- See the Codecademy tracks on [JavaScript](#) and [jQuery](#) for more practice.

Chapter 21

Bloggy: A blog app

In this next project, we'll develop a simple blog using a similar workflow that we used in the Hello world app.

Model

1. Navigate to your “django” directory, activate your virtualenv, then create a new project called **bloggy**:

```
1 $ 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:

```
1 $ python manage.py runserver
```

Open <http://localhost:8000/> and you should see the familiar 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` as a value to 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 synchronize the database in the next step.

```
1 'ENGINE': 'django.db.backends.sqlite3',  
2 'NAME': 'bloggy.db',
```

5. Next, create and sync the database:

```
1 $ 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:

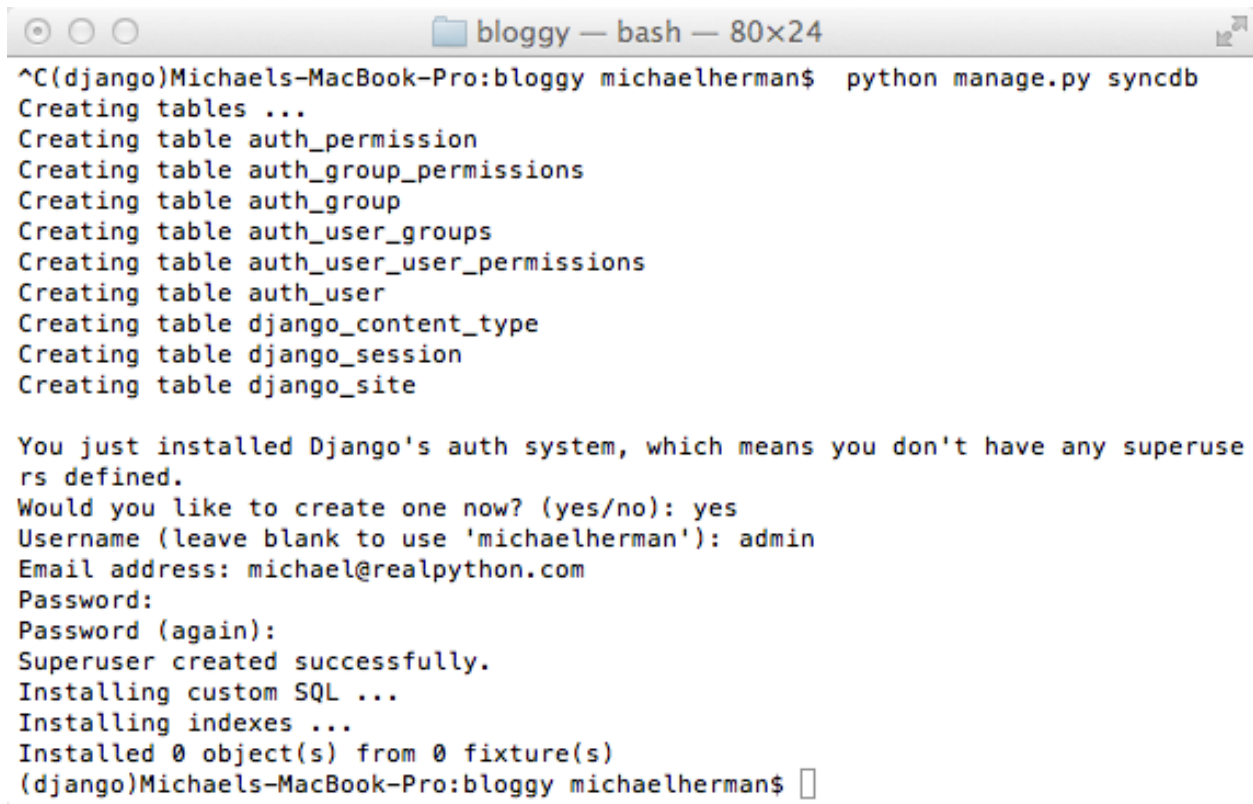
You should now see a the *bloggy.db* file in your project root directory.

6. Start a new app:

```
1 $ python manage.py startapp blog
```

7. Define the model for your application using the Django ORM (Object Relation Mapping) functions rather than with vanilla SQL. To do so, add the following code to *models.py*:

```
1 from django.db import models  
2  
3 class Post(models.Model):  
4     created_at = models.DateTimeField(auto_now_add=True)  
5     title = models.CharField(max_length=100)  
6     content = models.TextField()
```



```
^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$
```

Figure 21.1: syncdb

This class, `Post()` which inherits some of the basic properties from the standard Django Model class, defines the database table as well as each field - `created_at`, `title`, and `content`, representing a single blog post.

NOTE: 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 objects¹.

Note that the primary key - which is a unique id - (which we don't even need to define) and the `created_at` timestamp will both be automatically generated for us when `Post` objects are added to the database. We will just need to explicitly add the `title` and `content` when creating new objects (database rows).

8. Add the new app to the `settings.py` file:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'blog'
```

9. Before we sync the database (to convert our Django models into SQL tables), we can view the actual SQL statements that will be used by running the following command:

```
1 $ python manage.py sql blog
```

Your output should look like this:

```
1 BEGIN;  
2 CREATE TABLE "blog_post" (  
3     "id" integer NOT NULL PRIMARY KEY,  
4     "created_at" datetime NOT NULL,  
5     "title" varchar(100) NOT NULL,  
6     "content" text NOT NULL  
7 )  
8 ;  
9  
10 COMMIT;
```

NOTE: Compare the field types here to the field types used in our `models.py` file. You can view all the available field types on the Django [website](https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/). Also, did you notice the primary key?

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

10. We can also check to see if there are any errors in the data model:

```
1 $ python manage.py validate
```

11. And finally, execute the underlying SQL statements to create the database tables by synchronizing the database:

```
1 $ 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:

```
1 $ python manage.py shell
```

2. Let's add some data to the table we just created via the database API.² Start by importing the Post model we created:

```
1 >>> from blog.models import Post
```

3. If you search for objects in the table (somewhat equivalent to the sql statement `select * from blog_post`) you should find that it's empty:

```
1 >>> Post.objects.all()
2 []
```

Add some data!

4. To add new rows, we can use the following commands:

```
1 >>> 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")
2 >>> p.save()
3 >>> p = Post(title="Charting Best Practices", 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
'companys success. Here is a look at five charting best practices for
optimal data visualization and analysis.")
4 >>> p.save()
5 >>> p = Post(title="Understand Your Support System Better With Sentiment
Analysis", content="'Theres 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
```

²<https://docs.djangoproject.com/en/1.5/topics/db/queries/>

```

6 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()

```

NOTE: Remember that we don't need to add a primary key, `id`, or the `created_at` time stamp as those are auto-generated.

5. Now if you search for all objects, three objects should be returned:

```

1 >>> Post.objects.all()
2 [<Post: Post object>, <Post: Post object>, <Post: Post object>]

```

NOTE: When we use the `all()` or `filter()` functions, a `QuerySet` (list) is returned, which is iterable.

Notice how `<Post: Post object>` returns absolutely no distinguishing information about the object. Let's change that.

6. Open up your `models.py` file and add the following code:

```

1 def __unicode__(self):
2     return self.title

```

Your file should now look like this:

```

1 from django.db import models
2
3 class Post(models.Model):
4     created_at = models.DateTimeField(auto_now_add=True)
5     title = models.CharField(max_length=100)
6     content = models.TextField()
7
8     def __unicode__(self):
9         return self.title

```

This is a bit confusing, as Python Classes are usually returned with `__str__`, not `__unicode__`. We're using `unicode` because Django models use `unicode` by default³.

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:

```

1 [<Post: What Am I Good At>, <Post: Charting Best Practices>, <Post:
  Understand Your Support System Better With Sentiment Analysis>]

```

³<https://docs.djangoproject.com/en/1.5/ref/models/instances/#django.db.models.Model.unicode>

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 *models.py* file:

```
1 def __unicode__(self):
2     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 when you're done so it's just returning the title again:

```
1 def __unicode__(self):
2     return self.title
```

Open up your SQLite Browser to make sure the data was added correctly:

7. Let's look at how to query the data from the Shell.

- Import the Model:

```
1 >>> from blog.models import Post
```

- Return objects by a specific id:

```
1 >>> Post.objects.filter(id=1)
```

- Return objects by ids > 1:

```
1 >> Post.objects.filter(id__gt=1)
```

- Return objects where the title contains a specific word:

```
1 >>> 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 Django [documentation](#). And if you want a challenge, add more data from within the Shell via SQL and practice querying using straight SQL. Then delete all the data. Finally, see if you can add the same data and perform the same queries again within the Shell - but with objects via the Django ORM rather than SQL.

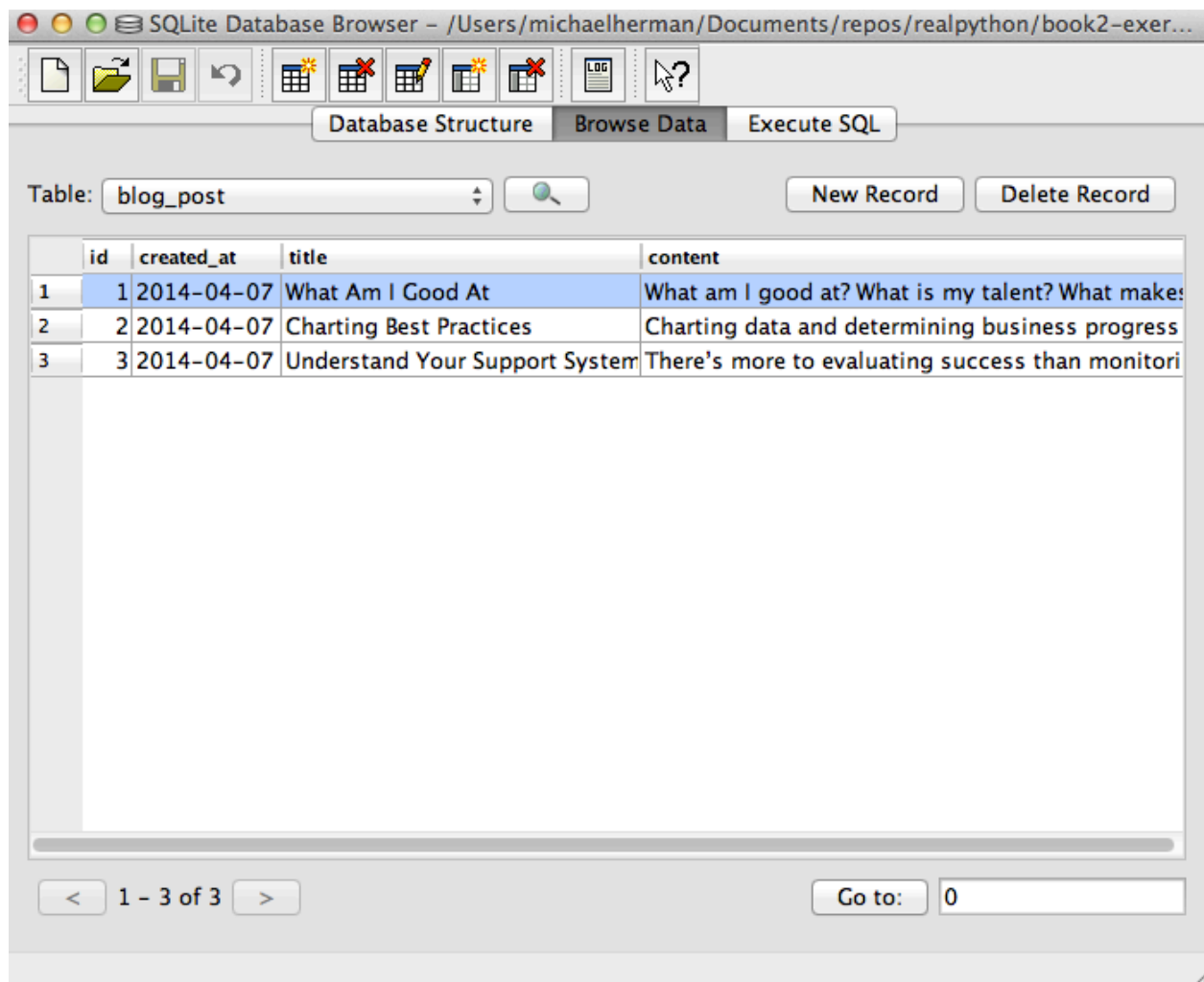


Figure 21.2: sqlite browser django

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 views. Make sure to include at least one test for each function within your *models.py* files.

1. Add the following code to *tests.py*:

```
1 from django.test import TestCase
2 from models import Post
3
4 class PostTests(TestCase):
5     def test_str(self):
6         my_title = Post(title='This is a basic title for a basic test case')
7         self.assertEqual(str(my_title), 'This is a basic title for a basic
            test case',)
```

2. Run the test case:

```
1 $ python manage.py test blog
```

NOTE: 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: `sh 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.

NOTE: 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 in general, it's probably much easier to access and modify the data stored within the models using the Django web-based admin. This is one of the most powerful built in feature that Django has to offer.

1. To access the admin, we first need to enable access to it then and setup the correct URL.
 - Open the *settings.py* file and uncomment the `'django.contrib.admin'` line within the `INSTALLED_APPS` tuple:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'blog',  
9     'django.contrib.admin'  
10 )
```

- Then sync the database again:

```
1 $ python manage.py syncdb
```

- Open your Project's *urls.py* file and uncomment these three lines-

```
1 from django.contrib import admin  
2 admin.autodiscover()  
3 url(r'^admin/', include(admin.site.urls)),
```

- Within your “blog” directory add a new file called *admin.py* that contains the following code:

```
1 from blog.models import Post  
2 from django.contrib import admin  
3  
4 admin.site.register(Post)
```

Here, we're simply telling Django which models we want available to the admin.

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:
3. Add some more posts, change some posts, delete some posts. Go crazy.

Django administration

Site administration

Auth	
Groups	+ Add Change
Users	+ Add Change
Blog	
Posts	+ Add Change
Sites	
Sites	+ Add Change



Figure 21.3: django admin

Custom Admin View

We can customize the admin views by simply editing our *admin.py* files. For example, let's say that when you view the posts in the admin - <http://localhost:8000/admin/blog/post/>, you also want to not only see the title, but the time the post was created at (from our `created_at` field in the model).

Django makes this easy.

1. Start by creating a `PostAdmin()` class that inherits from `admin.ModelAdmin` from the *admin.py* within the “blog” directory:

```
1 class PostAdmin(admin.ModelAdmin):  
2     pass
```

2. Next, in that new class, add a variable called `list_display` and set it equal to a tuple that includes the fields from the database we want displayed:

```
1 class PostAdmin(admin.ModelAdmin):  
2     list_display = ('title', 'created_at')
```

3. Finally, register this class with with Django's admin interface:

```
1 admin.site.register(Post, PostAdmin)
```

View the changes at <http://localhost:8000/admin/blog/post/>.

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 to the root called “templates” then within that directory add a “blog” directory, and then add the path the correct paths to our *settings.py* file, just like we did in the Hello World app:

```
1 import os
2
3 SETTINGS_DIR = os.path.dirname(__file__)
4 PROJECT_PATH = os.path.join(SETTINGS_DIR, os.pardir)
5 PROJECT_ROOT = os.path.abspath(PROJECT_PATH)
6
7 ...
8
9 TEMPLATE_DIRS = (
10     os.path.join(PROJECT_ROOT, 'templates'),
11 )
```

index.html

1. **view:** Add the following code to the *views.py* file:

```
1 from django.http import HttpResponse
2 from blog.models import Post
3 from django.template import Context, loader
4
5 def index(request):
6     latest_posts = Post.objects.all().order_by('-created_at')
7     t = loader.get_template('blog/index.html')
8     c = Context({'latest_posts': latest_posts,})
9     return HttpResponse(t.render(c))
```

2. **template:** Create a new file called *index.html* within the “templates/blog” directory and add the following code:

```
1 <h1>Bloggy: a blog app</h1>
2 {% if latest_posts %}
3 <ul>
4 {% for post in latest_posts %}
5     <li>{{post.created_at}} | {{post.title}} | {{post.content}}</li>
6 {% endfor %}
7 </ul>
```

```
8 {% endif %}
```

3. **url:** Add a *urls.py* file to your “blog” directory, then add the following code:

```
1 from django.conf.urls import patterns, include, url
2 from blog import views
3
4 urlpatterns = patterns('',
5     url(r'^$', views.index, name='index'),
6 )
```

4. Now update our Project’s *urls.py* file:

```
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     # Uncomment the next line to enable the admin:
9     url(r'^admin/', include(admin.site.urls)),
10    url(r'^blog/', include('blog.urls')),
11 )
```

5. 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

- April 6, 2014, 7:32 p.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.
- April 6, 2014, 7:31 p.m. | Charting Best Practices | 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.
- April 6, 2014, 7:31 p.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

Figure 21.4: bloggy basic view

Let’s clean this up a bit.

1. **template (second iteration):** Update *index.html*

```

1 <html>
2 <head>
3   <title>Bloggy: a blog app</title>
4 </head>
5 <body>
6 <h2>Welcome to Bloggy!</h2>
7 {% if latest_posts %}
8 <ul>
9   {% for post in latest_posts %}
10    <h3><a href="/blog/{{ post.id }}">{{ post.title }}</a></h3>
11    <p><em>{{ post.created_at }}</em></p>
12    <p>{{ post.content }}</p>
13    <br>
14  {% endfor %}
15 </ul>
16 {% endif %}
17 </body>
18 </html>

```

2. 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.
3. Before moving on, notice how we used two kinds of curly braces within the template. The first, `{% ... %}`, is used for adding Python logic/expressions such as a `if` statements or `for` loops, and the second, `{{ ... }}`, is used for inserting variables or the results of an expression.

post.html

1. **url:** Update the `urls.py` file within the “blog” directory:

```

1 from django.conf.urls import patterns, include, url
2 from blog import views
3
4 urlpatterns = patterns('',
5     url(r'^$', views.index, name='index'),
6     url(r'^(?P<post_id>\d+)/$', views.post, name='post'),
7 )

```

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. There's also a chapter on regular expressions within the first Real Python course. You can test out regular expressions using the [Python Regular Expression Tool](#):

2. **view:** Add a new function to `views.py`:

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()
{'post_id': u'1'}

# Run findall
>>> regex.findall(string)
[u'1']
```

Figure 21.5: regex test

```

1 from django.shortcuts import get_object_or_404
2
3 def post(request, post_id):
4     single_post = get_object_or_404(Post, pk=post_id)
5     t = loader.get_template('blog/post.html')
6     c = Context({'single_post': single_post,})
7     return HttpResponse(t.render(c))

```

This function accepts two arguments: 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.

3. **template:** Create a new template called *post.html*:

```

1 <html>
2 <head>
3   <title>Bloggy: {{ single_post.title }}</title>
4 </head>
5 <body>
6   <h2>{{ single_post.title }}</h2>
7   <ul>
8     <p><em>{{ single_post.created_at }}</em></p>
9     <p>{{ single_post.content }}</p>
10    <br/>
11  </ul>
12  <p>Had enough? Return <a href="/blog">home</a>.<br/>
13 </body>
14 </html>

```

4. Back on the development server, test out the links for each post. They should all be working now.

Friendly Views

Take a look at our current URLs for displaying posts - /blog/1, /blog/2, and so forth. Although this works, it's not the most human readable. Instead, let's update this so that the post title is used in the URL rather than the primary key.

To achieve this, we need to update our *views.py* and *urls.py* files as well as the *index.html* template.

1. **view:** Update the `index()` function within the *views.py* file within the “blog” directory:

```
1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     t = loader.get_template('blog/index.html')
4     context_dict = {'latest_posts': latest_posts,}
5     for post in latest_posts:
6         post.url = post.title.replace(' ', '_')
7     c = Context(context_dict)
8     return HttpResponse(t.render(c))
```

The main difference is that we're creating a `post.url` using a `for` loop to replace the spaces in a post name with underscores:

```
1 for post in latest_posts:
2     post.url = post.title.replace(' ', '_')
```

Thus, a post title of “test post” will convert to “test_post”. This will make our URLs look even better. If we didn't remove the spaces, they would show up as “test%20post”, which is difficult to read. Try this out if you're curious.

2. **template:** Now update the actual URL in the template.

Replace:

```
1 <h3><a href="/blog/{% post.id %}">{% post.title %}</a></h3>
```

With:

```
1 <h3><a href="/blog/{% post.url %}">{% post.title %}</a></h3>
```

3. **view:** The `post()` function is still accepting an id. Let's update that:

```
1 def post(request, post_name):
2     single_post = get_object_or_404(Post, title=post_name.replace('_', ' '))
3     t = loader.get_template('blog/post.html')
4     c = Context({'single_post': single_post,})
5     return HttpResponse(t.render(c))
```

Now, this function takes a new argument `post_name`, which we use to grab the post from the database. Notice how we have to replace the underscores with spaces so that it matches *exactly* what's in the database.

4. **url:** Finally, let's update the regex in the `urls.py` file:

```
1 url(r'^(?P<post_name>\w+)/$', views.post, name='post'),
```

Here we changed the [regular expression](#) to match a sequence of alphanumeric characters before the trailing forward slash. Test this out in the [Python Regular Expression Tool](#).

Run the server. Now you should have some URLs that are a little easier on the eye.

Homework

- Please read about the [Django Models](#) for more information on the Django ORM syntax.

South

Before moving on, let's look at how to handle database migrations (database schema changes) in Django with [South](#).

Why use South? Because any time you need to update or make changes to your database models, you must first *delete* the database, make the *changes*, and then *resync* your database. This will destroy all data you previously had in your database. You can bypass this process by using South to manage database migrations. Let's look at a simple workflow.

1. Open up your *models.py* file and add three new fields to the table:

```
1 tag = models.CharField(max_length=20, blank=True)
2 image = models.ImageField(upload_to="images", blank=True, null=True)
3 views = models.IntegerField(default=0)
```

NOTE: By setting `blank=True`, we are indicating that the field is not required and can be left blank when adding a new row to the table.

2. Since we're going to be working with images, install [Pillow](#):

```
1 $ pip install Pillow
```

3. Sync the database in your terminal:

```
1 $ python manage.py syncdb
```

4. Fire up your development server and login to the admin page, and then try to add a new row to the Post model. You should see the error `table blog_post has no column named tag` because the fields we tried to add didn't get added to the database. That's a problem. Fortunately, there is an easy solution: South. ⁴

5. Install South via pip:

```
1 $ pip install south
```

6. Then make sure to add it to your `INSTALLED_APPS` within your *settings.py* file. Remove the `tag`, `image`, and `view` fields from *models.py*, then sync the database and run the following code to initiate the migration:

```
1 $ python manage.py convert_to_south blog
```

7. Once again add the new fields to the model and run these commands to update the schema:

⁴<http://south.aeracode.org/>


```

1 $ python manage.py schemamigration blog --auto
2 $ python manage.py migrate blog

```

8. Run the server, go the Django admin, and you now should be able to add new rows that include a tag, an image, and/or the number of views . Boom.

Add a few posts with the new fields. Then update the remaining posts with tags and views. Do you remember how to update the which fields are displayed in the admin? Add the views to the admin.

9. Now let's update the the application so that tags and images will be viewed on the post page.

- Update *post.html*:

```

1 <html>
2 <head>
3   <title>Bloggy: {{ single_post.title }}</title>
4 </head>
5 <body>
6 <h2>{{ single_post.title }}</h2>
7 <ul>
8   <p><em>{{ single_post.created_at }}</em></p>
9   <p>{{ single_post.content }}</p>
10  <p>Tag: {{ single_post.tag }}</p>
11  <br>
12  <p>
13    {% if single_post.image %}
14    
15    {% endif %}
16  </p>
17 </ul>
18 <p>Had enough? Return <a href="/blog">home</a>.<br/>
19 </body>
20 </html>

```

- Update *settings.py*:

```

1 MEDIA_ROOT = os.path.join(PROJECT_ROOT, 'media')
2 MEDIA_URL = '/media/'
3 STATIC_ROOT = os.path.join(PROJECT_ROOT, 'static')
4 STATIC_URL = '/static/'

```

Make sure to create these directories as well.

- Update the Project-level *urls.py* file:

```

1 from settings import MEDIA_ROOT
2

```

```
3 urlpatterns = patterns('',
4     # Uncomment the next line to enable the admin:
5     url(r'^admin/', include(admin.site.urls)),
6     url(r'^blog/', include('blog.urls')),
7     url(r'^media/(?P<path>.*)$', 'django.views.static.serve',
8         {'document_root': MEDIA_ROOT}),
9 )
```

- Add some more posts from the admin. Make sure to include images. Check out the results: <http://localhost:8000/blog/>.

Styles

Before adding any styles, we need to break our templates into base and child templates, so that the child templates inherit the HTML and styles from the base 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:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Bloggy: a blog app</title>
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <meta name="description" content="">
8     <meta name="author" content="">
9
10    <!-- Le styles -->
11    <link
12      href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"
13      rel="stylesheet">
14    <style>
15      body {
16        padding-top: 60px; /* 60px to make the container go all the way to
17          the bottom of the topbar */
18      }
19    </style>
20    <script src="http://code.jquery.com/jquery-1.11.0.min.js"
21      type="text/javascript"></script>
22    {% block extrahead %}
23    {% endblock %}
24    <script type="text/javascript">
25      $(function(){
26        {% block jquery %}
27        {% endblock %}
28      });
29    </script>
30  </head>
31
32  <body>
33
34    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
35      <div class="container">
36        <div class="navbar-header">
37          <button type="button" class="navbar-toggle"
38            data-toggle="collapse" data-target=".navbar-collapse">
39            <span class="sr-only">Toggle navigation</span>
```

```

35         <span class="icon-bar"></span>
36         <span class="icon-bar"></span>
37         <span class="icon-bar"></span>
38     </button>
39     <a class="navbar-brand" href="/blog">Bloggy</a>
40 </div>
41 <div class="collapse navbar-collapse">
42     <ul class="nav navbar-nav">
43         <li><a href="#">Add Post</a></li>
44     </ul>
45 </div><!--/.nav-collapse -->
46 </div>
47 </div>
48
49 <div id="messages">
50     {% if messages %}
51         {% for message in messages %}
52             <div class="alert alert-{{message.tags}}">
53                 <a class="close" data-dismiss="alert">×</a>
54                 {{message}}
55             </div>
56         {% endfor %}
57     {% endif %}
58 </div>
59
60 <div class="container">
61     {% block content %}
62     {% endblock %}
63 </div> <!-- /container -->
64 </body>
65 </html>

```

NOTE: Make sure to put the *base.html* template in the main “templates” directory, while the other templates go in the “blog” directory. You could use this same base template for all your apps.

2. Update *index.html*:

```

1  {% extends 'base.html' %}
2
3  {% block content %}
4  {% if latest_posts %}
5  <ul>
6      {% for post in latest_posts %}
7          <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
8          <p><em>{{ post.created_at }}</em></p>
9          <p>{{ post.content }}</p>

```

```

10     <br>
11     {% endfor %}
12 </ul>
13 {% endif %}
14 {% endblock %}

```

3. Update *post.html*:

```

1  {% extends 'base.html' %}
2
3  {% block content %}
4  <h2>{{ single_post.title }}</h2>
5  <ul>
6      <p><em>{{ single_post.created_at }}</em></p>
7      <p>{{ single_post.content }}</p>
8      <p>Tag: {{ single_post.tag }}</p>
9      <br>
10     <p>
11         {% if single_post.image %}
12             
13         {% endif %}
14     </p>
15 </ul>
16 <p>Had enough? Return <a href="/blog">home</a>.<br>
17 {% endblock %}

```

Take a look at the results. Amazing what five minutes (and [Bootstrap 3](#)) can do.

Update the Index View

Next, let's go ahead and update the Index View to return the top five posts by popularity (most views). If there are less than five posts, this will return all posts.

1. Update `*views.py`:

```
1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     popular_posts = Post.objects.order_by('-views')[:5]
4     t = loader.get_template('blog/index.html')
5     context_dict = {'latest_posts': latest_posts, 'popular_posts' :
6                     popular_posts,}
7     for post in latest_posts:
8         post.url = post.title.replace(' ', '_')
9     c = Context(context_dict)
10    return HttpResponse(t.render(c))
```

2. Update the `index.html` template, so that it loops through the `popular_posts`:

```
1 <h4>Must See:</h4>
2 <ul>
3     {% for popular_post in popular_posts %}
4         <li><a href="/blog/{{ popular_post.title }}">{{ popular_post.title
5         }}</a></li>
6     {% endfor %}
7 </ul>
```

Updated file:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 {% if latest_posts %}
5 <ul>
6     {% for post in latest_posts %}
7         <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
8         <p><em>{{ post.created_at }}</em></p>
9         <p>{{ post.content }}</p>
10        <br>
11    {% endfor %}
12 </ul>
13 {% endif %}
14 <h4>Must See:</h4>
15 <ul>
16     {% for popular_post in popular_posts %}
```

```

17     <li><a href="/blog/{{ popular_post.title }}">{{ popular_post.title
18         }}</a></li>
19     {% endfor %}
20 </ul>
    {% endblock %}

```

3. Next let's add the Bootstrap [Grid System](#) to our *index.html* template:

```

1  {% extends 'base.html' %}
2
3  {% block content %}
4  <div class="row">
5      <div class="col-md-8">
6          {% if latest_posts %}
7          <ul>
8              {% for post in latest_posts %}
9                  <h3><a href="/blog/{{ post.url }}">{{ post.title }}</a></h3>
10                 <p><em>{{ post.created_at }}</em></p>
11                 <p>{{ post.content }}</p>
12                 <br>
13             {% endfor %}
14         </ul>
15         {% endif %}
16     </div>
17     <div class="col-md-4">
18         <h4>Must See:</h4>
19         <ul>
20             {% for popular_post in popular_posts %}
21                 <li><a href="/blog/{{ popular_post.title }}">{{ popular_post.title
22                     }}</a></li>
23             {% endfor %}
24         </ul>
25     </div>
    {% endblock %}

```

For more info on how the Grid System works, please view [this](#) blog post.

4. If you look at the actual URLs for the popular posts, you'll see that they have the original URLs, not the friendly URLs that we'd like to see. Let's change that.

The easiest way to correct this is to just add another `for` loop to the Index View:

```

1  for popular_post in popular_posts:
2      popular_post.url = popular_post.title.replace(' ', '_')

```

Then update the URL within the *index.html* file:

```

1  <li><a href="/blog/{{ popular_post.url }}">{{ popular_post.title }}</a></li>

```

Test this out. It should work.

5. Finally, let's add a `encode_url()` function to `views.py` to clean up the code:

```
1 def index(request):
2     latest_posts = Post.objects.all().order_by('-created_at')
3     popular_posts = Post.objects.order_by('-views')[:5]
4     t = loader.get_template('blog/index.html')
5     context_dict = {'latest_posts': latest_posts, 'popular_posts' :
6                     popular_posts,}
7     for post in latest_posts:
8         post.url = encode_url(post.title)
9     for popular_post in popular_posts:
10        popular_post.url = encode_url(popular_post.title)
11    c = Context(context_dict)
12    return HttpResponse(t.render(c))
13
14 # helper function
15 def encode_url(url):
16     return url.replace(' ', '_')
```

Test this out again.

Homework

- Update the Bootstrap Grid in the `posts.html` file to show the popular posts.

Forms

In this last section, we'll look at adding forms to our Blog to allow end-users to add posts utilizing Django's built-in [form handling](#) features. Such features allow you to ⁵:

1. display an HTML form with automatically generated form widgets (like a text field or date picker);
2. check submitted data against a set of validation rules;
3. redisplay a form in case of validation errors; and
4. convert submitted form data to the relevant Python data types.

In short, forms take user data, validate the data, and then convert the data to Python objects.

Since we have a database-driven application, we will be taking advantage of a particular type of form called a [ModelForm](#). These forms map the user input to specific columns in the database.

To simplify the process of form creation, let's split the workflow into four steps:

1. Create a *forms.py* file to add fields to your form
2. Add the handling of the form logic to the View
3. Add or update a template to display the form
4. Add a urlpattern for the new View

Let's get started.

Create a *forms.py* file to add fields to your form

Include the following code in *blog/forms.py*:

```
1 from django import forms
2 from blog.models import Post
3
4 class PostForm(forms.ModelForm):
5     created_at = forms.DateTimeField(widget=forms.HiddenInput(), required=False)
6     title = forms.CharField(max_length=100)
7     content = forms.CharField(widget=forms.Textarea())
8     tag = forms.CharField(required=False, max_length=20)
9     image = forms.ImageField(required=False)
10    views = forms.IntegerField(widget=forms.HiddenInput(), initial=0)
11
12    class Meta:
13        model = Post
```

⁵<https://docs.djangoproject.com/en/1.5/topics/forms/>

Here we're creating a new `ModelForm` that's mapped to our model via the `Meta()` inner class: `model = Post`.

Notice how each of our form fields has an associated column in the database. This is required.

Add the handling of the form logic to the View

Next, let's update our app's view for handling the form logic - e.g., displaying the form, saving the form data, alerting the user about validation errors, etc.

Add the following code to `blog/views.py`:

```
1 def add_post(request):
2     context = RequestContext(request)
3     if request.method == 'POST':
4         form = PostForm(request.POST, request.FILES)
5         if form.is_valid(): # is the form valid?
6             form.save(commit=True) # yes? save to database
7             return redirect(index)
8         else:
9             print form.errors # no? display errors to end user
10    else:
11        form = PostForm()
12    return render_to_response('blog/add_post.html', {'form': form}, context)
```

Also be sure to update the imports:

```
1 from django.http import HttpResponseRedirect
2 from django.template import Context, loader, RequestContext
3 from django.shortcuts import get_object_or_404, render_to_response, redirect
4
5 from blog.models import Post
6 from blog.forms import PostForm
```

What's going on here?

1. First we determine if the request is a GET or POST. If the former, we are simply going to display the form, `form = PostForm()`; if the latter, we are going to process the form data:

```
1 form = PostForm(request.POST, request.FILES)
2 if form.is_valid(): # is the form valid?
3     form.save(commit=True) # yes? save to database
4     return redirect(index)
5 else:
6     print form.errors # no? display errors to end user
```

2. If it's a POST request, we first determine if the supplied data is valid or not.

Essentially, forms have two different types of validation that are triggered when `is_valid()` is called on a form: field and form validation.

Field validation, which happens at the form level, validates the user inputs against the arguments specified in the `ModelForm` - i.e., `max_length=100`, `required=False`, etc. Be sure to look over the official Django Documentation on [Widgets](#) to see the available fields and the parameters that each can take.

Once the fields are validated, the values are converted over to Python objects and then [form validation](#) occurs via the form's `clean` method. Read more about this method [here](#).

Validation ensures that Django does not add any data to the database from a submitted form that could potentially harm your database.

Again, each of these forms of validation happen implicitly as soon as the `is_valid()` method is called. You can, however, customize the process. Read more about the overall process from the links above or for a more detailed workflow, please see the Django form documentation [here](#).

3. After the data is validated, Django either saves the data to the database, `form.save(commit=True)` and redirects the user to the index page or outputs the errors to the end user.

NOTE: Did you notice the `render_to_response()` function. We'll discuss this more in the final Django chapter

Add or update a template to display the form

Moving on, let's create the template called `add_posts.html` within our "templates/blog" directory:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 <h2>Add a Post</h2>
5 <form id="post_form" method="post" action="/blog/add_post/"
6     enctype="multipart/form-data">
7     {% csrf_token %}
8     {% for hidden in form.hidden_fields %}
9         {{ hidden }}
10    {% endfor %}
11
12    {% for field in form.visible_fields %}
13        {{ field.errors }}
14        {{ field.label_tag }}
15        {{ field }}
16    {% endfor %}
17
18    <input type="submit" name="submit" value="Create Post">
```

```
18 </form>
19 {% endblock %}
```

Here, we have a `<form>` element, which loops through both the visible and hidden fields. Only the visible fields will produce markup. Read more about such loops [here](#).

Also, within our visible fields loop, we are displaying the validation errors, `{{ field.errors }}` as well as the name of the field, `label_tag`. You never want to do this for the hidden fields since this will really confuse the end user. You just want to have tests in place to ensure that the hidden fields generate valid data each and every time. We'll discuss this in a bit.

Finally, did you notice the `{% csrf_token %}` tag? This is a Cross-Site Request Forgery (CSRF) token, meant to help secure that a specific POST request is initiated from a form. This is required by Django. Please read more about it from the official Django [documentation](#).

Alternatively, if you want to keep it simple, you can render the form with one tag:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 <h2>Add a Post</h2>
5 <form id="post_form" method="post" action="/blog/add_post/"
6     enctype="multipart/form-data">
7     {% csrf_token %}
8
9     {{ form }}
10
11     <input type="submit" name="submit" value="Create Post">
12 </form>
13 {% endblock %}
```

Test this out. Since we're not adding any custom features to the form in the above template, we can use just this simplified template and get the same functionality. If you are interested in further customization, check out the Django [documentation](#).

NOTE: When you want users to upload files via a form, you must set the `enctype` to `multipart/form-data`. If you do not remember to do this, you won't get an error; the upload just won't be saved - so, it's vital that you remember to do this. You could even write your templates in the following manner to ensure that you include `enctype="multipart/form-data"` in case you don't know whether users will be uploading files ahead of time:

```
1 {% if form.is_multipart %}
2     <form enctype="multipart/form-data" method="post" action="">
3 {% else %}
4     <form method="post" action="">
5 {% endif %}
```

```
6 {{ form }}
7 </form>
```

Add a urlpattern for the new View

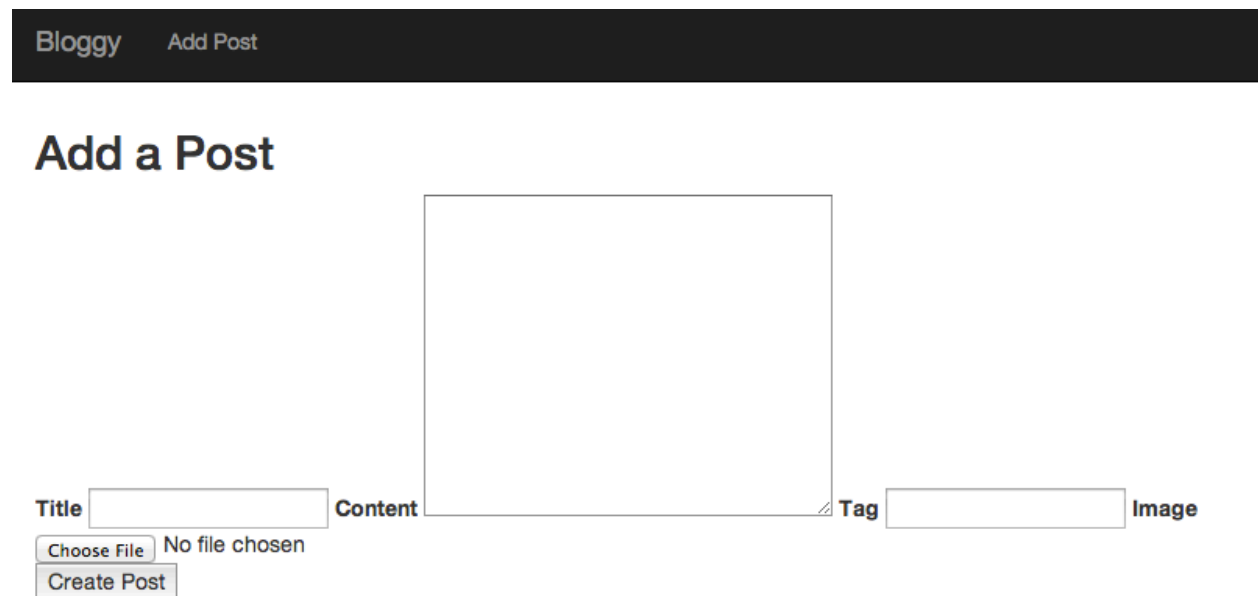
Now we just need to map the view, `add_post` to a URL:

```
1 from django.conf.urls import patterns, include, url
2 from blog import views
3
4 urlpatterns = patterns('',
5     url(r'^$', views.index, name='index'),
6     url(r'^add_post/$', views.add_post, name='add_post'), # form
7     url(r'^(?P<post_name>\w+)/$', views.post, name='post'),
8 )
```

That's it. Just add the proper link to our *base.html* template - `Add Post` - then test it out.

Styling Forms

Navigate to the form template: http://localhost:8000/blog/add_post/.



The screenshot shows a web browser window with a dark header bar containing the text "Bloggy" and "Add Post". Below the header, the page title "Add a Post" is displayed. The main content area contains a form with a large text input field labeled "Content". To the left of the "Content" field is a "Title" label and a text input field. To the right of the "Content" field is a "Tag" label and a text input field. Below the "Content" field is an "Image" label and a text input field. At the bottom of the form, there are three buttons: "Choose File", "No file chosen", and "Create Post".

Figure 21.6: `bloggy_form`

Looks pretty bad. We can clean this up quickly with Bootstrap, specifically with the [Django Forms Bootstrap](#) package.

Setup is simple:

1. Install the package with pip - `pip install django-forms-bootstrap`
2. Add the package to your `INSTALLED_APPS` tuple:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'blog',  
9     'django.contrib.admin',  
10    'django.contrib.admindocs',  
11    'south',  
12    'django_forms_bootstrap',  
13 )
```

3. Then simply update your template:

```
1 {% extends 'base.html' %}  
2  
3 {% block content %}  
4 {% load bootstrap_tags %}  
5 <h2>Add a Post</h2>  
6 <form id="post_form" method="post" action="/blog/add_post/"  
7     enctype="multipart/form-data">  
8     {% csrf_token %}  
9  
10    {{ form | as_bootstrap }}  
11  
12    <input type="submit" name="submit" value="Create Post" class="btn  
13        btn-primary">  
14 </form>  
15 {% endblock %}
```

Restart the server. Check out the results:

Homework

- See if you can change the redirect URL after a successful form submission, `return redirect(index)`, to redirect to the newly created post.

What else could you add to this app? Comments? User Login/Authentication?

Add a Post

Title

Content

Tag

Image

Choose File

No file chosen

Create Post

Figure 21.7: bloggy_form_2

Homework

- Go over the Django Quick-start 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.

Chapter 22

Django: Ecommerce Site

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, - 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 e-commerce site.

First, let's talk about Rapid Web Development.

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 mock-up through iterations: Prototype, Review, Refine. Repeat. 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 mock-up 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? Will the end user care? They better. Will this differentiate 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 start-ups. Speed is the main advantage that start-ups have over their larger, more established competition. The key is to understand each layer of the development process, from beginning to end.

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.

NOTE 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 (MVP). 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.

Setup your Project and App

NOTE: We are purposely creating this project using an older version of Django and Bootstrap. In the next course you will learn how to take this project structure and upgrade to the latest

Basic Setup

1. Create a new directory called “django_mvp”. Create and activate a virtualenv with that directory.

2. Install Django:

```
1 $ pip install django==1.5.5
```

3. Start a new Django project:

```
1 $ django-admin.py startproject mvp
```

4. Add your Django project folder to Sublime as a new project.

5. Setup a new app:

```
1 $ python manage.py startapp main
```

NOTE: 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:

```
1
2 main
3     __init__.py
4     models.py
5     tests.py
6     views.py
7 manage.py
8 mvp
9     __init__.py
10    settings.py
11    urls.py
12    wsgi.py
```

6. Add your app to the `INSTALLED_APPS` section of `settings.py` as well as your project’s absolute paths (add this to the top of `settings.py`):

```

1 INSTALLED_APPS = (
2     'django.contrib.auth',
3     'django.contrib.contenttypes',
4     'django.contrib.sessions',
5     'django.contrib.sites',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'main',

```

and

```

1 from os.path import abspath, dirname, normpath, join
2
3 DJANGO_ROOT = dirname(dirname(abspath(__file__)))

```

7. Create a new directory within the root directory called “templates”, and then add the absolute path to the *settings.py* file:

```

1 TEMPLATE_DIRS = (
2     normpath(join(DJANGO_ROOT, 'templates')),
3 )

```

8. Configure the admin page.

Uncomment the following url pattern in *settings.py*:

```

1 'django.contrib.admin',

```

Uncomment these three lines from *urls.py*:

```

1 from django.contrib import admin
2 admin.autodiscover()
3
4 url(r'^admin/', include(admin.site.urls)),

```

WARNING: 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 in production as you do in development. We will set up a *requirements.txt* file later to handle this. Finally, it may be necessary to develop on a virtual machine to recreate the exact OS and even the same OS release so that bugs found within production are easily reproducible in development. The bigger the app, the greater the need for this. Much of this will be addressed in the next course.

Add a landing page

Now that the main app is up, let's quickly add a bare bones landing page.

1. Add the following code to the app's *views.py* file:

```
1 from django.shortcuts import render_to_response
2 from django.template import RequestContext
3
4 def index(request):
5     return render_to_response('index.html',
6                               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 our Project's *urls.py*:

```
1 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* (the parent template) within the templates directory, and add the following code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Your MVP</title>
6     <meta name="viewport" content="width=device-width, initial-scale=1.0">
7     <meta name="description" content="">
8     <meta name="author" content="">
9
10    <!-- Le styles -->
11    <link
12      href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css"
13      rel="stylesheet">
14    <style>
15
16      body {
17        padding-top: 20px;
18        padding-bottom: 40px;
19      }
20
21      /* Custom container */
22      .container-narrow {
```

```

21     margin: 0 auto;
22     max-width: 700px;
23 }
24 .container-narrow > hr {
25     margin: 30px 0;
26 }
27
28 /* Main marketing message and sign up button */
29 .jumbotron {
30     margin: 60px 0;
31     text-align: center;
32 }
33 .jumbotron h1 {
34     font-size: 72px;
35     line-height: 1;
36 }
37 .jumbotron .btn {
38     font-size: 21px;
39     padding: 14px 24px;
40 }
41
42 /* Supporting marketing content */
43 .marketing {
44     margin: 60px 0;
45 }
46 .marketing p + h4 {
47     margin-top: 28px;
48 }
49
50 </style>
51 <script src="http://code.jquery.com/jquery-1.11.0.min.js"
52     type="text/javascript"></script>
53     {% block extrahead %}
54     {% endblock %}
55 <script type="text/javascript">
56     $(function(){
57         {% block jquery %}
58         {% endblock %}
59     });
60 </script>
61 </head>
62
63 <body>
64
65     <div class="container-narrow">
66
67         <div class="masthead">

```

```

67     <ul class="nav nav-pills pull-right">
68         <li><a href="{% url 'home' %}">Home</a></li>
69         <li><a href="#">About</a></li>
70         <li><a href="#">Contact</a></li>
71     </ul>
72     <h3><span class="fui-settings-16 muted">Your MVP!</span></h3>
73 </div>
74
75 <hr>
76
77 {% if messages %}
78 <div class="alert alert-success">
79     <div class="messages">
80         {% for message in messages %}
81             {{ message }}
82         {% endfor %}
83     </div>
84 </div>
85 {% endif %}
86
87 <div class="container-narrow">
88     {% block content %}
89     {% endblock %}
90 </div>
91
92 <hr>
93
94 <div class="footer">
95     <p>Copyright © 2014 <a href="http://www.yoursite.com">Your MVP</a>.
96     Developed by <a href="http://www.mherman.org">Your Name Here</a>.
97     Powered by Django.</P></p>
98 </div>
99
100 </body>
101 </html>

```

Next, add the *index.html* (child) template:

```

1 {% extends 'base.html' %}
2
3 {% block content %}
4
5 <div class="jumbotron">
6     
8     <h1>Please put some text here.</h1>

```


[illegible]

```

34 </div>
35
36 <div class="span6">
37   <h3>(4) Four</h3>
38   <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>
39   <h3>(5) Five</h3>
40   <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>
41   <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>
42   <h3>(6) Six</h3>
43   <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>
44   </p>
45 </div>
46
47 </div>
48
49 {% endblock %}

```

4. Fire up your server:

```

1 $ python manage.py runserver

```

Looking good so far, but let's add some more pages. Before that, though, let's take a step back and talk about Bootstrap.

Bootstrap

We used the [Bootstrap](#) front end framework to quickly add a basic design. To avoid looking too much like just a generic Bootstrap site, the design 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 good 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 a trial and error process. You just have to make a few changes, then refresh the browser, see how they look, and then make more changes or update or revert old changes. Again, it takes time to know what will look good. Eventually, after

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](#)

Figure 22.1: mvp part 1

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.

SEE ALSO: If you're working on your own application, [Jetstrap](#) is a great resource for creating quick Bootstrap prototypes. Try it out.

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*:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11 )
```

Update the DATABASES section in *settings.py*:

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         'NAME': 'test.db'  
5     }  
6 }
```

Update the MIDDLEWARE_CLASSES of *settings.py*:

```
1 MIDDLEWARE_CLASSES = (  
2     'django.middleware.common.CommonMiddleware',  
3     'django.contrib.sessions.middleware.SessionMiddleware',  
4     'django.middleware.csrf.CsrfViewMiddleware',  
5     'django.contrib.auth.middleware.AuthenticationMiddleware',  
6     'django.contrib.messages.middleware.MessageMiddleware',  
7     'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',  
8 )
```

Add the following pattern to *urls.py*:

¹<https://docs.djangoproject.com/en/1.5/ref/contrib/flatpages/>

```
1 url(r'^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:

```
1 {% extends 'base.html' %}
2
3 {% block content %}
4 {{ flatpage.content }}
5 {% endblock %}
```

2. Sync the database and create a superuser:

```
1 $ python manage.py syncdb
```

NOTE: The syncdb command searches your *models.py* files and creates database tables for them.

3. Update the About url in the *base.html* file:

```
1 <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:

The HTML within the content portion:

```
1 <br>
2
3 <p>You can add some text about yourself here. Then when you are done, just
4   add a closing HTML paragraph tag.</p>
5
6 <ul>
7   <li>Bullet Point # 1</li>
8   <li>Bullet Point # 2</li>
9   <li>Bullet Point # 3</li>
10  <li>Bullet Point # 4</li>
11 </ul>
12 <br/>
13 <p>You can add a link or an email address <a
14   href="http://www.realpython.com">here</a> if you want. Again, you don't
15   have to, but I <strong>highly</strong> recommend it. Cheers! </em></p>
16
17 <h3>Ready? Contact us!</h3>
```

Change flat page

URL:	<input type="text" value="/pages/about/"/>
Example: '/about/contact/'. Make sure to have leading and trailing slashes.	
Title:	<input type="text" value="About"/>
Content:	<div><p>
</p><p><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></p><p></p><p>Bullet Point # 1</p><p>Bullet Point # 2</p><p>Bullet Point # 3</p><p>Bullet Point # 4</p><p></p><p>
</p><p><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></p><p><h3>Ready? Contact us!</h3></p></div>

Figure 22.2: django flat page

Now navigate to <http://localhost:8000/pages/about/> to view the new About page.

Nice, right? Next, let's add a Contact Us page.

Contact App

1. Create a new app:

```
1 $ python manage.py startapp contact
```

2. Add the new app to your *settings.py* file:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11    'contact',
```

3. Setup a new model to create a database table:

```
1 from django.db import models  
2 import datetime  
3  
4 class ContactForm(models.Model):  
5     name = models.CharField(max_length=150)  
6     email = models.EmailField(max_length=250)  
7     topic = models.CharField(max_length=200)  
8     message = models.CharField(max_length=1000)  
9     timestamp = models.DateTimeField(auto_now_add=True,  
10                                     default=datetime.datetime.now)  
11  
12     def __unicode__(self):  
13         return self.email  
14  
15 class Meta:  
16     ordering = ['-timestamp']
```

4. Sync the database:

```
1 $ python manage.py syncdb
```

5. Add a view:

```
1 from django.shortcuts import render_to_response
```

```

2 from django.http import HttpResponseRedirect
3 from django.template import RequestContext, loader
4 from .forms import ContactView
5 from django.contrib import messages
6
7 def contact(request):
8     form = ContactView(request.POST or None)
9     if form.is_valid():
10         our_form = form.save(commit=False)
11         our_form.save()
12         messages.add_message(request, messages.INFO, 'Your message has been
13 sent. Thank you.')
14         return HttpResponseRedirect('/')
15     t = loader.get_template('contact.html')
16     c = RequestContext(request, {'form': form,})
17     return HttpResponse(t.render(c))

```

6. Update your Project's *urls.py* with the following pattern:

```

1 url(r'^contact/', 'contact.views.contact', name='contact'),

```

7. Create a new file within the “contacts” directory called *forms.py*:

```

1 from django.forms import ModelForm
2 from .models import ContactForm
3 from django import forms
4
5 class ContactView(ModelForm):
6     message = forms.CharField(widget=forms.Textarea)
7     class Meta:
8         model = ContactForm

```

8. Add another new file called *admin.py*:

```

1 from django.contrib import admin
2 from .models import ContactForm
3
4
5 class ContactFormAdmin(admin.ModelAdmin):
6     class Meta:
7         model = ContactForm
8
9 admin.site.register(ContactForm, ContactFormAdmin)

```

9. Add a new file to the templates directory called *contact.html*:


```

1 {% extends 'base.html' %}
2
3
4 {% block content %}
5
6 <h3><center>Contact Us</center></h3>
7
8 <form action='.' method='POST'>
9 {% csrf_token %}
10
11 {{ form.as_table }}
12 <br>
13 <button type='submit' class='btn'>Submit</button>
14 </form>
15
16 {% endblock %}

```

10. Update the Contact url in the *base.html* file:

```

1 <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 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.

User Registration upon Payment

In this last section, we will be looking at how to implement a payment and user registration/authentication system. For this payment system, 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 path to `STATICFILES_DIRS` in the `settings.py` file and add the “static” directory to the “project_name” directory:

```
1 STATICFILES_DIRS = (  
2     normpath(join(DJANGO_ROOT, 'static')),  
3 )
```

2. Create a new app:

```
1 $ python manage.py startapp payments
```

3. Add the new app to `settings.py`:

```
1 INSTALLED_APPS = (  
2     'django.contrib.auth',  
3     'django.contrib.contenttypes',  
4     'django.contrib.sessions',  
5     'django.contrib.sites',  
6     'django.contrib.messages',  
7     'django.contrib.staticfiles',  
8     'django.contrib.admin',  
9     'main',  
10    'django.contrib.flatpages',  
11    'contact',  
12    'payments',
```

4. Install stripe:

```
1 $ pip install stripe==1.9.2
```

5. Update `models.py`:

```
1 from django.db import models  
2  
3 class User(models.Model):  
4     name = models.CharField(max_length=255)  
5     email = models.CharField(max_length=255, unique=True)
```

```

6     password = models.CharField(max_length=60)
7     last_4_digits = models.CharField(max_length=4, blank=True, null=True)
8     stripe_id = models.CharField(max_length=255)
9     created_at = models.DateTimeField(auto_now_add=True)
10    updated_at = models.DateTimeField(auto_now=True)
11
12    def __str__(self):
13        return self.email

```

NOTE: This model replaces the default User model.

6. Sync the database:

```

1 $ python manage.py syncdb

```

7. Add the following import, `from payments import views`, and patterns to `urls.py`:

```

1 url(r'^sign_in$', views.sign_in, name='sign_in'),
2 url(r'^sign_out$', views.sign_out, name='sign_out'),
3 url(r'^register$', views.register, name='register'),
4 url(r'^edit$', views.edit, name='edit'),

```

8. Add the remaining templates and all of the static files from the files in the [repository](#).

9. Add a new file called `admin.py`:

```

1 from django.contrib import admin
2 from .models import User
3
4 class UserAdmin(admin.ModelAdmin):
5     class Meta:
6         model = User
7
8 admin.site.register(User, UserAdmin)

```

10. Add another new file called `forms.py`:

```

1 from django import forms
2 from django.core.exceptions import NON_FIELD_ERRORS
3 from django.utils.translation import ugettext_lazy as _
4
5 class PaymentForm(forms.Form):
6     def addError(self, message):
7         self._errors[NON_FIELD_ERRORS] = self.error_class([message])
8
9 class SigninForm(PaymentForm):

```

```

10     email = forms.EmailField(required = True)
11     password = forms.CharField(required = True,
12                                widget=forms.PasswordInput(render_value=False))
13
14 class CardForm(PaymentForm):
15     last_4_digits = forms.CharField(required = True, min_length = 4,
16                                     max_length = 4, widget = forms.HiddenInput())
17     stripe_token = forms.CharField(required = True, widget =
18                                     forms.HiddenInput())
19
20 class UserForm(CardForm):
21     name = forms.CharField(required = True)
22     email = forms.EmailField(required = True)
23     password = forms.CharField(required = True, label=(u'Password'),
24                                widget=forms.PasswordInput(render_value=False))
25     ver_password = forms.CharField(required = True, label=(u' Verify
26                                Password'), widget=forms.PasswordInput(render_value=False))
27
28     def clean(self):
29         cleaned_data = self.cleaned_data
30         password = cleaned_data.get('password')
31         ver_password = cleaned_data.get('ver_password')
32         if password != ver_password:
33             raise forms.ValidationError('Passwords do not match')
34         return cleaned_data

```

11. Update *payments/views.py*:

```

1 from django.db import IntegrityError
2 from django.http import HttpResponseRedirect
3 from django.shortcuts import render_to_response, redirect
4 from django.template import RequestContext
5 from payments.forms import PaymentForm, SigninForm, CardForm, UserForm
6 from payments.models import User
7 import mvp.settings as settings
8 import stripe
9 import datetime
10
11 stripe.api_key = settings.STRIPE_SECRET
12
13 def soon():
14     soon = datetime.date.today() + datetime.timedelta(days=30)
15     return {'month': soon.month, 'year': soon.year}
16
17 def sign_in(request):
18     user = None
19     if request.method == 'POST':

```

```

20     form = SigninForm(request.POST)
21     if form.is_valid():
22         results = User.objects.filter(email=form.cleaned_data['email'])
23         if len(results) == 1:
24             if results[0].check_password(form.cleaned_data['password']):
25                 request.session['user'] = results[0].pk
26                 return HttpResponseRedirect('/')
27             else:
28                 form.addError('Incorrect email address or password')
29         else:
30             form.addError('Incorrect email address or password')
31     else:
32         form = SigninForm()
33
34     print form.non_field_errors()
35
36     return render_to_response(
37         'sign_in.html',
38         {
39             'form': form,
40             'user': user
41         },
42         context_instance=RequestContext(request)
43     )
44
45 def sign_out(request):
46     del request.session['user']
47     return HttpResponseRedirect('/')
48
49 def register(request):
50     user = None
51     if request.method == 'POST':
52         form = UserForm(request.POST)
53         if form.is_valid():
54
55             #update based on your billing method (subscription vs one time)
56             customer = stripe.Customer.create(
57                 email = form.cleaned_data['email'],
58                 description = form.cleaned_data['name'],
59                 card = form.cleaned_data['stripe_token'],
60                 plan="gold",
61             )
62             # customer = stripe.Charge.create(
63             #     description = form.cleaned_data['email'],
64             #     card = form.cleaned_data['stripe_token'],
65             #     amount="5000",
66             #     currency="usd"

```

```

67         # )
68
69         user = User(
70             name = form.cleaned_data['name'],
71             email = form.cleaned_data['email'],
72             last_4_digits = form.cleaned_data['last_4_digits'],
73             stripe_id = customer.id,
74             password = form.cleaned_data['password']
75         )
76
77         # user.set_password(form.cleaned_data['password'])
78
79         try:
80             user.save()
81         except IntegrityError:
82             form.addError(user.email + ' is already a member')
83         else:
84             request.session['user'] = user.pk
85             return HttpResponseRedirect('/')
86
87     else:
88         form = UserForm()
89
90     return render_to_response(
91         'register.html',
92         {
93             'form': form,
94             'months': range(1, 12),
95             'publishable': settings.STRIPE_PUBLISHABLE,
96             'soon': soon(),
97             'user': user,
98             'years': range(2011, 2036),
99         },
100         context_instance=RequestContext(request)
101     )
102
103 def edit(request):
104     uid = request.session.get('user')
105
106     if uid is None:
107         return HttpResponseRedirect('/')
108
109     user = User.objects.get(pk=uid)
110
111     if request.method == 'POST':
112         form = CardForm(request.POST)
113         if form.is_valid():

```

```

114         customer = stripe.Customer.retrieve(user.stripe_id)
115         customer.card = form.cleaned_data['stripe_token']
116         customer.save()
117
118
119         user.last_4_digits = form.cleaned_data['last_4_digits']
120         user.stripe_id = customer.id
121         user.save()
122
123         return HttpResponseRedirect('/')
124
125     else:
126         form = CardForm()
127
128     return render_to_response(
129         'edit.html',
130         {
131             'form': form,
132             'publishable': settings.STRIPE_PUBLISHABLE,
133             'soon': soon(),
134             'months': range(1, 12),
135             'years': range(2011, 2036)
136         },
137         context_instance=RequestContext(request)
138     )

```

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:

```

1  #update based on your billing method (subscription vs one time)
2  #customer = stripe.Customer.create(
3  #email = form.cleaned_data['email'],
4  #description = form.cleaned_data['name'],
5  #card = form.cleaned_data['stripe_token'],
6  #plan="gold",
7  #)
8  customer = stripe.Charge.create(
9  description = form.cleaned_data['email'],
10 card = form.cleaned_data['stripe_token'],
11 amount="5000",
12 currency="usd"
13 )

```

12. Update *main/views.py*:

```

1 from django.shortcuts import render_to_response
2 from django.template import RequestContext

```

```

3 from payments.models import User
4
5 def index(request):
6     uid = request.session.get('user')
7     if uid is None:
8         return render_to_response('index.html')
9     else:
10        return render_to_response('user.html', {'user':
            User.objects.get(pk=uid)})

```

13. Update the *base.html* template:

```

1 {% load static %}
2
3 <!DOCTYPE html>
4 <html lang="en">
5     <head>
6         <meta charset="utf-8">
7         <title>Your MVP</title>
8         <meta name="viewport" content="width=device-width, initial-scale=1.0">
9         <meta name="description" content="">
10        <meta name="author" content="">
11
12        <!-- Le styles -->
13        <link
14            href="//netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css"
15            rel="stylesheet">
16        <style>
17
18            body {
19                padding-top: 20px;
20                padding-bottom: 40px;
21            }
22
23            /* Custom container */
24            .container-narrow {
25                margin: 0 auto;
26                max-width: 700px;
27            }
28            .container-narrow > hr {
29                margin: 30px 0;
30            }
31
32            /* Main marketing message and sign up button */
33            .jumbotron {
34                margin: 60px 0;
35                text-align: center;

```



```

34     }
35     .jumbotron h1 {
36         font-size: 72px;
37         line-height: 1;
38     }
39     .jumbotron .btn {
40         font-size: 21px;
41         padding: 14px 24px;
42     }
43
44     /* Supporting marketing content */
45     .marketing {
46         margin: 60px 0;
47     }
48     .marketing p + h4 {
49         margin-top: 28px;
50     }
51
52 </style>
53 <script src="http://code.jquery.com/jquery-1.11.0.min.js"
54     type="text/javascript"></script>
55     {% block extrahead %}
56     {% endblock %}
57 <script type="text/javascript">
58     $(function(){
59         {% block jquery %}
60         {% endblock %}
61     });
62 </script>
63 <script src="https://js.stripe.com/v1/" type="text/javascript"></script>
64 <script type="text/javascript">
65     //<![CDATA[
66     Stripe.publishableKey = '{{ publishable }}';
67     //]]>
68 </script>
69 <script src="{% get_static_prefix %}jquery.js"
70     type="text/javascript"></script>
71 <script src="{% get_static_prefix %}application.js"
72     type="text/javascript"></script>
73 </head>
74
75 <body>
76
77     <div class="container-narrow">
78
79         <div class="masthead">
80             <ul class="nav nav-pills pull-right">

```

```

78     <li><a href="{% url 'home' %}">Home</a></li>
79     <li><a href="/pages/about">About</a></li>
80     <li><a href="{% url 'contact' %}">Contact</a></li>
81     {% if user %}
82         <li><a href="{% url 'sign_out' %}">Logout</a></li>
83     {% else %}
84         <li><a href="{% url 'sign_in' %}">Login</a></li>
85         <li><a href="{% url 'register' %}">Register</a></li>
86     {% endif %}
87     </ul>
88     <h3><span class="fui-settings-16 muted">Your MVP!</span></h3>
89 </div>
90
91 <hr>
92
93 {% if messages %}
94 <div class="alert alert-success">
95     <div class="messages">
96         {% for message in messages %}
97             {{ message }}
98         {% endfor %}
99     </div>
100 </div>
101 {% endif %}
102
103 <div class="container-narrow">
104     {% block content %}
105     {% endblock %}
106 </div>
107
108 <hr>
109
110 <div class="footer">
111     <p>Copyright © 2014 <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></div>
112
113
114 </div>
115
116 </body>
117 </html>

```

14. Update the large button on *index.html*:

```

1 <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*:

```
1 STRIPE_SECRET = 'sk_test_HgLQiBd5nbM9DdWP1IdsQYYB'  
2 STRIPE_PUBLISHABLE = 'pk_test_A1Lf9aHAN7kB6qZqn1FH5CHy'
```

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:

“python

import stripe
stripe.api_key = “sk_test_HgLQiBd5nbM9DdWP1IdsQYYB”

stripe.Plan.create(amount=2000, interval=‘month’, name=‘Amazing Gold Plan’, currency=‘usd’, id=‘gold’) “

Finally, after you process the dummy payment, make sure the user is added to the database as well as the [dashboard](#) on Stripe:

SEE ALSO: Refer to the Stripe [documentation](#) and [API reference docs](#) for more info.

Name

John Eskey

Email

jack@ryan.com

Password

.....

Verify Password

.....

Credit card number

42424242424242

Security code (CVC)

123

Expiration date

8



2013



Register

Figure 22.3: register

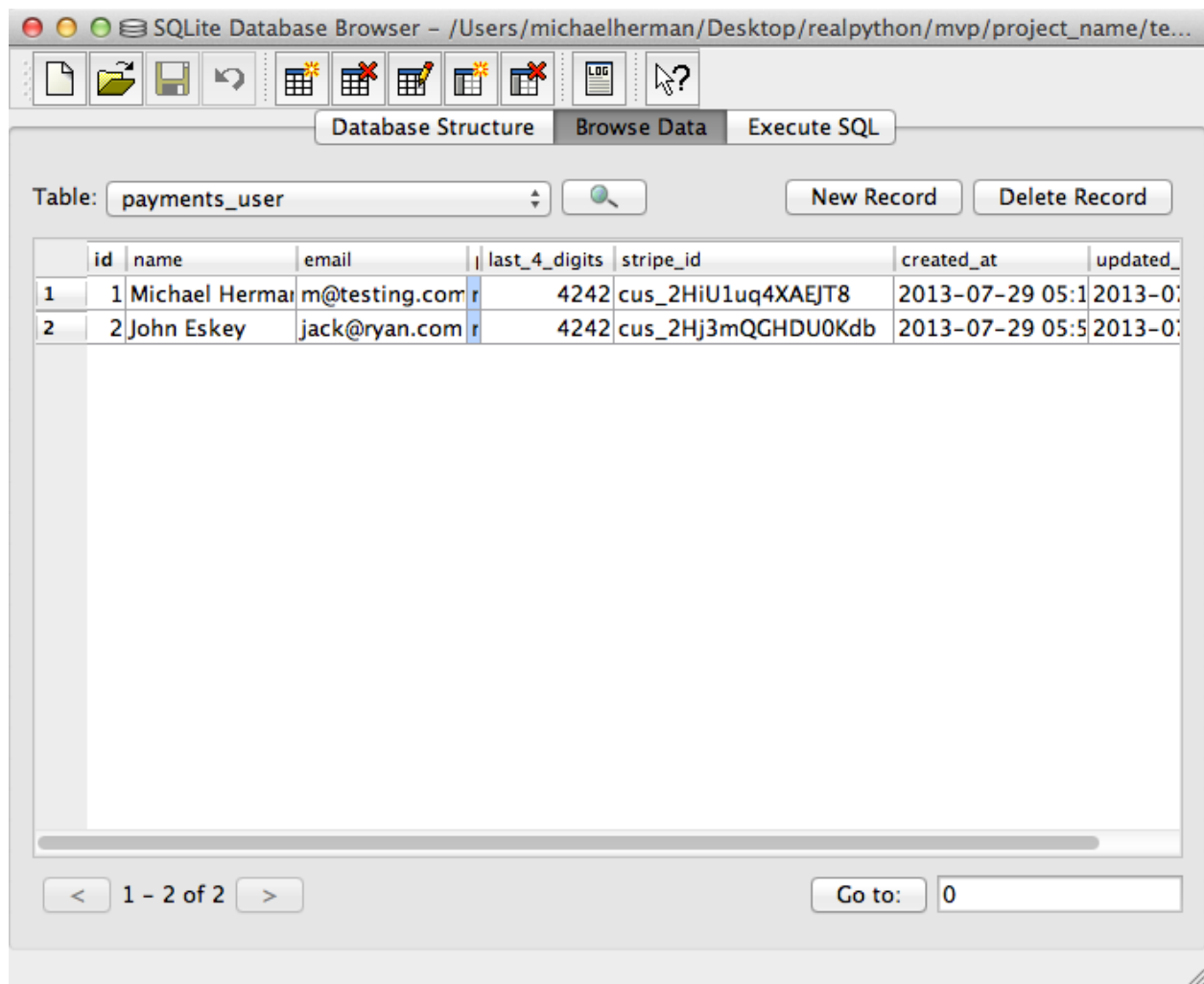


Figure 22.4: stripe database

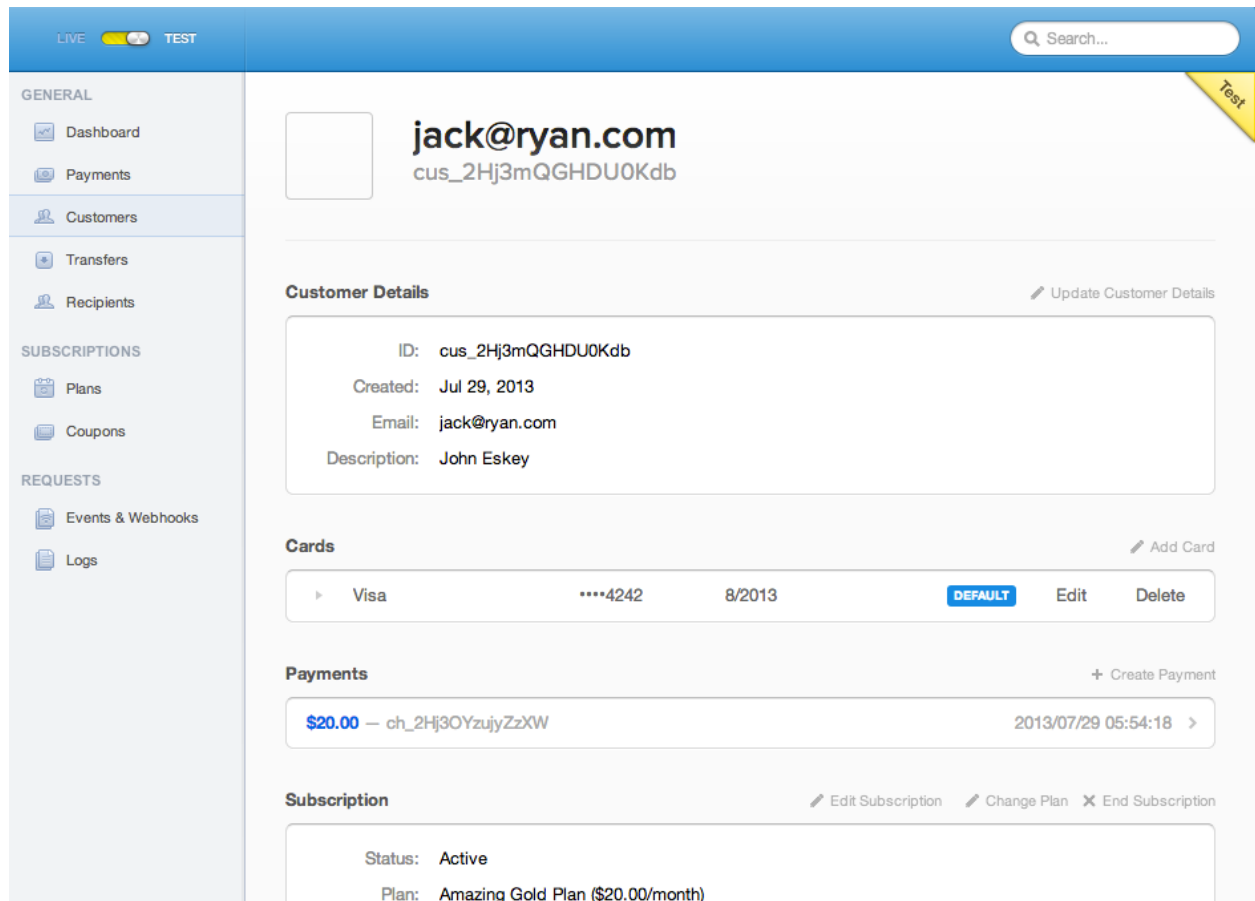


Figure 22.5: stripe

Chapter 23

Appendix A: Installing Python

Windows 7

Start by downloading Python 2.7.6 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 2.7.6, you need to add the following directories to your PATH:

- C:\Python27\
- C:\Python27\Scripts\
- C:\PYTHON27\DLLs\
- C:\PYTHON27\LIB\

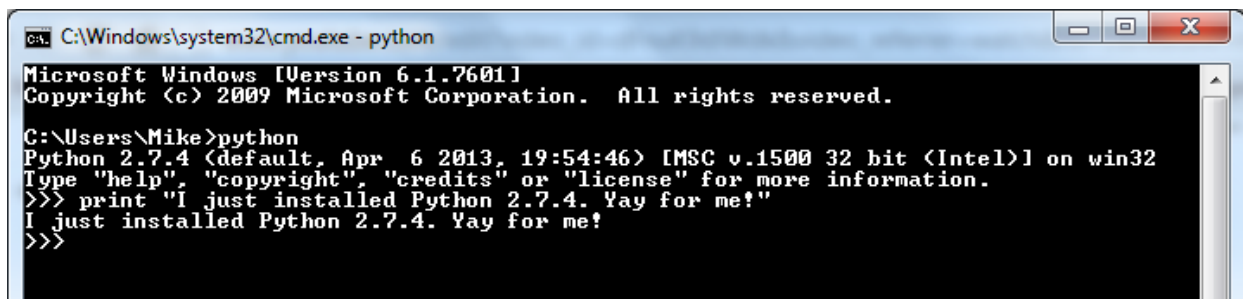
Open your power shell and run the following statement:

```
[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:

Video

Watch the video [here](#) for assistance.



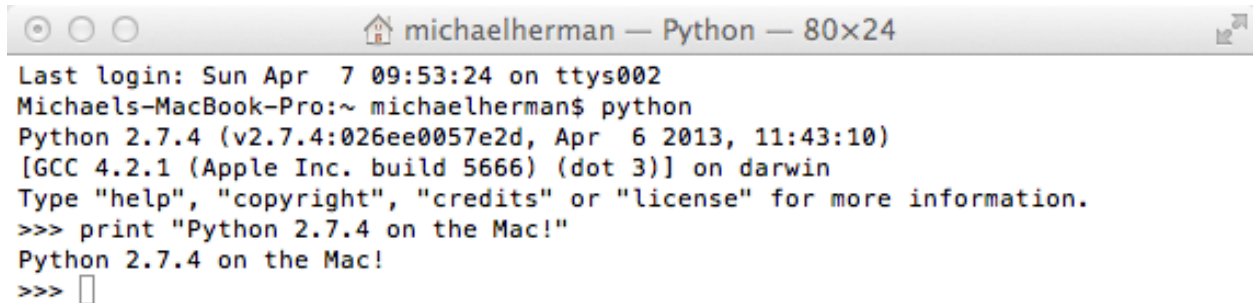
```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

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!
>>>
```

Figure 23.1: windows install

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:

A screenshot of a Mac OS X terminal window. The title bar shows a home icon, the name 'michaelherman', and the window title 'Python — 80x24'. The terminal content shows the user logging in, typing 'python', and seeing the Python 2.7.4 version information and a prompt to type 'help', 'copyright', 'credits', or 'license'. The user then types '>>> print "Python 2.7.4 on the Mac!"' and the output 'Python 2.7.4 on the Mac!' is displayed. The prompt '>>>' is followed by a cursor.

```
Last login: Sun Apr  7 09:53:24 on ttys002
Michael's-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!
>>> █
```

Figure 23.2: mac install

If you don't currently have version 2.7.6, go ahead and upgrade by downloading the latest [installer](#). Once downloaded, double-click the file to install.

Linux

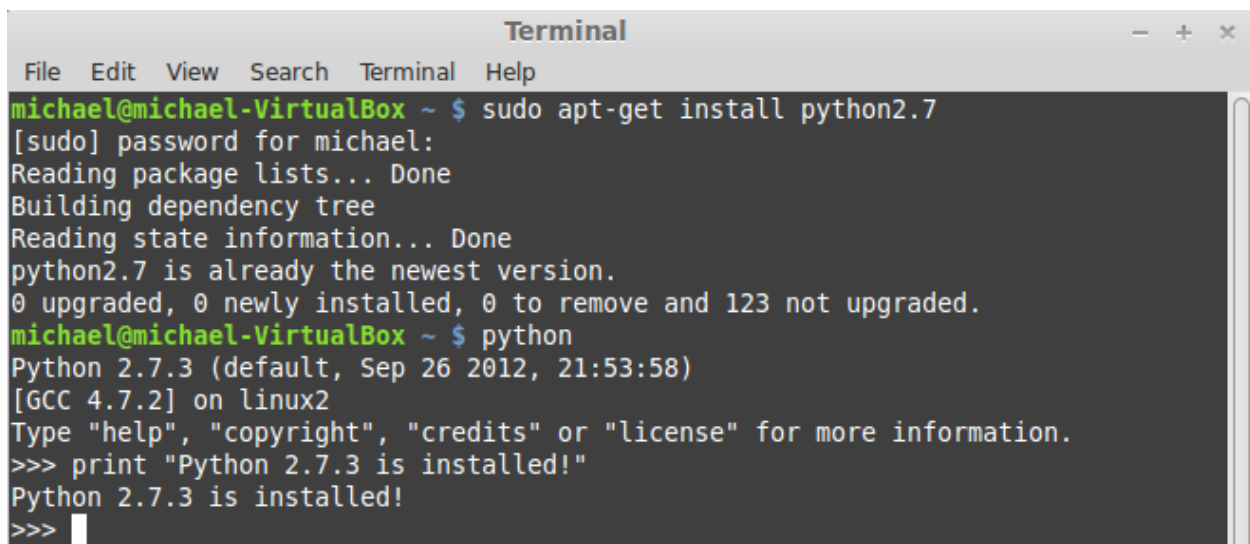
If you are using Ubuntu, Linux Mint, or another Debian-based system, enter the following command in your terminal:

```
1 sudo apt-get install python2.7
```

Or you can download the tarball directly from the official Python [website](#). Once downloaded, run the following commands:

```
1 $ tar -zxvf [mytarball.tar.gz]
2 $ ./configure
3 $ make
4 $ sudo make install
```

Once installed, fire up the terminal and type python to get to the shell:



```
Terminal
File Edit View Search Terminal Help
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!
>>> 
```

Figure 23.3: linux install

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.

Chapter 24

Appendix B: Supplementary Materials

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:

```
1 import ftplib
2
3 server = ''
4 username = ''
5 password = ''
6
7 # Initialize and pass in FTP URL and login credentials (if applicable)
8 ftp = ftplib.FTP(host=server, user=username, passwd=password)
9
10 # Create a list to receive the data
11 data = []
12
13 # Append the directories and files to the list
14 ftp.dir(data.append)
15
16 # Close the connection
17 ftp.quit()
18
```

```

19 # Print out the directories and files, line by line
20 for l in data:
21     print(l)

```

Save the file. *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:

```

1 server = 'ftp.debian.org'
2 username = 'anonymous'
3 password = 'anonymous'

```

Also add the following line just after you create the list:

```

1 # change into the debian directory
2 ftp.cwd('debian')

```

Keep everything else the same, and save the file again. Now you can run it.

If done correctly your output should look like this:

```

1 -rw-rw-r-- 1 1176 1176 1066 Feb 15 09:23 README
2 -rw-rw-r-- 1 1176 1176 1290 Jun 26 2010 README.CD-manufacture
3 -rw-rw-r-- 1 1176 1176 2598 Feb 15 09:23 README.html
4 -rw-r--r-- 1 1176 1176 176315 Mar 03 19:52 README.mirrors.html
5 -rw-r--r-- 1 1176 1176 87695 Mar 03 19:52 README.mirrors.txt
6 drwxr-sr-x 15 1176 1176 4096 Feb 15 09:22 dists
7 drwxr-sr-x 4 1176 1176 4096 Mar 07 01:52 doc
8 drwxr-sr-x 3 1176 1176 4096 Aug 23 2013 indices
9 -rw-r--r-- 1 1176 1176 8920680 Mar 07 03:15 ls-lR.gz
10 drwxr-sr-x 5 1176 1176 4096 Dec 19 2000 pool
11 drwxr-sr-x 4 1176 1176 4096 Nov 17 2008 project
12 drwxr-xr-x 3 1176 1176 4096 Oct 10 2012 tools

```

Next, let's take a look at how to download a file from a FTP server.

Code:

```

1 import ftplib
2 import sys
3
4 server = 'ftp.debian.org'
5 username = 'anonymous'
6 password = 'anonymous'
7
8 # Defines the name of the file for download
9 file_name = sys.argv[1]
10
11 # Initialize and pass in FTP URL and login credentials (if applicable)
12 ftp = ftplib.FTP(host=server, user=username, passwd=password)
13
14 ftp.cwd('debian')
15
16 # Create a local file with the same name as the remote file
17 with open(file_name, "wb") as f:
18
19     # Write the contents of the remote file to the local file
20     ftp.retrbinary("RETR " + file_name, f.write)
21
22 # Closes the connection
23 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:

```

1 $ python ftp_download.py README

```

Finally, let's take a look at how to upload a file to a FTP Server.

Code:

```

1 import ftplib
2 import sys
3
4 server = ''
5 username = ''
6 password = ''
7
8 # Defines the name of the file for upload
9 file_name = sys.argv[1]
10
11 # Initialize and pass in FTP URL and login credentials (if applicable)
12 ftp = ftplib.FTP(host=server, user=username, passwd=password)
13

```

```

14 # Opens the local file to upload
15 with open(file_name, "rb") as f:
16
17     # Write the contents of the local file to the remote file
18     ftp.storbinary("STOR " + file_name, f)
19
20 # Closes the connection
21 ftp.quit()

```

Save the file. *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.ofees.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:

```

1 $ python ftp_upload.py uploadme.txt

```

Check to ensure that the file has been uploaded to the remote directory

NOTE: 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.

Working with SFTP

SFTP (Secure File Transfer Protocol) is used for securely exchanging files over the Internet. From an end-user's perspective 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:

```
1 $ pip install pysftp
```

Now, let's try to list the directory contents from a remote server via SFTP.

Code:

```
1 import pysftp
2
3 server = ''
4 username = ''
5 password = ''
6
7 # Initialize and pass in SFTP URL and login credentials (if applicable)
8 sftp = pysftp.Connection(host=server, username=username, password=password)
9
10 # Get the directory and file listing
11 data = sftp.listdir()
12
13 # Closes the connection
14 sftp.close()
15
16 # Prints out the directories and files, line by line
17 for l in data:
18     print(l)
```

Save the file. 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 do 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".

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:

```

1 import pysftp
2 import sys
3
4 server = ''
5 username = ''
6 password = ''
7
8 # Defines the name of the file for download
9 file_name = sys.argv[1]
10
11 # Initialize and pass in SFTP URL and login credentials (if applicable)
12 sftp = pysftp.Connection(host=server, username=username, password=password)
13
14 # Download the file from the remote server
15 sftp.get(file_name)
16
17 # Closes the connection
18 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:

```

1 $ python test_sftp_download.py remotefile.csv

```

Finally, let's take a look at how to upload a file to an SFTP Server.

Code:

```

1 # ex_appendixB.2c.py - SFTP File Upload
2
3 import pysftp
4 import sys
5
6 server = ''
7 username = ''
8 password = 'p@'
9
10 # Defines the name of the file for upload
11 file_name = sys.argv[1]
12
13 # Initialize and pass in SFTP URL and login credentials (if applicable)
14 sftp = pysftp.Connection(host=server, username=username, password=password)
15
16 # Upload the file to the remote server
17 sftp.put(file_name)
18

```



```
19 # Closes the connection
20 sftp.close()
```

When you run, make sure you specify a file name from your local directory on the command line.

For example:

```
1 $ python test_sftp_download.py remotefile.csv
```

Check to ensure that the file has been uploaded to the remote directory

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:

```
1 # Sending Email via SMTP (part 1)
2
3
4 import smtplib
5 from email.MIMEText import MIMEText
6 from email.MIMEMultipart import MIMEMultipart
7
8 # inputs for from, to, subject and body text
9 fromaddr = raw_input("Sender's email: ")
10 toaddr = raw_input('To: ')
11 sub = raw_input('Subject: ')
12 text = raw_input('Body: ')
13
14 # email account info from where we'll be sending the email from
15 smtp_host = 'smtp.mail.com'
16 smtp_port = '###'
17 user = 'username'
18 password = 'password'
19
20 # parts of the actual email
21 msg = MIMEMultipart()
22 msg['From'] = fromaddr
23 msg['To'] = toaddr
24 msg['Subject'] = sub
25 msg.attach(MIMEText(text))
26
27 # connect to the server
28 server = smtplib.SMTP()
29 server.connect(smtp_host,smtp_port)
30
31 # initiate communication with server
32 server.ehlo()
33
34 # use encryption
35 server.starttls()
36
```

```

37 # login to the server
38 server.login(user, password)
39
40 # send the email
41 server.sendmail(fromaddr, toaddr, msg.as_string())
42
43 # close the connection
44 server.quit()

```

Save. *DO NOT* run.

Since this code is pretty self-explanatory (follow along with the comments), go ahead and update 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:

```

1 # email account info from where we'll be sending the email from
2 smtp_host = 'smtp.gmail.com'
3 smtp_port = 587
4 user = 'hermanmu@gmail.com'
5 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:

```

1 # ex_appendixB.3b.py - Sending Email via SMTP (part 2)
2
3
4 import smtplib
5 from email.MIMEText import MIMEText
6 from email.MIMEMultipart import MIMEMultipart
7
8 def mail(fromaddr, toaddr, sub, text, smtp_host, smtp_port, user, password):
9
10     # parts of the actual email
11     msg = MIMEMultipart()
12     msg['From'] = fromaddr
13     msg['To'] = toaddr
14     msg['Subject'] = sub
15     msg.attach(MIMEText(text))
16
17     # connect to the server
18     server = smtplib.SMTP()
19     server.connect(smtp_host, smtp_port)

```

```

20
21     # initiate communication with server
22     server.ehlo()
23
24     # use encryption
25     server.starttls()
26
27     # login to the server
28     server.login(user, password)
29
30     # send the email
31     server.sendmail(fromaddr, toaddr, msg.as_string())
32
33     server.quit()
34
35 if __name__ == '__main__':
36     fromaddr = 'hermanmu@gmail.com'
37     toaddr = 'hermanmu@gmail.com'
38     subject = 'test'
39     body_text = 'hear me?'
40     smtp_host = 'smtp.gmail.com'
41     smtp_port = '587'
42     user = 'hermanmu@gmail.com'
43     password = "it's a secret - sorry"
44
45     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:

```

1  # Receiving Email via IMAPLIB
2
3
4  import imaplib
5
6  # email account info from where we'll be sending the email from
7  imap_host = 'imap.gmail.com'
8  imap_port = '993'
9  user = 'hermanmu@gmail.com'
10 password = "It's a secret - sorry!"
11
12 # login to the mail server
13 server = imaplib.IMAP4_SSL(imap_host, imap_port)

```

```

14 server.login(user, password)
15
16 # select the inbox
17 status, num = server.select('Inbox')
18
19 #fetch the email and the information you wish to display
20 status, data = server.fetch(num[0], '(BODY[TEXT])')
21
22 # print the results
23 print data[0][1]
24 server.close()
25 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:

```

1 Content-Type: text/plain; charset="us-ascii"
2 MIME-Version: 1.0
3 Content-Transfer-Encoding: 7bit

```

Test Python - What up!

“Test Python - What Up!” is the body of my email.

Also, in the program `num[0]` it 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.