Edinburgh Napier University

SET09103 Advanced Web Technologies

Notes & Workbook 2017-2018

Dr Simon Wells

Overview

Contents

List of Figures

Part I

Labs & Practical Work

Chapter 1

Learning Environment Part #1

Because Linux is one of the most widespread operating systems found on web-servers it makes sense for us to use a form of Linux as our learning environment on this *Advanced* web technologies module. There are a lot of different versions of Linux, known as distributions or distros, and you might have heard or or even used some of them, for example, Debian or Ubuntu. Because we could spend an entire year learning about Linux we shall use a deliberately simplified version of a Linux environment which mimics many of the features found in a full Linux distibution. We shall also ignore many aspects, such as administration of Linux servers, so that we can focus on using Linux within the context of web-development. Our focus will be on the development of web-apps and an in depth investigation of what happens on the server side. That said, whilst we won't focus as much on the client, there is plenty of scope within the courseworks to put into practise your previous learning from the Web Techologies module from last year to make things pretty and provide a better user experience.

There will be a number of core steps involved in getting acquainted with the learning environment:

1. Levinux - Our self-contained, virtualised Linux server which runs within Windows (or on OS X or even under Linux itself)
2. SSH - To enable us to connect from our Windows machine to our virtual Linux machine
3. Basic Linux Usage at the command line
4. Vim - This is our *non-graphical* editor that we shall use to write our source code
5. Git - We shall use Git to store our source code and to "hand in" our coursework assignments

By the end of this section of the module you should be able to start Levinux (our Linux virtual server), log in to Linux using SSH, navigate around the command-line of our linux server, use Vim to edit our files, and use Git to store our files. This constitutes a core set of tools that should allow you to log into almost any Linux server, whether a very small one running on a Raspberry Pi[1] or a very large one, for

[1] `https://www.raspberrypi.org`

example the top 10 fastest supercomputers in the world all run a version of Linux[2], and feel quite at home very quickly. I am not expecting you to achieve all of this in just a single lab session, but you should accomplish this easily within the lab *and* this week's self-study time.

IMPORTANT It is a good habit for you to keep notes whenever you are learning a new tool or environment. I keep a textfile on my computer which also means that I can occasionally copy and paste commands if necessary. This way you won't have to immediately remember how you solved a problem but can look how you did it last time. This makes the whole process much less frustrating. It is how I learned to work with Linux and Git and I still refer to my notes every so often when trying to do something that I rarely do and can't quite remember the syntax for.

1.1 Levinux

Levinux[3] is a tiny Linux virtual server. It has just enough software installed so that you can learn about and get comfortable with a Linux system, but not so many tools that it becomes too daunting. It does however give you a launchpad from which to explore the wider world of Unix-style operating systems. If you get comfortable with Levinux then you shouldn't feel (too) lost trying any other Linux distribution. In fact the best introduction to Levinux and what it can do is probably that from the creator of Linux:

"The micro Linux distribution known as Levinux (download 25 MB) is a tiny virtual Linux server that runs from USB or Dropbox with a double-click (no install or admin rights required) on Macs, Windows or Linux PCsmaking it the perfect learning environment, and way to run & keep your code safe for life! Think of it as an introduction to old-skoolmore relevant now then ever as Linux/Unix gets embedded into everything, with an emphasis on an actual running Python/Flask web app that you can tear apart and do whatever you want with."
Mike Levin `http://mikelev.in/ux/`

Because Levinux is so tiny you can run it easily, without needing to install it, just unzip and click to run. Levinux runs just like any other program on your desktop machine can run from your H: drive, in dropbox, or on a thumb drive. The system can be started by double clicking and you can move the same thumbdrive between different operating systems, e.g. between the Windows machines in the Napier labs and an OS X laptop, and run the exact same Linux environment.

The magic that makes this possible is QEMU[4], emulator software that provides a way to run virtual machines and has been important to the VirtualBox[5] and Android Virtual Device infrastructure[6].

[2] `http://www.top500.org/lists/2015/06/`
[3] `http://www.levinux.com`
[4] `http://wiki.qemu.org/Main_Page`
[5] `https://www.virtualbox.org`
[6] `https://developer.android.com/tools/devices/index.html`

Levinux is basically, a set of scripts and programs that host a Linux install within a very lightweight virtual environment. The scripts manage how Levinux runs and enable it to run seamlessly across different host platforms (like Linux, Mac OS, and Windows). The programs include things like QEMU, mentioned earlier, and various virtual harddrives that are where the Linux system is installed. The specific Linux distribution that is used in Levinux is called 'Tiny Core Linux'[7] which is about 10MB in size and boots into a minimal command line Linux environment.

Getting started with Levinux is fairly straightforward. Download levinux-master.zip from the Levinux website[8]. The whole download should be around 25MB. Copy the zip file to somewhere where you want to store it, for example, a thumbdrive with 512MB free space is a good idea as you can then take your work away with you. Extract Levinux from the zip archive (i.e. right click and the zip archive and select "Extract All...". Don't just double click the archive as this will fail in interesting ways). Open the newly extract directory called levinux-master and double click the file called "PipulateWindows" (NB. If you are on OS X then just double-click "Pipulate.app" or if on Linux the run "PipulateLinux.sh").

When Levinux runs, your operating system might tell you that your firewall has blocked certain features or ask for permission to allow access to certain features. You should grant access when asked. Levinux will now load and you should see something similar to th windows displayed in the following figure:

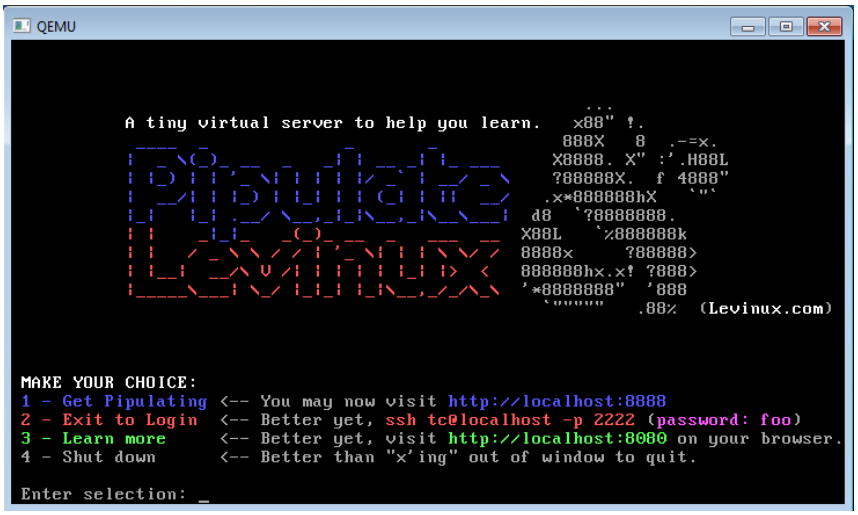

Figure 1.1: Levinux Welcome Screen

If you hit option 2 then you will immediately be able to log in to Levinux using the following credentials:

Username: tc	**Password:** foo

You should now see something like the following figure:

[7] http://tinycorelinux.net/
[8] http://mikelev.in/ux/

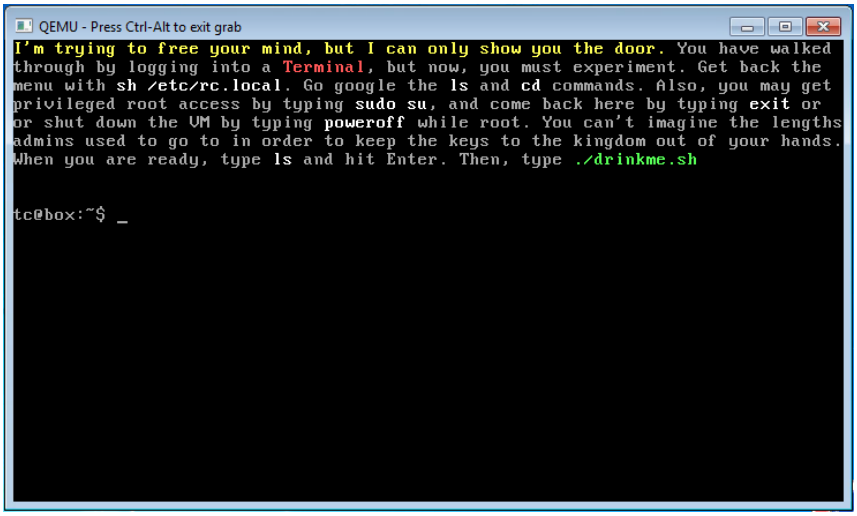

Figure 1.2: The first screen after logging into Levinux

The next thing we need to know about is how to safely shutdown our Levinux installation. Whilst logged in we can use the following command:

```
tc@box:~$ sudo poweroff
```

Which will cause Levinux to shutdown safely. If you are on the screen shown in figure 1.1 then you can just hit '4' then press <enter>. Remember, Levinux is a full computer system, even if it is running within another host system. Just as you wouldn't just hit the power button on your computer to turn it off, or at least you shouldn't, if you do that then stop it now. Instead, you should always follow the correct shut-down procedure for an operating system which gives it an opportunity to tidy itself up and clear up resources that are used. Doing anything else risks corrupting your installation so that it won't work correctly anymore and this is just as true for Levinux as it is for any other operating system.

1.2 SSH

SSH which stands for 'Secure Shell' is a tool for logging in to remote servers. We can use SSH to access Levinux as this provides us with an easy way to log in multiple times, e.g. have multiple shells (command lines) open and logged in to Levinux. The advantage of having multiple shells is that we can edit and save a program in one shell and run the program and monitor output in another shell. As you can imagine, this is very useful when we are developing new web apps on our Levinux server.

IMPORTANT TIP Once Levinux has booted, don't log directly into it. Instead, minimise the Levinux window and always log in using SSH, e.g. PuTTY, instead. This has many advantages but the main one is that you get a scrolling window so that you can see earlier commands or any output bigger than a single window of text. Additionally, PuTTy handles different keyboard layouts much better than Levinux so if you are using a French or German (or other) keyboard then this is the solution you need to get the keys you press to match the output on screen.

If you are on Linux or OS X then you will already have a version of SSH installed and all you need to do is open a terminal and type the following:

```
$ ssh tc@localhost -p 2222
```

then enter the password 'foo' when prompted.

However if you are on Windows then your will need to download an SSH client. The most popular client for Windows is called PuTTY[9]. Download it, put it somewhere safe where you can access it then double click it to run. There is no need to install PuTTY as it is very portable. When PuTTY runs you will be presented with the following screen:

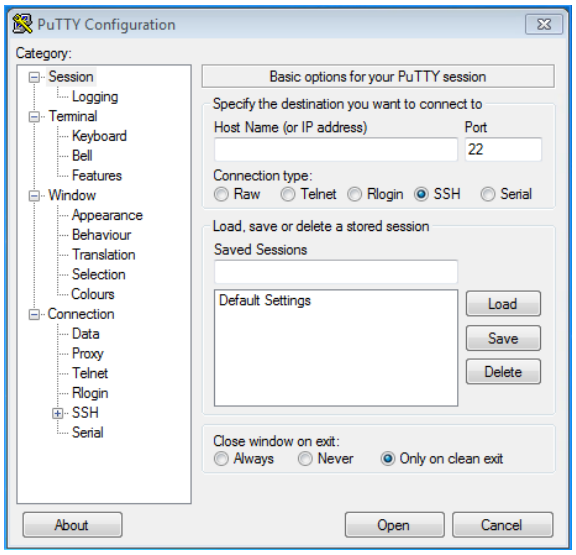

Figure 1.3: The PuTTY Window after you load it

You need to enter some connection details into this window as follows:

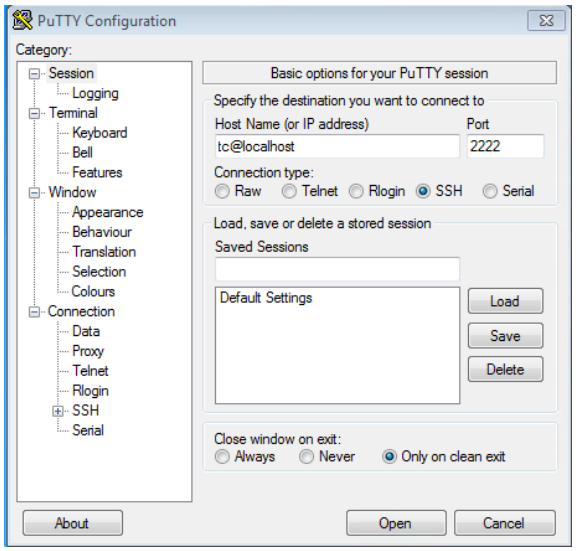

Figure 1.4: The PuTTY Window with completed connection details

[9] `http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe`

You might be presented with the following window:

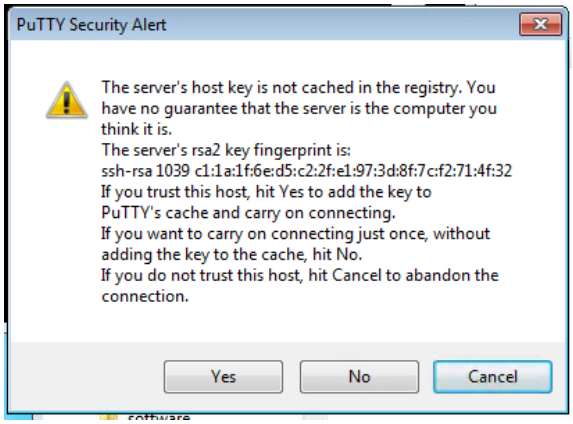

Figure 1.5: The PuTTY alert Window

If so, you can just click the 'Yes' button. If all goes well you should now be presented with a login window for Levinux and all you have to do is type in the password (remember the password is 'foo') and you should get a command line shell on your Levinux server.

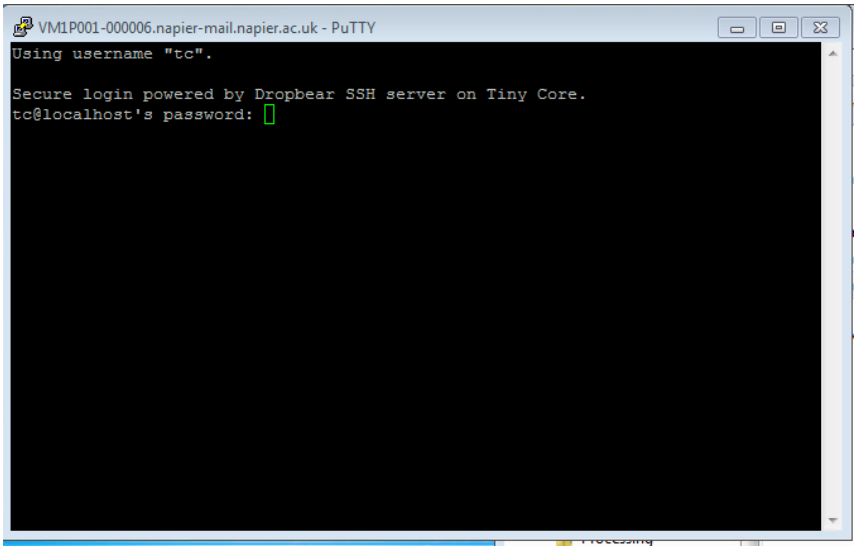

Figure 1.6: The PuTTY login Window

Remember, you can repeat this process with PuTTY as many times as required to get yourself enough shells into the Levinux server to make things easier to work with[10].

IMPORTANT TIP I usually use SSH to log into my Levinux installation at least twice. This gives me a window to edit text in and a window to test my code in. Often I will also have a third window open which I use for command

[10]This is not the best way to work with multiple shells. Another, more powerful way is to use a shell multiplexer, my favourite is one called Screen (`https://www.gnu.org/software/screen/`) but that is yet another tool to learn, and another set of commands, so lets try to keep things *reasonably* simple

line interaction, Git commits, and testing API calls using cURL, but two is the minimum for a nice development setup - Remember, this is your development environment so you might as well take some time to set it up nicely and so that you can work more productively.

1.3 Basic Linux Usage

When we first log in to Linux we will see a prompt, a place where we can type commands. It looks something like this:

```
tc@box:~$
```

This simply means that user 'tc' (short for tinycore, the default user that you logged in as) is logged into the machine called 'box'. The ':' is merely a separate between the user@machine part of the prompt and the next part '~' which is your current location in the filesystem. The tilde or '~' symbol is used on Linux machines to mean your home directory.

Linux has a file system, a way for all of the files that make up the running system to be organised hierarchically, just like Windows. However, on Linux the file system is organised a little differently. Instead of starting at 'C:' the Linux file systems starts with '/' which is also known as the "root" of the filesystem. The word root is used because the Linux filesystem is shaped like a tree. All of the resources that you can access, such as your own files, are located at some level somewhere within the tree. An advantage of the Linux approach is that, when you add extra hard-drives, or mount network resources, instead of extra drive letters, all of your resources get mounted within the tree, e.g. /Volumes/Web might be the path to a remove web server and /media/cdrom might be the path to your CD-Rom drive. But other than that, from a basic user-oriented perspective, both Windows and Linux file systems are a hierarchical collection of files and directories in which any given directory might contain zero or more files or child directories.

When you log in you will be located in your own directory, called your "home" directory which is located in the filesystem tree at

```
/home/tc
```

You can see where you are in the filesystem by typing

```
$ pwd
```

which is short for print working directory or tell me where I am. You can type this anywhere you have a prompt. Also, no matter where you navigate to within the file system you can always return to your home directory by typing:

```
$ cd
```

You can navigate around the file system using the cd command which is short for 'change directory'[11]. The default version, without an argument takes you home, as we said before, but if you supply an argument then you can change the current directory. Let's try that now by changing to the root of the filesystem:

```
$ cd /
```

If you now use pwd you should see that the output is different to what it was before. You are no longer in your home directory but are in the root directory instead. Now ues the cd command without an argument to go home and use pwd to see where you are again. You can also step up through the directory hierarchy by using the '..' argument:

```
$ cd ..
```

'..' is an *alias*, a label that has a default meaning, which in this case means "move into the parent directory". Explore the filesystem for a while using the cd, cd /, cd .., and pwd commands.

There is a limit to what we can do with just these commands because we can only move into our home directory, or else navigate up through the tree to the file system root directory. We need a couple more commands to let us see what is inside a directory and to move into a new directory. For this we use the 'ls' command which is short for list or list contents to see what files or child directories are in our current directory, and the cd command but with the name of a child directory as the argument. So, we can use 'ls' as follows:

```
$ ls
```

to list the files in the current directory. If you try this now you will see that your home directory contains a few default files and folders e.g.

```
tc@box:~$ ls
Pipulate.sh  Recipe.sh    drinkme.sh   htdocs/
```

Now we can create new files in our home directory quite easily using touch, e.g.

```
$ touch testfile.txt
```

which should give us something like this:

```
tc@box:~$ ls
Pipulate.sh   Recipe.sh     drinkme.sh    htdocs/       testfile.txt
```

We can also create new directories using the 'mkdir' which means make directory

```
$ mkdir testdirectory
```

[11] You'll notice that many Linux commands are shortened versions of longer words. This is partly designed to reduce the amount of typing that you do. It may seem silly now but when you are changing directory hundreds of times a day it is much nicer to type cd than change-directory each time.

which results in

```
tc@box:~$ ls
Pipulate.sh    drinkme.sh     testdirectory/
Recipe.sh      htdocs/        testfile.txt
```

We can also move files around using the 'cp' and 'mv' commands which are short for copy and move respectively. Let's see them in action; first we will make a copy of testfile.txt then move the copy into the testdirectory:

```
tc@box:~$ cp testfile.txt testfile2.txt
tc@box:~$ ls
Pipulate.sh    drinkme.sh     testdirectory/ testfile2.txt
Recipe.sh      htdocs/        testfile.txt
tc@box:~$ mv testfile2.txt testdirectory/
tc@box:~$ ls
Pipulate.sh    drinkme.sh     testdirectory/
Recipe.sh      htdocs/        testfile.txt
tc@box:~$ ls testdirectory/
testfile2.txt
```

Notice that in this example we passed an argument, the name of a directory 'testdirectory' to the ls command and this caused the contents of testdirectory to be listed instead of the current directory. We can do that with any directory that we have access to.

Finally we might want to delete files to keep things tidy. We can use the 'rm' command to remove files, e.g.

```
tc@box:~$ rm testfile.txt
rm: remove 'testfile.txt'? y
tc@box:~$ ls
Pipulate.sh    Recipe.sh      drinkme.sh     htdocs/        testdirectory/
```

We can use also use rm to remove directories, however, by default we get this behaviour:

```
tc@box:~$ rm testdirectory/
rm: 'testdirectory' is a directory
```

What we need to do instead is to supply some options to the rm command. We need rm to act recursively, that means move into the specified directory and delete its contents and we also need to force rm not to stop and ask us for each file whether it should be deleted. We there fore need to use rm as follows:

```
tc@box:~$ ls
Pipulate.sh    Recipe.sh      drinkme.sh     htdocs/        testdirectory/
tc@box:~$ rm -rf testdirectory/
tc@box:~$ ls
Pipulate.sh  Recipe.sh    drinkme.sh   htdocs/
```

Of all the commands we have met so far, rm is the only one that can do any real damage. rm -rf could conceivably delete all of your files and directories if the command is executed in the wrong place. As a rule it is probably worth only making changes within your home directory, e.g. /home/tc, whilst working on exercises in this module, that way you are less likely to destroy your Levinux install accidentally and have to start again. However because it is easy to set up and run a new Levinux

instance you can afford to experiment because if you destroy your Levniux instance you can always start again. You should however, especially once you start writing code in Levinux, keep backups of anything that you will need to use again such as the code for your courseworks.

There are many, many more commands than just these. In fact you can do much more with the command line than you can with graphical tools. However, this should be enough for you to get started and is enough for you to be able to create and delete files and directories, to navigate the file system hierarchy, to list the contents of directories, and to view the contents of files.

To do more exploration of Linux, you can of course experiment with Levinux. Another good place to start is the Linux Zoo site[12] which offere online virtual machines, more information, and a number of tutorials. Particularly the Linux Zoo "essential Linux" pages[13] which explain in more depth some of the tools that we have already covered plus many many more.

1.4 Vim

Vim is a command-line based text editor. It is based on an earlier editor called Vi (Vim is VI improved, hence Vim, which is a little easier to use). You might ask why we don't just use a GUI text editor like Notepad, or the editor in Visual Studio. The main reason is because we are talking about advanced web technologies, and dealing with them often involves accessing remote servers. Furthermore, the majority of servers do not have a graphical interface and only have installed the most minimal and robust set of tools. The one editor that you are almost guaranteed to find on any Unix or Linux server is Vi and by learning Vim you are well placed to handle Vi. As a result it makes sense for us to become familar with it. Many of us actually find that the shortcuts and powerful control that Vim offers us mean that we just concentrate on learning one editor very well and only use that editor rather than moving to a different editor each time we need to write a different type of document or program in a different language.

What makes Vim different from many of the editors that you will be familar with, like Notepad, is that there is no role for the mouse, no buttons to click at all, only keyboard shortcuts, so we shall learn just enough of those in this module to be able to edit basic documents[14]. A second very important aspect of Vim is that it is a *modal* editor. When you use Vim you use it in different modes, when you are typing content into a file then you are in edit mode, and when you are entering commands you are in command mode. When Vim starts it is in command mode and anything you type will be interpreted as a command for Vim to perform. You switch into edit mode by typing 'i' (for insert) and you can return to command mode at any time merely be hitting the escape key.

[12] `http://www.linuxzoo.net`

[13] `http://linuxzoo.net/page/intro.html`

[14] However there are hundreds of Vim commands and a really clever things is that most of the commands can be chained together so that you can automate many editing tasks.

IMPORTANT If you are ever unsure what mode you are in the you can just hit escape a couple of times to ensure you are in edit mode. From here you can just type 'i' again to enter the edit mode.

Start Vim by typing Vim at the prompt:

```
tc@box:~$ vim
```

and you will see something similar to the following:

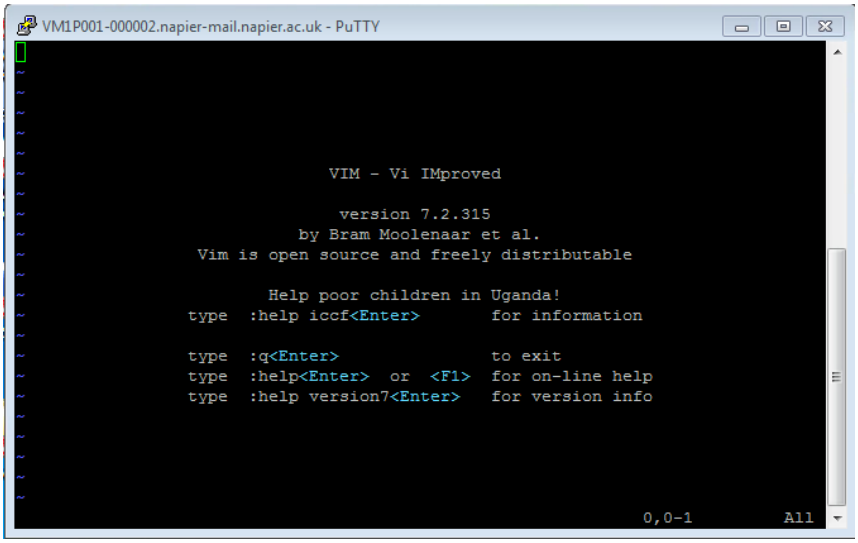

Figure 1.7: The default Vim Editor window

Let's do the easiest thing first. Let's quit Vim. To do this we enter the command mode by hitting escape then enter ':q' (where is q is short for quit) and hit enter, e.g.

```
<ESC>:q<ENTER>
```

Your Vim window should look something like this when the command is entered (but before you press enter):

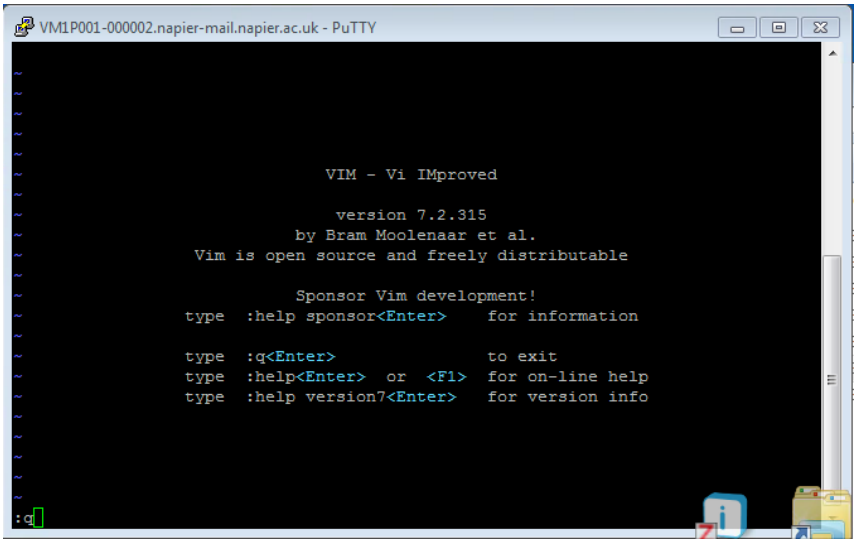

Figure 1.8: The Vim Editor with ':q' command entered

Once you hit enter you will be dumped back to the Linux prompt. Two other useful commands to be aware of for quitting are as follows:

- To quit and discard any changes, i.e. if you have already made changes to a document that you don't want to save:

```
<ESC>:q!<ENTER>
```

- To quit and save changes, we use :wq for (w)rite and (q)uit:

```
<ESC>:wq<ENTER>
```

Let's start Vim again and actually edit some text. This time when Vim starts you need to press 'i' for (i)nsert to enter the edit mode. You can now type away to your hearts content. When you are ready to save the file you can enter the command mode and type :w for (w)rite. If the file doesn't have a filename you will get a message to that effect so, with a new file that isn't yet saved you can us :w filenname.txt (where filename.txt is the name of the file that you want to create). This file will then be created in whichever directory you were in when you started Vim.

There is **A LOT** to learn in Vim. The easiest way to do that is to just use it. There are many online resources that teach you how to use Vim but two of my favourites are:

1. The Open Vim Tutorial: `http://www.openvim.com/`

2. Vim Adventures: `http://vim-adventures.com/`

There is also a cribsheet of useful commands in section A.2.

1.5 Git

Git is a source control system that enables you to keep track of your source code, its history and any changes you make. Git can be used to track any file but is most efficient and best suited when used only with textual files. Because Git is a *distributed* source control system it works very well to enable groups of people to work on the same source code as well as supporting experimenting with your code, trying out lots of different ideas in separate *branches* (which are a bit like a copy of your code but with tools to help manage that copy and support re-integrating it with your main source tree if you want to), and being able to roll back to an earlier version if you decide you have take a wrong turn.

Whilst you might have seen Git before, perhaps as a plugin to an IDE or as a standalone GUI app, Git is primarily a command line application, so we shall use it to get our source code in and out of Levinux.

IMPORTANT We shall use Git to support the hand-in of courseworks in this module so you should get familiar with it as soon as possible. A good place to start is by dipping into the Git SCM book[15].

There are also numerous interactive tutorials and resources to help you get started with Git:

1. Github's Learn Git 15 minute tutorial: `https://try.github.io/levels/1/challenges/1`
2. Learn Git Branching `http://pcottle.github.io/learnGitBranching/`
3. Git Immersion `http://gitimmersion.com/lab_01.html`

You should also create either a Github account[16] or a Bitbucket account[17] (or both if you like) then create a repository within your new account called 'set09103'. You will push all of your code throughout the module into this repository and at hand in time I will pull a copy for marking. The advantage of this appraoch is that at any point, if you need help with your code, then we have a copy that I can see remotely. However this only works if you keep adding your code to your repository. That means whenever you make changes you need to (1) add them, (2) commit them with an explanatory message, and (3) push the changes from your local repository to the shared one on Github or Bitbucket.

Once you have gone through the steps in section **??** you will already have Git installed in Levinux. In order to use it we have to do a couple of things. Log into Levinux then do the following (obviously replace my email address and name with your own):

```
$ git config --global user.email "s.wells@napier.ac.uk"
$ git config --global user.name "Simon Wells"
```

It is easiest to create a new repository through the interface on Github or Btbucket then use the repository cloning address to make a local copy as this sets up all the

addresses automatically for pushing and pulling code. Once a repository is set up and you have cloning address you can do something similar to the following (where `https://siwells@github.com/siwells/sandpit.git` is the name of one of my repositories on github, your's will obviously be different):

```
$ git clone https://siwells@github.com/siwells/sandpit.git
```

We can then make changes within the new local clone of, in this case the 'sandpit' repository, then add, commit, and push as follows (again the details of adding and committing depend upon the exact files that you have altered. In this case we'll assume that a file test.txt has been edited):

```
$ git add .
$ git commit -m "Added introductory example"
$ git push
```

Again, as for Linux and Vim, there are many options and powerful features that you can learn to use with Git, however we shall try to keep things as simple as possible for now.

> **ADVANCED** If you are comfortable with using Git and SSH then you might want to try setting up SSH keys so that you don't have to use a password each time you push code into your remote repository.

1.6 Wrapping Up

Obviously this has only been the most basic of introductions to web development using a Linux server. There is much much more that you could learn about any one of the tools that we have introduced and it is well worth your time to explore additional resources and reading for each of them. It is very likely that you will experience some or even all of these technologies, in some form, at some point of your career and putting in some extra effort now will mean that you are much more capable later.

Chapter 2

Learning Environment Part #2

After last weeks extravaganza of new tools, we will only introduce two new tools this week, the Python language and a Python Library for developing web applications called Flask. By the end of this weeks practical work you should be able to build a simple 'Hello Napier' web app using Python and Python-Flask which is hosted on and runs within our simple Levinux Virtual Linux environment and whose source code is pushed to our personal Github (or Bitbucket) repositories.

> **VERY IMPORTANT** The work this week builds directly on last week so if you haven't finished working through chapter 1 then it is best to do that first or you will probably get horribly stuck.

2.1 Python

Python[1] is a very useful programming language and much of its popularity stems from the fact that it is easy to get a lot done with having to write too much code. We already have Python installed in Levinux and ready to use. We can run Python by typing its name in the terminal, e.g.

```
tc@box:~$ python
Python 2.7.10 (default, May 25 2015, 09:55:35)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This starts the Python Read-Evaluate-Print-Loop or REPL in which we can type Python commands and get immediate output. To exit the REPL we type 'quit()' which will return us to the Linux shell, e.g.

```
tc@box:~$ python
Python 2.7.10 (default, May 25 2015, 09:55:35)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
tc@box:~$
```

Here is the traditional 'Hello Napier' program in Python (try it out for yourself):

[1] https://www.python.org/

```
tc@box:~$ python
Python 2.7.10 (default, May 25 2015, 09:55:35)
[GCC 4.9.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello Napier"
Hello Napier
>>> quit()
tc@box:~$
```

The REPL is very useful for trying out bits of Python as you are learning and also for interactively developing code and analysing data. There are super-charged REPLs and related tools such as iPython[2] and iPython notebooks[3] which are used in data science and data analysis. However, once we are comfortable with the Python syntax, we will mostly want to run Python scripts, so let's do that.

Create a new file called first.py, open it in Vim then edit it as follows:

```
1 print "Hello Napier"
```

Save and close first.py then execute the following command in the shell:

```
tc@box:~/src$ python first.py
Hello Napier
tc@box:~/src$
```

Congratulations. Another first. You just ran your very first Python script. All we will really do from now on in the remainder of this module is build on top of this basic script in order to build our own web-apps. However, we shall have some help along the way from some really excellent libraries so don't worry, we shan't build everything from scratch.

Now the best thing to do is to learn some Python syntax by exploring som of the excellent online 'learn Python' tutorials that are available. Some of them are interactive, so you can type directly into the web-site and get results. Others just give you exercises to do yourself and you can do those exercises in Python on our Levinux platform. Here are a few, in order of usefulness according to me, but many more are just a Google search away:

1. Learn Python the hard way: `http://learnpythonthehardway.org/book/`
2. The Python Practice Book: `http://anandology.com/python-practice-book/index.html`
3. A Byte of Python: `http://www.swaroopch.com/notes/python/`
4. Code Combat: `https://codecombat.com/`
5. Python for you and me: `http://pymbook.readthedocs.org/en/latest/`
6. The Hitchhiker's Guide to Python: `http://docs.python-guide.org/en/latest/#the-hitchhiker-s-guide-to-python`

[2] `http://ipython.org/`
[3] `http://ipython.org/notebook.html`

7. Hands-On Python Tutorial: `http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html`
8. The Python Challenge: `http://www.pythonchallenge.com/` - Some quite touch challenges that you can solve using Python (NB. Assumes you already know what you are doing)
9. Python Tutor: `http://www.pythontutor.com/` - Helps visualise the execution of Python code. Again, assumes that you have some prior knowledge of Python syntax.

The links towards the top of the list are aimed at those completely new to the Python language whereas those further down the list will help those who have some Python knowledge already. Another good way to practice a new language or to improve your existing abilities, not just in Python, but in any language, is to try to solve a set of problems using the language. I often use Project Euler when starting out with a new language but there are also many others:

1. Project Euler: `https://projecteuler.net/`
2. Stack Exchange Code Golf: `http://codegolf.stackexchange.com/`
3. Code kata: `http://codekata.com/`
4. Reddit Daily Programmer: `https://www.reddit.com/r/dailyprogrammer`
5. Programming Praxis: `http://programmingpraxis.com/`
6. Rosetta Code: `http://rosettacode.org/wiki/Main_Page`
7. International Collegiate Programming Contest Problems Index: `http://acm.hit.edu.cn/judge/ProblemIndex.php`
8. Algorithmist: `http://www.algorithmist.com/index.php/Main_Page`

It is also worth creating a git repository of any code that you create when learning a new language because if you take a break from that language to do something else, you will have somethings to remind you of where you were up to before. It is also a neat way to share your solutions with others. If you exhaust all that lot then another great practice approach is to try to implement your own versions of basic (and advanced) algorithms and data structures. For example, if you really want to challenge yourself, and your knowledge of a programming language, read about and implement probabilistic datastructures like Bloom Filters which I find quite interesting, or Fountain Codes which are used in tools like Bittorrent.

2.2 Python-Flask

Python-Flask, or just Flask[4], is a Python based microframework for developing websites in Python. If you have run the drinkme.sh and Pipulate.sh scripts from Chapter 1 then Flask will already be installed and ready to run. We shall leave all of the nitty gritty of installing and administrating software on a Linux server as an exercise that is outside of the remit of this module and just focus on using the tools now that we have them.

[4] `http://flask.pocoo.org/`

Flask includes a lot of useful tools; a development server and debugging tools so that we can run our flask apps during development without needing to run a separate external server like Apache[5] or NGinX[6]. Obviously if we were deploying our apps in the real world then we would host our web-apps differently. We would use a secure Linux install on a server with access to the external world and would use a good HTTP server, like NGinX, and a WSGI-compliant app-server, like uWSGI[7]. We might even use a load balancer like Haproxy[8] to enable us to scale our operations. We will look at all of these aspects later in the module but for now we shall concentrate on the development server and a simplified development environment. Flask also has integrated unit-testing support, RESTful request dispatching, templating, to help generate HTML pages, using the Jinja2 tool.

Flask supports us in building web-apps that conform to the Python Web Server Gateway Interface (WSGI)[9]. The goal of WSGI is to make it easy to develop and deploy Python web-app, regardless of the library that is used. So developers can find a library that they find useful, or helps them to tackled their development task, but the output of the process, the web-app has know features and capabilities which means that any server that supports the WSGI can run the web-app. This is similar to what happened with web-app development in Java, where the servlet API means that many tools can be used to build a Java web-app and many web-servers can then run the app.

IMPORTANT The version of Levinux that we have used from the `http://levinux.com/` site does not forward port 5000. This is the port that we will access in our browser to see our web-apps. We have two solutions, the first is to download a new version of Levinux (Napier Edition) from the SET09103 Moodle page and set up a fresh environment (you will have to run the pipulate and drinkme scripts again. The alternative, which might wreck your version of Levinux and mean you have to start again anyway is to edit two of the startup scripts within Levinux. To ensure that your installation stays completely portable there are scripts to edit. The first is here Pipulate.app/Contents/MacOS/qemu32.bat and you need to add the following line between the similar lines for higher and lower port numbers:

```
1    -redir tcp:5000::5000 ^
```

Similarly we need to edit the Pipulate.app/Contents/Resources/qemuonmac.sh file to add the following line:

```
1    -redir tcp:5000::5000 \
```

Ensure that both edits are exactly as shown. If you have done it wrong then worst case scenario is that your Levinux install won't start. The othe likely failure is that Levinux will start but the port won't forward properly.

[5] `http://www.apache.org/`

[6] `https://www.nginx.com/resources/wiki/`

[7] `https://uwsgi-docs.readthedocs.org/en/latest/`

[8] `http://www.haproxy.org/`

[9] The WSGI standard is described in this Python Pep:`https://www.python.org/dev/peps/pep-0333/`. A PEP is a Python Enhancement Proposal and is the main way that the planned development of the Python language progresses.

2.3 Python Flask “Hello Napier”

Here is our first Python Flask app:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello World!"
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0')
```

IMPORTANT Don’t just copy and paste code from the PDF. This is for two very good reasons. The first is that Python is ‘white-space sensitive’. This means that Python uses whitespace as part of the layout of code, instead of using things like curly braces such as ‘{’ and ‘}’. So if you get spaces, tabs, and other whitespace characters mixed up in your file, which can easily happen if you copy and paste, then it will affect the indentation of the file (although perhaps not in a way that you can tell visibly because whitespace can all look pretty much the same). The second very good reason is that typing in the code counts as ‘deliberate practise’. You are more likely to remember stuff if you do it multiple times and typing it in the first time counts as the first opportunity to practise.

Type in the ‘Hello Napier’ code from above or get it from the module’s Git repository. Save the code in a file called hello.py then run it by executing the following command in the same directory where hello.py is stored:

```
$ python hello.py
```

This command causes our web-app to be run using the Flask development server which is really useful and fast for debugging during development. If everything goes well then you should see output similar to the following in your terminal:

```
tc@box:~$ python src/hello.py
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Congratulations. You now have your first Python web-app running. You can visit the page generated by Flask in a web browser. Just start up a browser in your host OS, e.g. in Windows when running Levinux in the lab. Although the output from Flask in your terminal tell you which address (and port) your app is running at ’0.0.0.0’ doesn’t appear to reliably resolve on Windows so it is best to navigate to `localhost:5000`:

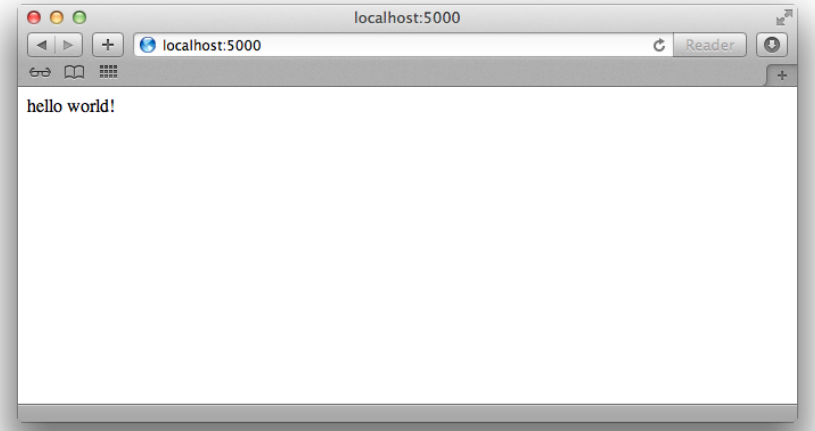

Figure 2.1: Your first Python Flask web app.

2.4 Wrapping Up

Right. Thats all the preliminaries in place for building web-apps in our learning environment. We can now run a small and lightweight, but otherwise similar to full Linux installs, virtualised server which will host our web-apps. We can log in to the server using SSH, navigate the Linux environment, and edit files using VIM. We can also take advantage of the installed Python language and Python-Flask web-app library to build our own web-apps.

CHALLENGE Create a Github or Bitbucket account. Bitbucket allows you to create an unlimited number of private repositories if you use your @napier address to register whereas Github restricts the number of private repos. That said, either site allows you to create as manay public repositories as you like. Create a remote repo in your Github or Bitbucket account called 'set09103'. Run Levinux and log in then clone your set09103 repo. Create a new file in your repo called 'hello.txt' and put the message "Hello Simon" in it then add, commit and push your changes to your remote repo. Once you have done this, check on either Github or Bitbucket to ensure that your text file is in the repo where you expect it to be then send an email to Simon containing the name of your account, which service, Github or Bitbucket, you are using, and the clone url for your set09103 repository. *If you use a private repository then you will have to add my account as a collaborator. On both Github and Bitbucket my account name is 'siwells'*. I will then pull everyone's repos to ensure that we are all at the same place in our learning. We will then use the set09103 repos as the place to store our coursework projects and for the hand-in. This just means that we have everything in place early before the hand-in deadline.

Over the next few chapters we will look at a whole host of things we can do with Flask. However you should also rest assured that all of the tools we are learning will prove useful to you at some point in your career and will help you to become the best developers that you can be.

Chapter 3

Python Flask: Debug Mode, Errors, Routing, & Static Files

The last two weeks have got us to the point where we can build a simple 'Hello Napier' web app using Python and Python-Flask hosted within our simple Levinux Virtual Linux environment and whose source code is pushed to our personal Github (or Bitbucket) repositories. We can now start to expand our web-app skills using Python-Flask. We'll start by looking at the debug mode which enables us to tell the Flask development server to do hot reloads of updated code whenever we save our edited python web-app file. This saves us lots of time as we begin to make lots of changes to our web-app. Additionally, we should expect to to see lots of errors. After all we are working with lots of new tools. So we shall look at Flask errors and how to deal with them. Then we will look at routing requests from web browsers to different URLs, basically how to build a tree of web addresses that each fire off a different Python function when the browser tries to access them. Finally we will look at serving up static files, like image files from the file system.

VERY IMPORTANT Chapters for this and subsequent weeks build directly on all of the skills learned in the first two chapter, particularly Levinux, Linux, SSH, and Vim from chapter 1 and Python and Python Flask from 2.

To some degree you can mix and match this and subsequent chapters and subsections to meet your needs, for example, the type of web app you want to build, but you should aim to cover all chapters by the end of the semester.

3.1 Flask Debug Mode

Recall that our 'Hello Napier' app looks like this:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello World!"
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0')
```

When we run this Python flask app using the python command in the terminal, e.g.

```
tc@box:~$ python src/hello.py
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
```

Python calls the run function of the app object and executes it. By default run() takes no arguments but we have used the *host='0.0.0.0'* argument to tell flask to allow connections from outside of the local machine, this is what enables us to access the web-app from Windows as by default we would only be able to access the web-app from within Levinux. However both the app object and the run function have a number of other features that we can use. Whilst developing a new web-app one of the most important is the debug mode which we can run by adding debug=True to our call of the run function, e.g.

```
1    app.run(host='0.0.0.0', debug=True)
```

Two important features of the debug mode are

1. Causing the development server to automatically restart each time we change our code, e.g. each time we save (*:w <ENTER>*) our file after editing it in Vim.

2. Printing out debug information and a Python stack trace in the browser so that we can work out what went wrong.

If we try to access a route that causes a Python error then instead of this

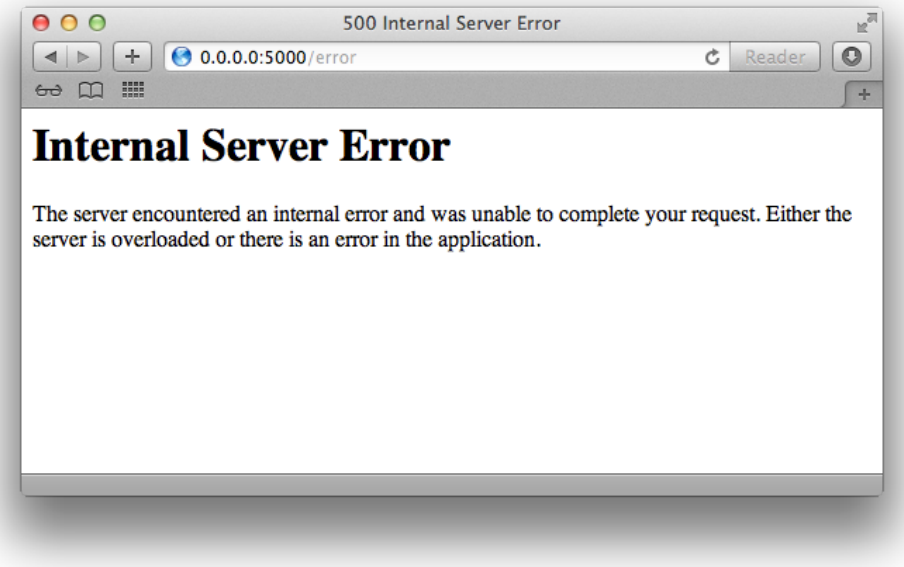

Figure 3.1: Flask internal server error

We will get a Python stack trace displayed in the browser (and printed on the output in the terminal where you ran the web-app). Here is an example but remember that the stack trace will differ greatly depending upon the type of error and the details of what went wonge so this is merely indicative:

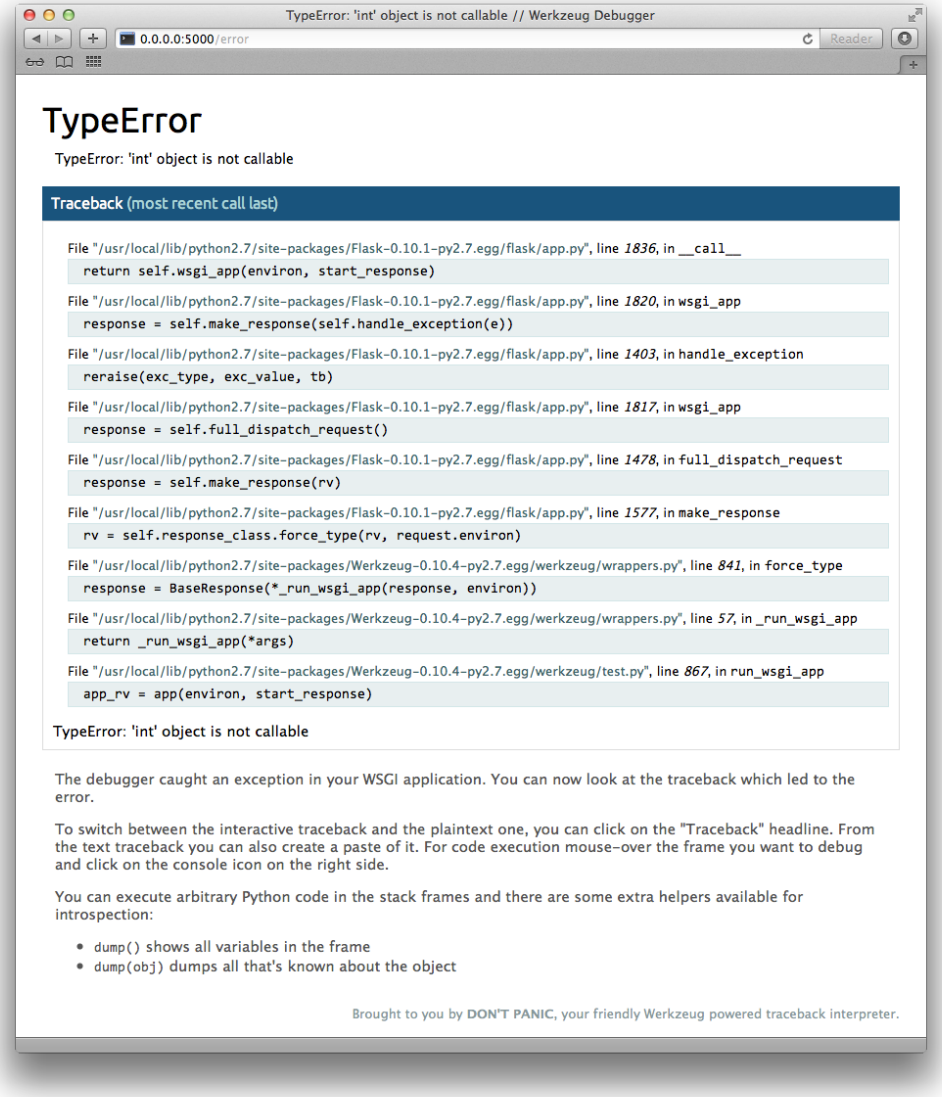

Figure 3.2: An error stack trace example

IMPORTANT If you ever run your web-app on a publically accessible server then you must turn debug mode off as it presents a significant security risk and can allow the execution of arbitrary code.

3.2 Flask Routing

Routing is what enables us to build sensible URLs and addresses for the pages of our web-app. You should consider the design of the address hierarchy for your web-app, the API, to be at least as important a consideration as the design of any content of your actual web-apps pages.

In flask, web addresses or URLs are called routes. To add more routes to your web-app you use the app.route decorator. You just write a new Python function then add a decorator for each one to make the function into a route. For example, in the following web-app we have 3 routes:

```
1  from flask import Flask, redirect, url_for, abort
2  app = Flask(__name__)
3
4  @app.route("/")
5  def root():
6    return "The default, 'root' route"
7
8  @app.route("/hello/")
9  def hello():
10   return "Hello Napier!!! :D"
11
12 @app.route("/goodbye/")
13 def goodbye():
14   return "Goodbye cruel world :("
15
16 if __name__ == "__main__":
17   app.run(host='0.0.0.0', debug=True)
```

3.3 Flask Redirects & Errors

You can redirect a user from one URL endpoint to another quite easily by using the redirect() function, for example, if we were building an app that required a user to be logged and we detected that the user wasn't logged in then we could redirect the user to a login page instead of the page that they requested, e.g.

```
1  from flask import Flask, redirect, url_for
2  app = Flask(__name__)
3
4  @app.route("/private")
5  def private():
6    # Test for user logged in failed
7    # so redirect to login URL
8    return redirect(url_for('login'))
9
10 @app.route('/login')
11 def login():
12   return "Now we would get username & password"
13
14 if __name__ == "__main__":
15   app.run(host='0.0.0.0', debug=True)
```

Note that this isn't an entire web-app, just the imports and indicative code for how a redirect could work. Obviously, to complete the scenario above we would also need code to check whether the request came from a logged in user and the login page would also need to accept and check any supplied credentials (but we will look at that type of functionality in subsequent chapters).

If we try to access a page that doesn't exist then we get a default error 404 Not Found status page like the following:

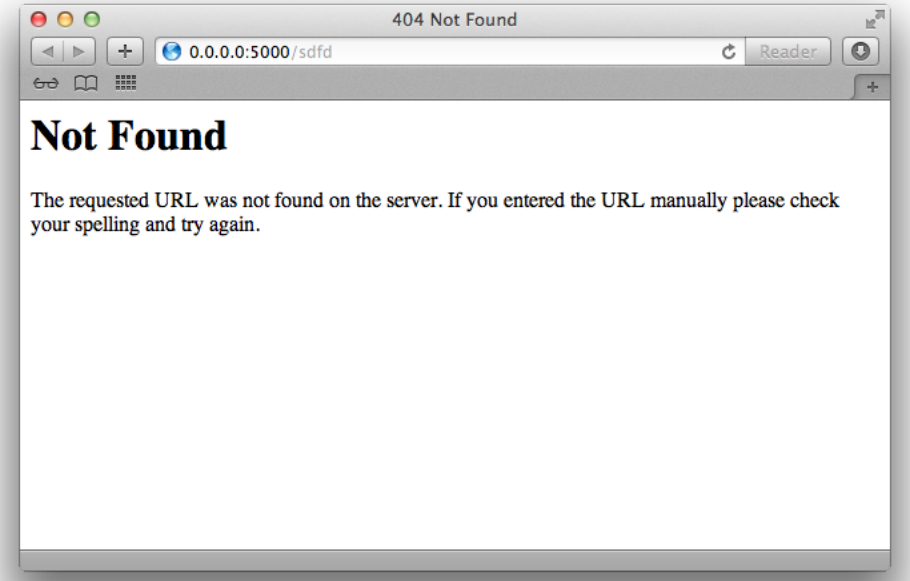

Figure 3.3: The default 404 Not Found page

However we can also provide our own error pages that better fit in with the design of our app or tell our user how to recover from the error. For example, you could provide a link to the root page or login page instead. Providing a custom HTTP error page is straightforward. Instead of the *@app.route* decrator we use the *app.errorhandler* decorator and pass it the code that we want to handle. We also add the code after the return message so that the code is returned to the browser.

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route("/")
5  def hello():
6    return "Hello Napier"
7
8  @app.errorhandler(404)
9  def page_not_found(error):
10   return "Couldn't find the page you requested.", 404
11
12 if __name__ == "__main__":
13   app.run(host='0.0.0.0', debug=True)
```

Now if we visit our web-app but use an address that doesn't exist (just make something up after the http://localhost:5000/ bit) we should see something like this:

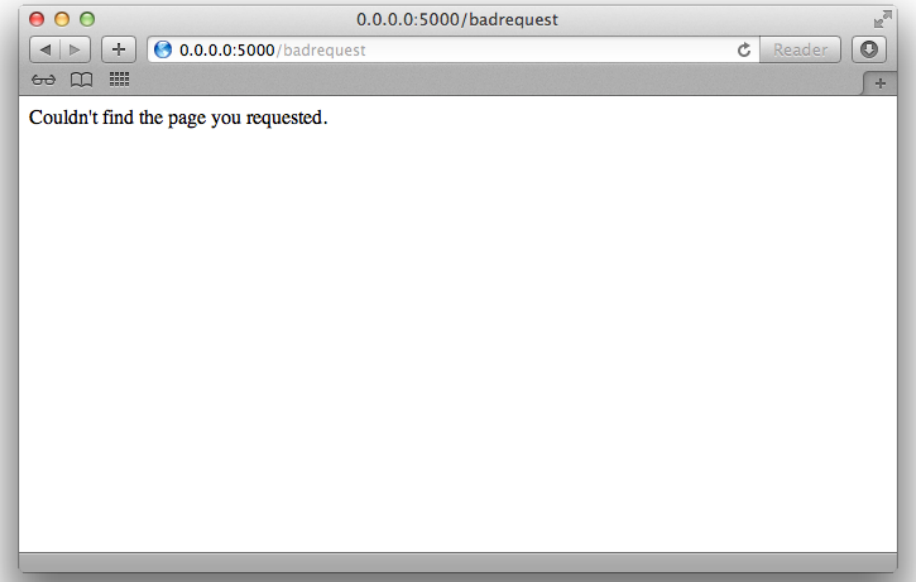

Figure 3.4: The default 404 Not Found page

Obviously this is not much better than the default but it does mean that we can start to build custom errors for our web-app, and once we start to look at returning HTML pages, using templates, and adding style with CSS then custom errors really become useful.

You should also be aware that there are many error codes that your web-app might conceivably respond to. Some, badly designed sites will only respond with a 404 regardless of the error that occured, but the range of error codes actually means that you can respond with appropriate information that enables your user to make an informed decision about what to do next. Try implementing some other error codes. Because it can be difficult to force these errors to occur there is a shortcut in Flask that allows us to inject a specific type of error at any point which can be useful for testing.

```
1  from flask import Flask, abort
2  app = Flask(__name__)
3
4  @app.route("/")
5  def hello():
6    return "Hello Napier"
7
8  @app.route('/force404')
9  def force404():
10   abort(404)
11
12 @app.errorhandler(404)
13 def page_not_found(error):
14   return "Couldn't find the page you requested.", 404
15
16 if __name__ == "__main__":
17   app.run(host='0.0.0.0', debug=True)
```

Note the line that contains "abort(404)". In this line the abort function is used which immediately causes an error to occur resulting in the corresponding errorhandler being called.

3.4 Flask Static Files

Even though we have seen some techniques for generating web-apps and pages dynamically from code, it is often useful to have static files, e.g. javascript, images, and CSS, that are stored in the filesystem. This enables you to incorporate useful standard web tools like, for example, JQuery, into your web-app, so you don't have to write *everything* in Python.

To use static files, all you have to do is to create directory called '*static*' that is a sub-directory (child) of the directory in which your web-app is located. Notice that usually, if you were deploying your web-app in the wild as a public web-site then you would let your web-server take the load of hosting and serving up the files themselves and we will consider this approach later in the module[1]. However, during development it is sufficient to use the *static* sub-directory. You can then access the static file, for example a CSS file called *style.css*, using theFlask url_for function like so:

```
1     url_for('static', filename='style.css')
```

NB. For this to work the file *style.css* must be stored in the /static/ subdirectory. Here is full example that display an image in your browser. To get his to work you will have to create the /static/ directory then put an image file into it. You can use the curl tool at the command line to retrieve a remote image from the internet like so:

```
$ curl -L -o vmask.jpg siwells.github.io/assets/images/vmask.jpg
```

which will store an image file called *vmask.jpg* in the directory in which the command is run. You can then use the following to display the file:

```
1  from flask import Flask, url_for, abort
2  app = Flask(__name__)
3
4  @app.route("/")
5  def hello():
6    return "Hello Napier"
7
8  @app.route('/static-example/img')
9  def static_example_img():
10   start = '<img src="'
11   url = url_for('static', filename='vmask.jpg')
12   end = '">'
13   return start+url+end, 200
14
15 if __name__ == "__main__":
```

[1]Deployment, hosting, administration, & tuning of high-performance web sites is outside the scope of this module, and could alone form an entire module, however we will consider and discuss a range of tools and techniquess for deploying Flask web-apps that are generally applicable to most web sites.

```
16  app.run(host='0.0.0.0', debug=True)
```

Notice how we have used a standard HTML image (img) tag, filled in the src attibute using the output from url_for, then concatenated the three strings together to form the string that is returned by the static_example_img function. I wonder what other HTML tags could be used to return information from a URL? Perhaps we could build entire web-pages this way, by just concatenating various sections of HTML to make an entire page....[2].

If you now visit `http://0.0.0.0:5000/static-example/img` you will see the relative URL for the image file something like this:

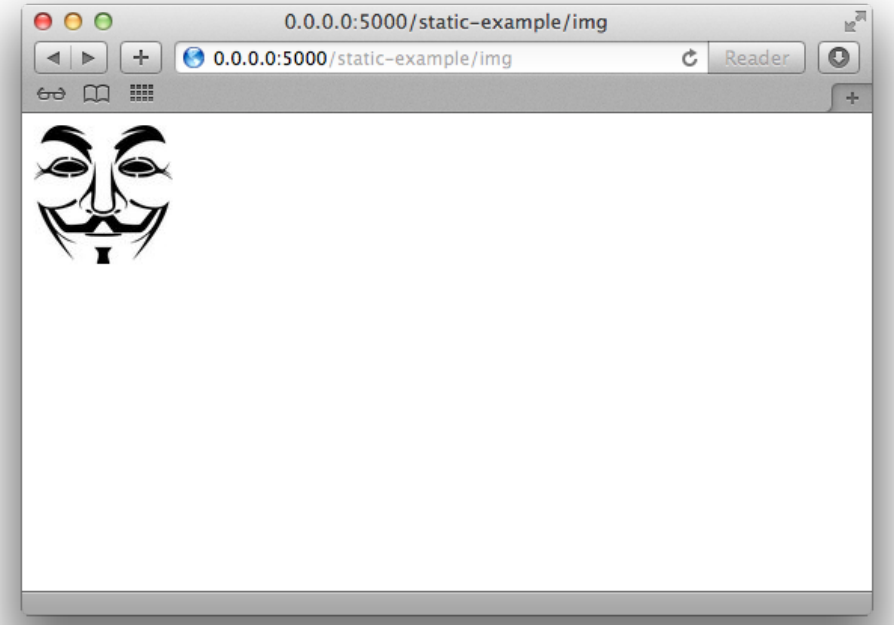

Figure 3.5: Displaying a static image using the *url_for* function

[2]You can do this, and you can return *any* HTML tags this way and build up quite complex pages. However in chapter **??** we will look at how we can use templates to design how our pages look using a mixture of HTML and special coding tags to dynamically build pure HTML responses.

Chapter 4

Python Flask: Requests & Responses

In the last chapter we looked at how to do routing. That is how to execute a different function depending upon which URL the browser requested. This week we will look in more detail at the requests themselves. Because requests can include more than just a simple HTTP GET we shall look at how to handle different HTTP methods, such as GET, POST, PUT, & DELETE, before looking at URL encoding of arguments to a route. The, after receiving and handling request data, we will look at the responses that we can return to the client.

4.1 Requests

When your browser connects to a URL it is making a **r**equest. In Flask, requests are Python objects that we can access and whose data we can reuse. For example, a request might carry a payload such as a document associated with a POST request and we will likely need to retrieve that payload document from the request in order to process the data and return an appropriate response.

You can, for development, debugging, and educational purposes, investigate the request object by printing the request.method, request.path, and request.form attributes to retrieve data about the actual request, e.g.

```
1 print request.method, request.path, request.form
```

4.1.1 HTTP Methods

HTTP uses various methods to move data around. The most commonly used method, and the one we have used exclusively until now is the default GET method. Look at the output from the Python Flask development server and you should see lines similar to the following:

```
10.0.2.2 - - [27/Sep/2015 18:59:51] "GET / HTTP/1.1" 200 -
10.0.2.2 - - [27/Sep/2015 18:59:58] "GET /hello HTTP/1.1" 200 -
10.0.2.2 - - [27/Sep/2015 19:00:05] "GET /goodbye HTTP/1.1" 200 -
```

Notice the part of each line that says GET? This is because we have been interested only in retrieving information using the *de facto* default HTTP method.

HTTP specifies a range of methods for requesting web resources and these methods are oftern referred to as HTTP Verbs. By default a client usually makes GET requests and most web resources will respond to a GET request, however a resource, identified by a route, can respond differently to different verbs. So, for example, we can make a GET request to retrieve a resource or a POST request to send information to the resource. We can then write code to respond to differnt requests in different ways.

As we said, a Flask route will accept GET requests by default, and Flask also supports HEAD requests automatically when GET is present, but we can also add support for other verbs to the route decorator method, e.g. to specify that a route can accept both GET and POST requests we would use the following decorator:

```
1 @app.route('/account', methods=['GET', 'POST'])
```

Within the method associated with that decorator we could then use an *if...else* block to execute different code, e.g.

```
1  from flask import Flask, request
2  app = Flask(__name__)
3
4  @app.route("/")
5  def root():
6    return "The default, 'root' route"
7
8  @app.route("/account/", methods=['GET', 'POST'])
9  def account():
10   if request.method == 'POST':
11     return "POST'ed to /account root\n"
12   else:
13     return "GET /account root"
14
15 if __name__ == "__main__":
16   app.run(host='0.0.0.0', debug=True)
```

To test this we can browse to `http://localhost:5000/account` and we should see the result of GET'ting the request.

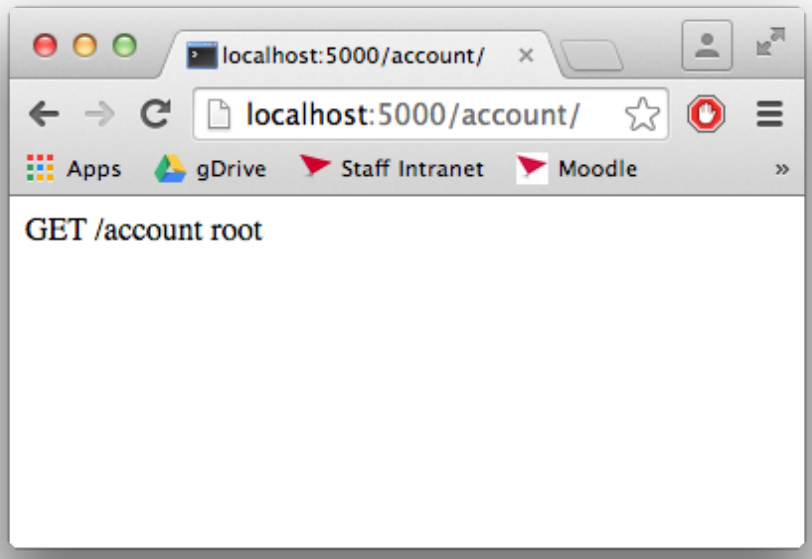

Figure 4.1: Result from GET'ting our account/ route

To test the POST call is slightly more involved for now. As we haven't implemented any HTML yet that is capable of calling this root using a POST request we need to mock one up. We can use cURL for that. The easiest way is to open a new SSH window and log in to your Levinux server then execute cURL locally on the server. We could do the same from Windows but working with the Windows command line is a pain (and we have already invested time in learning the Linux command line so we might as well continue with that. In Levinux you can now just run:

```
$ curl -i -X POST http://localhost:5000/account/
```

Which will give output like this:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 24
Server: Werkzeug/0.10.4 Python/2.7.10
Date: Sun, 04 Oct 2015 12:51:12 GMT

POST'ed to /account root
tc@box:~$
```

We can find out what HTTP method was used by checking the request method (as we saw in section 4.1.1). Data transmitted in a POST or PUT request can then be accessed using the form attribute of the request object.

4.1.2 Request & Request Form Data

Now let's look at an example that displays a form when we connect to the URL using GET then display a different page that uses data from the form when we submit a POST request by pressing the form's button.

```
1  from flask import Flask, request
2  app = Flask(__name__)
3
4  @app.route("/account/", methods=['POST','GET'])
5  def account():
6    if request.method == 'POST':
7      print request.form
8      name = request.form['name']
9      return "Hello %s" % name
10   else:
11     page ='''
12     <html><body>
13       <form action="" method="post" name="form">
14         <label for="name">Name:</label>
15         <input type="text" name="name" id="name"/>
16         <input type="submit" name="submit" id="submit"/>
17       </form>
18       </body><html>'''
19
20     return page
21
22 if __name__ == "__main__":
23   app.run(host='0.0.0.0', debug=True)
```

This should yield something similar to the following when we visit `http://localhost:5000/account/` in the browser:

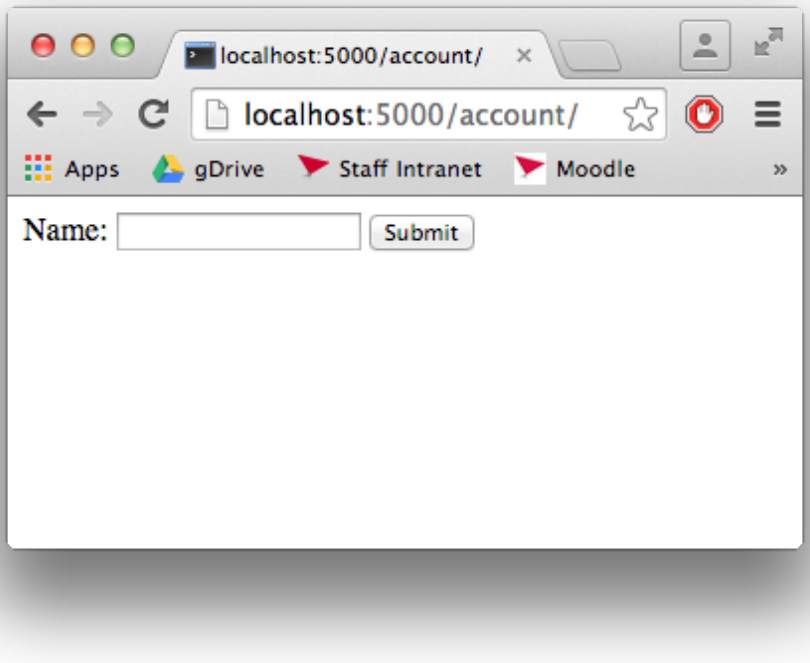

Figure 4.2: Our basic HTML form when GET'ting the account route

It is worth noting how we have used a Python multiline string, which starts with "' and ends with "' to build up an html page completely within out Python function.

When we enter data into the input box the click the button we should get different page displayed as a result of the form POST'ing:

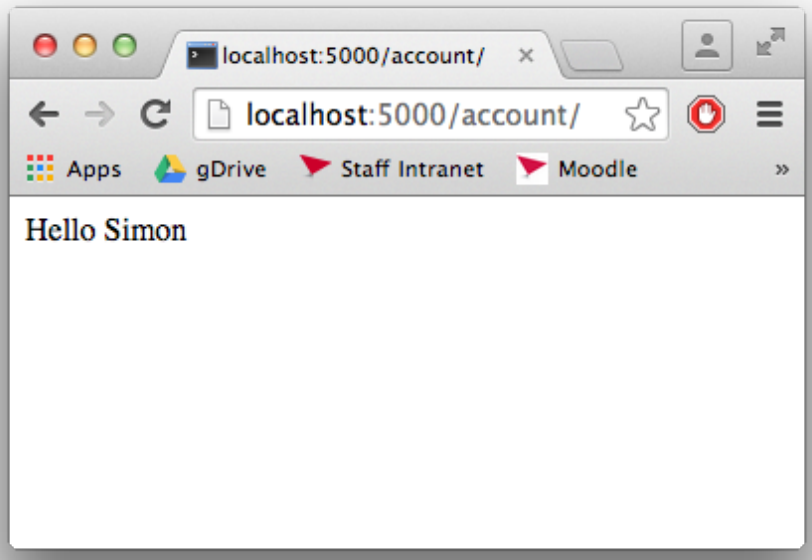

Figure 4.3: Page displayed after POST'ing the form

We will return to using POST, PUT, and other verbs in chapter 8 when we look at designing and building APIs that consume and return JSON and XML documents.

4.1.3 URL Variables

We can also construct URLs that have variables within them. For example, if we wanted a URL route that enabled us to retrieve a user's details by name then we might want a URL of the following pattern */account/<username>* where <username> is replaced by an actual name. We can achieve this using URL variables, e.g.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/hello/<name>")
5 def hello(name):
6   return "Hello %s" % name
7
8 if __name__ == "__main__":
9   app.run(host='0.0.0.0', debug=True)
```

We can now call the url, e.g. using the name 'simon' in the following `http://localhost:5000/hello/simon` and we would get the following output:

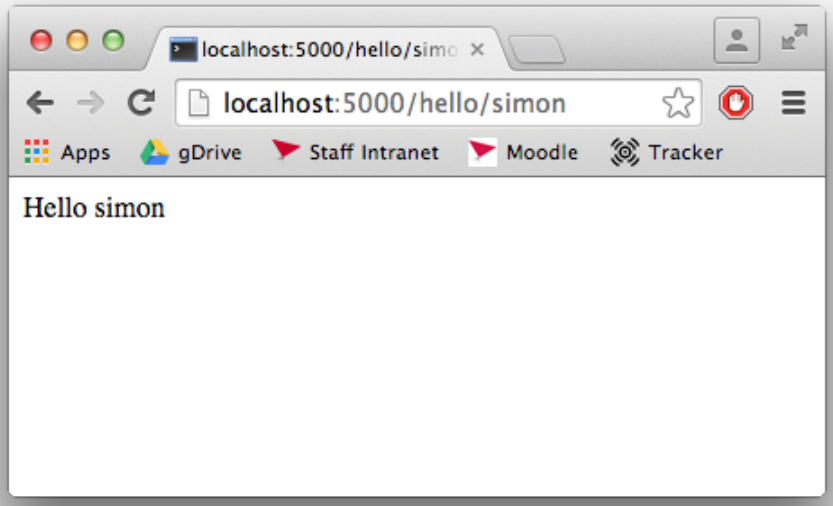

Figure 4.4: Using URL variables

By default URL variable are strings but we ca also specify other variable types such as int and floats, for example,

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/add/<int:first>/<int:second>")
5 def add(first, second):
6   return str(first+second)
7
8 if __name__ == "__main__":
9   app.run(host='0.0.0.0', debug=True)
```

With the following result:

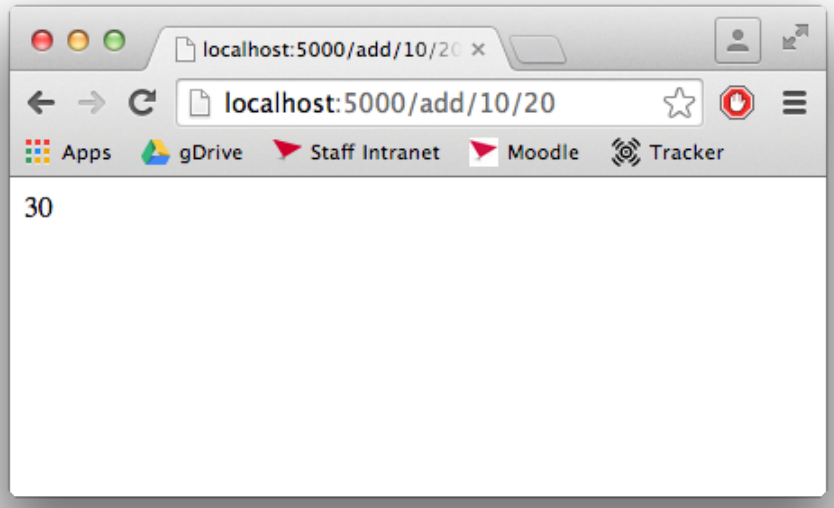

Figure 4.5: Output from using specific URL variable types

4.1.4 URL Parameters

Rather than construct and send a document to the server or use a POST'ed form, we will often want to send small amount of non-secure data to the server encoded within the URL. This is straightforward in Flask.

Parameters can be encoded in the URL when a client make a request. Flask can retrieve these parameters and use them by using the args attribute of the request object, e.g. for a URL that incorporates *?key=value* parameters similar to the following:

```
/update?colour=green
```

Then we can access the corresponding keys like so:

```
1 searchterm = request.args.get('key', '')
```

The value for key is then retrieved from the URL and stored in 'searchterm'. If no such key is supplied then the value in the second pair of quotes is used as a default instead but in the example above we have just used an empty string.

Now let's look at a simple example, which you can use to send your name as a URL encoded parameter, and have a route accept and process it we can do the following:

```
1 from flask import Flask, request
2 app = Flask(__name__)
3
4 @app.route("/hello/")
5 def hello():
6   name = request.args.get('name', '')
```

```
7   if name == '':
8     return "no param supplied"
9   else:
10    return "Hello %s" % name
11
12 if __name__ == "__main__":
13   app.run(host='0.0.0.0', debug=True)
```

When we run this without supplying a parameter, e.g `http://localhost:5000/hello/` then we get this output:

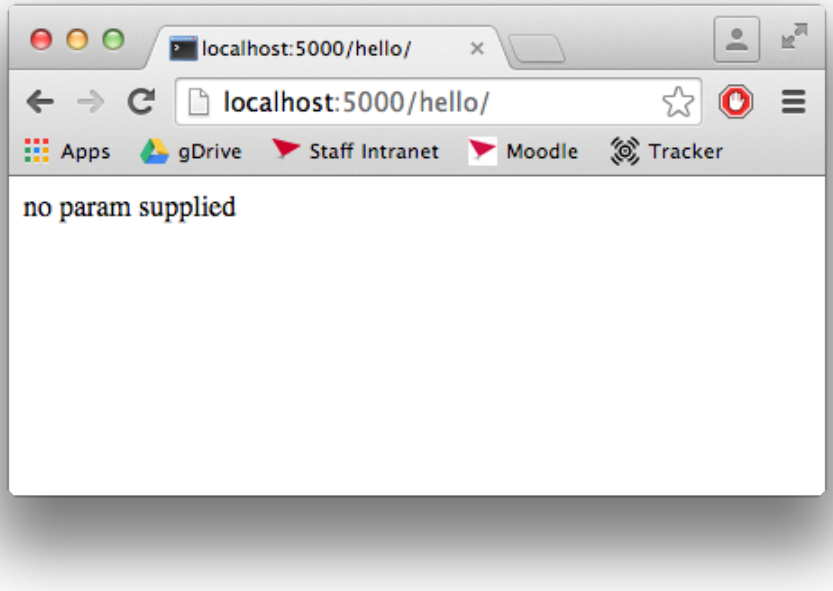

Figure 4.6: Output with no URL parameter

When we supply a parameter, e.g. `http://localhost:5000/hello/?name=simon` then we get this output:

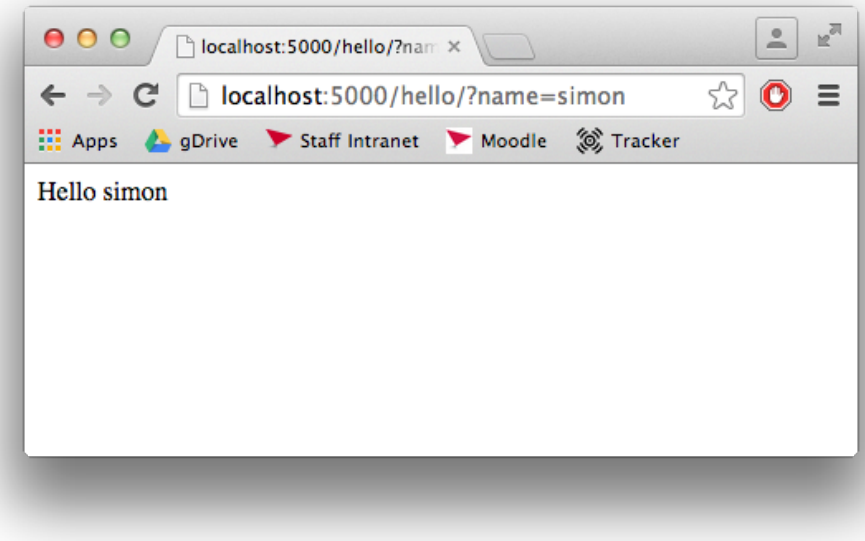

Figure 4.7: Output when supplying a ?name=simon URL parameter

4.1.5 Uploading Files

One final thing to consider that is related to client requests, is the matter of file uploading. We will occasionally want to enable our users to upload materials to our site and we can do this by ensuring that two things occur. The form which POSTs the data must cause the browser to transmit the file that we want to upload and our method that the route executes must also access the request and do something with the transmitted file. Otherwise the file will sit by default either in memory or in temptorary storage rather then being saved for later reuse. In the following I used a PNG image file that I had available as the uploaded file:

```
1  from flask import Flask, request
2  app = Flask(__name__)
3
4  @app.route("/account/", methods=['POST','GET'])
5  def account():
6    if request.method == 'POST':
7      f = request.files['datafile']
8      f.save('static/uploads/upload.png')
9      return "File Uploaded"
10   else:
11     page='''
12     <html>
13     <body>
14     <form action="" method="post" name="form" enctype="multipart/
           form-data">
15       <input type="file" name="datafile" />
16       <input type="submit" name="submit" id="submit"/>
17     </form>
18     </body>
19     </html>
20     '''
21     return page, 200
22
23 if __name__ == "__main__":
24   app.run(host='0.0.0.0', debug=True)
```

Notice that the file is being saved in a sub-directory of static called 'uploads'. This makes it easier to access the uploaded file and use it within our app. Look in 'static/uploads' for the new file. Perhaps you could combine this file upload facility with the image display example that we saw in section 3.4. For example:

```
from flask import Flask, request, url_for
app = Flask(__name__)

@app.route("/display/")
def display():
  return '<img src="'+url_for('static', filename='uploads/file.png'
     )+'"/>'

@app.route("/upload/", methods=['POST','GET'])
def account():
  if request.method == 'POST':
    f = request.files['datafile']
    f.save('static/uploads/file.png')
    return "File Uploaded"
  else:
    page='''
    <html>
    <body>
    <form action="" method="post" name="form" enctype="multipart/
       form-data">
      <input type="file" name="datafile" />
      <input type="submit" name="submit" id="submit"/>
    </form>
    </body>
    </html>
    '''
    return page, 200

if __name__ == "__main__":
  app.run(host='0.0.0.0', debug=True)
```

Of interest here is that when we access the display method repeatedly you should see in the output from the Flask development server something similar to this:

```
10.0.2.2 - - [05/Oct/2015 17:29:14] "GET /display/ HTTP/1.1" 200 -
10.0.2.2 - - [05/Oct/2015 17:29:15] "GET /static/uploads/file.png HTTP/1.1" 304
```

In this case we get the 304 because the image file hasn't changed (another reason we store it in our static repository. However, a final note on file uploads and static files. In a real-world deployment we usually wouldn't serve static files directly from within Flask as other HTTP servers like NGinX can do this much more reliably and efficiently. However for development and educational purposes using the Flask static directory is an acceptable approach.

4.2 Responses

When we return a value from a function it is automatically turned into a valid HTML response object. If the value is a String then it is used as the body of the response which is why when we just do something like this:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def root():
6     return "Hello Napier"
7
8 if __name__ == "__main__":
9   app.run(host='0.0.0.0', debug=True)
```

then our browser displays the String that the function returned. A 200/OK HTTP status code is also returned by default and the mimetype is set to text/html. We will return to this topic later when we look at API building and consider setting and returning custom headers.

Chapter 5

HTML Templates using Jinja2

Storing and writing our HTML code in Python as we have done in previous chapters is not a lot of fun. It is finicky and error-prone and doesn't give us much scope for doing interesting things like generating HTML pages on the fly. Luckily there is a solution for that. Using *templates* we can describe the basic layout of a page and sign-post those elements that Python can fill in with actual data. Python uses an external templating engine, called Jinja2[1] which is already installed in Levinux alongside Python and Python-Flask.

Jinja2 is a fully-fledged Python templating engine and is not dependent upon Flask. So if you were writing another app at some point in your career that outputs HTML pages then Jinja2 is a good option. For the moment though we shall use it exclusively with Flask.

5.0.1 Templates & Tags

The process is simple. We supply HTML templates, in a template folder, then, in our functions we tell Flask which template to render and return to the client using the render_template function. So we have a couple of setup tasks to do. First, create a templates folder in the same folder as your Python Flask app, e.g.

```
$ mkdir templates
```

Now create a simple HTML template, called hello.html inside the templates folder, e.g.

```
$ touch templates/hello.html
```

Now open hello.html in Vim for editing, e.g.

```
$ vim templates/hello.html
```

Now we can put some HTML and Jinja2 tags into our template so that we can use the template from within Flask:

```
1 <!doctype html>
2     <h1>Hello {{ user.name }}!</h1>
3 </html>
```

[1] http://jinja.pocoo.org/2/

What we have just done is create a template that is a mix of Jinja2 tags, indicated by the {{, and }} tags, and HTML tags, indicated by the < and > tags that we are already used to. The HTML tags are rendered as you would expect regular HTML to be treated, but we also have a variable placeholder for user.name which means that we can supply a variable to replace the name placeholder with. Let's use our template now in a quick Flask app:

```
1  from flask import Flask, render_template
2  app = Flask(__name__)
3
4  @app.route('/hello/<name>')
5  def hello(name=None):
6      user = {'name': name}
7      return render_template('hello.html', user=user)
8
9  if __name__ == "__main__":
10     app.run(host='0.0.0.0', debug=True)
```

Now if we call `localhost:5000/hello/simon` then we should instead see this rendered template:

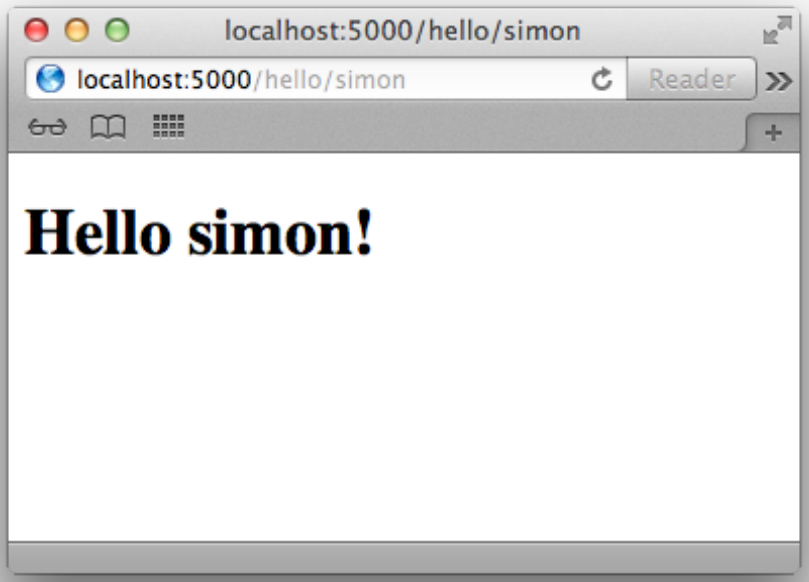

Figure 5.1: Rendered HTML with a very simple template & a single parameter

5.0.2 Templates with Conditional Arguments

We can also use Jinja2 templates to perform conditional behaviours, for example, rendering our HTML differently depending upon the data that is passed in. This lets us do things like personalise a page if we have a person's name or else provide a default generic page if we don't. Let's look at the template (conditional.html) for such a scenario:

```
<!doctype html>
{% if name %}
    <h1>Hello {{ name }}!</h1>
{% else %}
    <h1>Hello from Napier!</h1>
{% endif %}
</html>
```

The Jinja2 tags cause conditional behaviour to occur; in this case they form an if...else clause, just like we have seen in many other procedural languages like Java, C, or even Python. Let's use our template now in a quick Flask app:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('conditional.html', name=name)

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

There are a couple of things to notice here:

1. Notice how we have stacked up two @app.route calls - yes, a single function can have multiple routes defined for it, any of which can cause the function to be executed.

2. We have also used a URL variable in one of the route so that we can supply a name. Notice that the hello function takes a name argument and that it is set to 'name=None' - this just means that we have set a default value for name in case the route without the URL variable is used.

3. In the hello function we use the render_template function from the Flask library which basically looks in the templates directory for a template whose name matched the one that we have supplied, 'hello_template.html'. The function then *completes* the template, i.e. exchanging the Jinja2 tags for valid HTML tags, according to the arguments that we provide. In this case we provide the name=name argument, which will either have the value of None or else will be equal to the name that we supplied when we called the route. This argument is used to determine which path of the if...else clause to follow in the template and what value to replace any template variables with.

If we now call `localhost:5000/hello` then we should see the following:

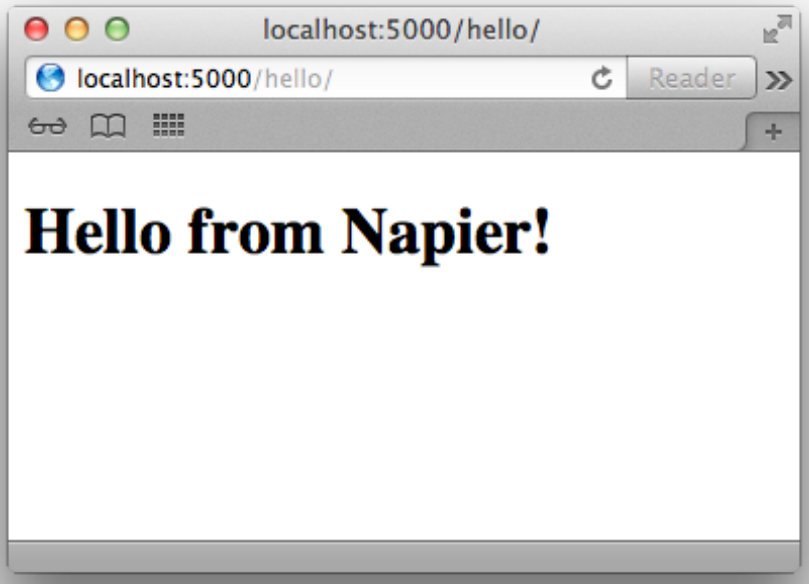

Figure 5.2: Conditional template rendering without URL arguments

But if we call `localhost:5000/hello/simon` then we should instead see this rendered template:

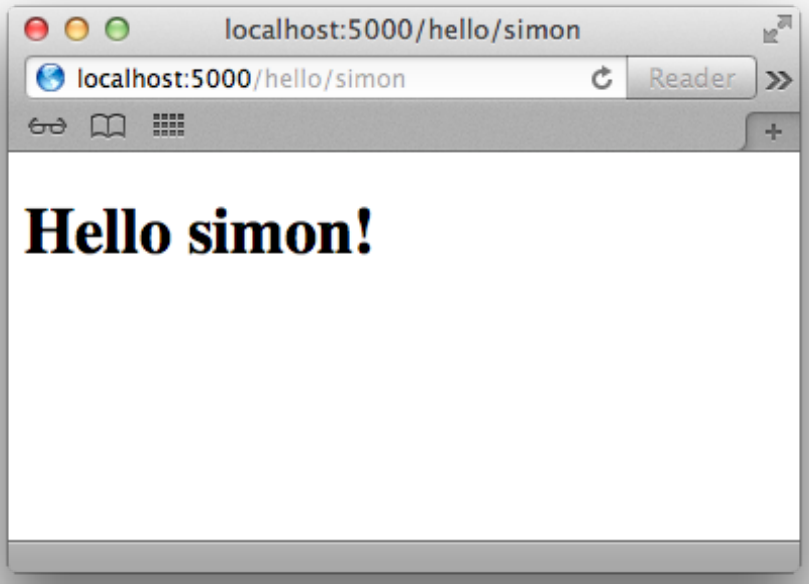

Figure 5.3: Conditional template rendering with a single URL argument

5.0.3 Templates & Collections

A very useful technique for generating HTML is to build a list or dictionary in Python then pass that collection into the template and cause the template to iterate over the elements of the collection. Let's look at a simple example now; here is a simple template that incorporates a Jinja2 looping construct:

```
<!doctype html>
<body>
  <ul>
    {% for name in names %}
     <li>{{ name }}</li>
    {% endfor %}
  </ul>
</body>
</html>
```

We can now use this template in a Python Flask function as illustrated here:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/users/')
def users():
  names = ['simon', 'thomas', 'lee', 'jamie', 'sylvester']
  return render_template('loops.html', names=names)

if __name__ == "__main__":
  app.run(host='0.0.0.0', debug=True)
```

Notice that we merely constructed a Python list, a simple data structure, that is a list of names. We then passed that list into the render_template function for processing by the templating engine. If we now visit `localhost:5000/users/` we should see that our Python list has been rendered as an unordered HTML list.

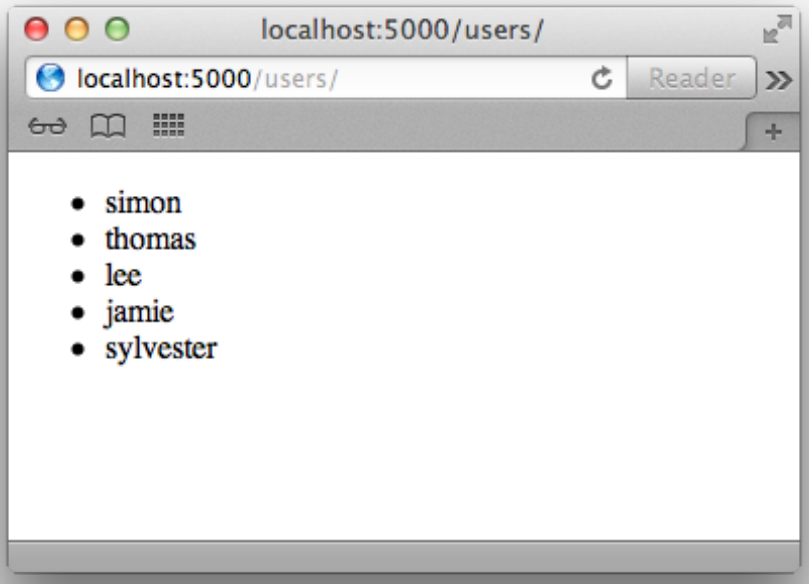

Figure 5.4: Looping over data in within a template to generate HTML

By using simple looping techniques and carefully considering the data that we pass from our Python function into a template, we can generate complex and dynamic HTML layouts.

5.0.4 Template Inheritance

Because multiple templates can also be used to define headers, footers, and any other sub-part of individual pages in your app this means you can easily change and manage the look and feel of your apps. This approach is called *Template Inheritance* and means that you can, for example, define a header or menu-bar just once, e.g. in a template called menu.html and then include that template in every other pages which you want to display the menu. If you ever wish to add or remove a menu item then you only have to make a single edit to the menu template. Nice isn't it? This follows an established software design pattern of attempting to separate application logic separate from the presentation, markup or layout of data. Although you might ask, "what about the logic in the template?", you are correct, there is a little bit of overlap where logic directly concerns rendering the templates but for the most part, done correctly, all of the computation of your web-app should be done in Flask and the rendering into HTML is done separately in Jinja2. This is a pretty good balance I think.

To demonstrate template inheritance we will first define a base template, then two templates that inherit from it. We will then create some routes that render the templates. So let's start with our base template:

```
1 <html>
2 <head>
3     <title>Template Inheritance Example</title>
```

```
</head>
<body>
    <h1>Title stored in the base template</h1>
    <h2>With a subtitle</h2>

    {% block content %} {% endblock %}

</body>
</html>
```

For the most part this is just a normal HTML file except that it incorporates some Jinja2 tags to set out blocks that we have called 'content'. It is these blocks that are replaced within the templates that inherit from the base template. We can now inherit this base template and reuse the common elements as follows:

```
{% extends "base.html" %}
{% block content %}
  <p>First example template. It contains some stuff</p>
{% endblock %}}
```

The main point to be aware of is that the *derived* template specifies that it inherits from the base template. Just to demonstrate that this works for different templates, let's create a second template that also inherits from the base template but which contains different content to the last pone.

```
{% extends "base.html" %}
{% block content %}
  <p>A second example. It's different to the other one.</p>
{% endblock %}}
```

We can make a Flask file which has routes that render our templates.

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/inherits/')
def inherits():
    return render_template('base.html')

@app.route('/inherits/one/')
def inherits_one():
    return render_template('inherits1.html')

@app.route('/inherits/two/')
def inherits_two():
    return render_template('inherits2.html')

if __name__ == ("__main__"):
    app.run(host='0.0.0.0', debug=True)
```

Here we have three different routes. The first one '/inherits/' merely shows what the base template looks like when it is rendered without additional templates that inherit from it. In the other two routes '/inherits/one/' and '/inherits/two/' we see how the base template is modified when it is extended by two different inheriting

templates. Using this approach we can build a hierarchy of page templates whilst minimising the amount of reptition by ensuring that each element that will be repeated between HTML pages is inherited from a parent template.

This is the output from our first template that inherits from base.html. It displays some content that comes from inherits1.html and some headings that are inherited from the base template.

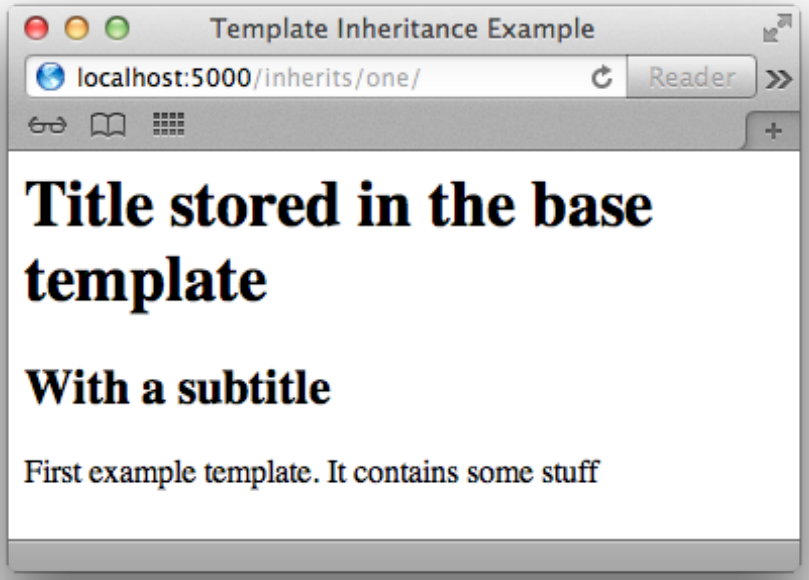

Figure 5.5: The first page that inherits from our base template

Our second page that inherits from the base template. Notice that the same headers are displayed as on the other page. However we have different content as defined by the inherits2.html template.

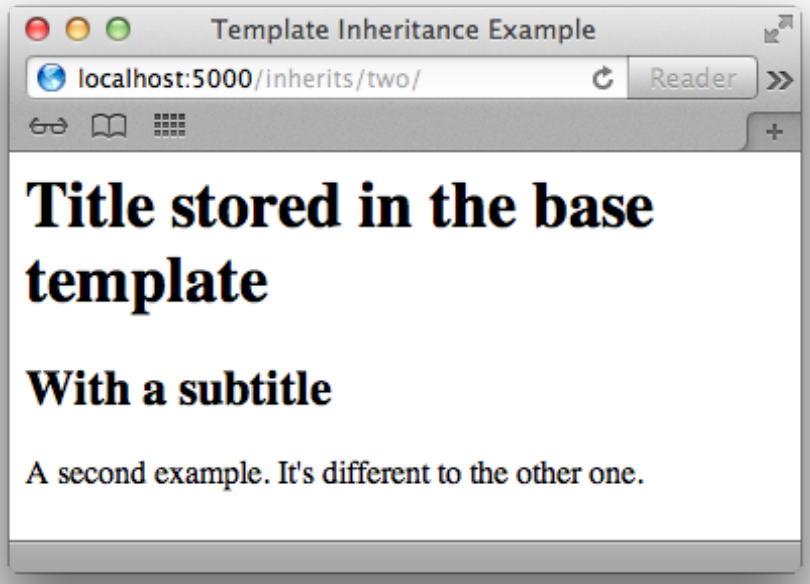

Figure 5.6: The second page that inherits from our base template

Just for completion, let's also look at what the base template looks like when rendered. We don't *have* to provide a route to it, but if we do then we can view the HTML that it generates:

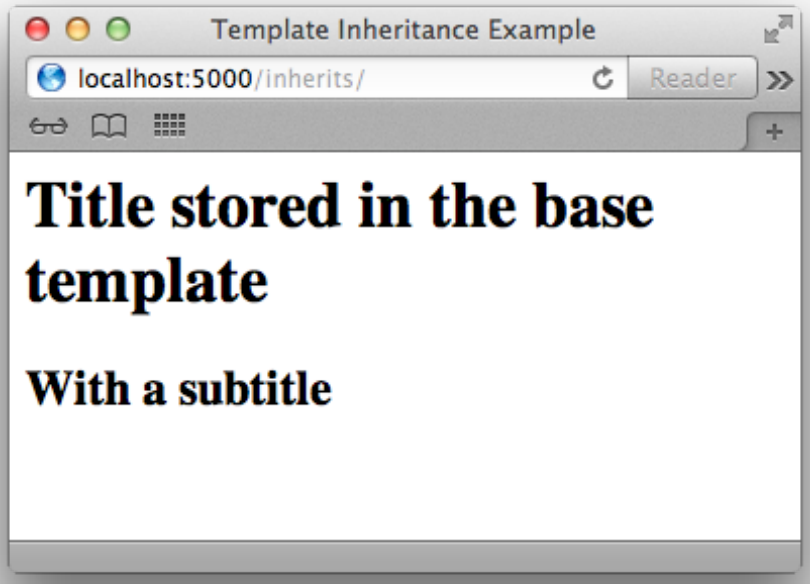

Figure 5.7: The rendered base template

You can and should use templates to describe all of the HTML pages that you want your web-apps to use. This approach is a consistent and very powerful method for generating HTML and managing the look of your web-apps.

Chapter 6

Flask: Configs, Sessions, Message Flashing, Logging, & Testing

6.1 Configuration & Config Files

Quite often we need to do external configuration of our Flask app, for example, if we are using external tools, like Babel, to control translations into different languages, if we need to specify an encryption key for setting up secure sessions, or a password for accessing an external mail server. We can achieve this by using configuration files, external text files that Flask can read at start up and which provide data to Flask about the environment it is running in.

Create a sub-directory called 'etc' just like you did for the static and templats directories. Your config files will live here. Now create a new text file in the etc directory called 'defaults.cfg', this will be the default file that your flask app reads in at start up. We can now use that file to store some configuration data. Let's start by moving our debug and host settings into the config file instead of setting them in code, e.g. Add the following to your defaults.cfg:

```
1 [config]
2 debug =  True
3 ip_address = 0.0.0.0
4 port = 5000
5 url = http://127.0.0.1:5000
```

Our config file can be used to store whichever configuration values we decide we want to support in a flask app. For illustration purposes I have used the debug flag na ip, url, port numbers settings. We can now create a Flask app that uses those values, for example,

```
1 import ConfigParser
2
3 from flask import Flask
4
5 app = Flask(__name__)
6
7 @app.route('/')
8 def root():
```

```
9     return "Hello Napier from the configuration testing app"
10
11  @app.route('/config/')
12  def config():
13    str = []
14    str.append('Debug:'+app.config['DEBUG'])
15    str.append('port:'+app.config['port'])
16    str.append('url:'+app.config['url'])
17    str.append('ip_address:'+app.config['ip_address'])
18    return '\t'.join(str)
19
20  def init(app):
21      config = ConfigParser.ConfigParser()
22      try:
23          config_location = "etc/defaults.cfg"
24          config.read(config_location)
25
26          app.config['DEBUG'] = config.get("config", "debug")
27          app.config['ip_address'] = config.get("config", "ip_address
                ")
28          app.config['port'] = config.get("config", "port")
29          app.config['url'] = config.get("config", "url")
30      except:
31          print "Could not read configs from: ", config_location
32
33  if __name__ == '__main__':
34      init(app)
35      app.run(
36          host=app.config['ip_address'],
37          port=int(app.config['port']))
```

Notice how we are 'getting' the value for each key from the config file then storing those values in the app.config object.

6.2 Sessions

Sessions are a way to manage user data between requests by storing small amounts of data within a cookie. Cookies are small data files that are stored in the users browser and are suitable for small amounts of, ideally non-private, data. Sessions rely on cookies that are cryptographically secured by Flask using a secret key to ensure that the content of the cookies hasn't been altered by any process other than the Flask app that created it. However, the content of a cookie can easily be read by the client or whilst it is transmitted between the client and server during a request[1].

Flask provides an interface for using 'secured cookies' or *sessions* rom our Python code and we can consider a session to be a small data store for keys and value, a form of dictionay. As a result we can set data into a session, query that data, and remove that data,

To add a key value pair we merely treat the session object as a Python dictionary[2], e.g.

[1] Unless the communication has been secured with HTTP but that is another topic

[2] If you are unsure about Python dictionaries or 'dicts' then you should do some background reading on this topic in the Python language documentation

```
# Set key=value, name=simon into a session
session['name'] = simon
```

You can then retrieve the value set for name in the session but because this key might not exist in the session we need to wrap everything in a try-except structure to catch the KeyError that might be raised[3].

```
try:
    if(session['name']):
        return str(session['name'])
except KeyError:
    pass
```

The only other thing that we might need to do is to remove a specified key and it's associated value from the session. We do this using the pop function, e.g.

```
session.pop('name', None)
```

We can then put it all together into a single demonstration web-app like so.

```
from flask import Flask, session

app = Flask(__name__)
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'

@app.route('/')
def index():
    return "Root route for the sessions example"

@app.route('/session/write/<name>/')
def write(name=None):
    session['name'] = name
    return "Wrote %s into 'name' key of session" % name

@app.route('/session/read/')
def read():
    try:
      if(session['name']):
          return str(session['name'])
    except KeyError:
      pass
    return "No session variable set for 'name' key"

@app.route('/session/remove/')
def remove():
    session.pop('name', None)
    return "Removed key 'name' from session"

if __name__ == "__main__":
  app.run(host='0.0.0.0', debug=True)
```

[3]Again, if you are unsure about Python errors & try-execept then you should do background reading on this

Notice the 'app.secret_key' line. This is our secret key that is used to secure our session cookie so should really be stored securely, either in a config file or typed in by hand at startup, but never put in the code repository. However for demonstration purposes this is sufficient for now. The key above is sufficient for the lab work but you would generate a unique key for any real deployment and would keep it secret. You can generate a key easily using Python. Start the Python interpreter and use the os.urandom function to generate a new key that you can then copy and paste into your Flask app, e.g.

```
$ Python
...
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x01O<!\xd5\xa2\xa0\x9fR"\xa1\xa8
    '
```

6.3 Message Flashing

User feedback is an important consideration when trying to design a good user experience (UX). Message flashing is just one aspect of UX that Flask provides to enable easy user feedback. The scenario is quite simple, when the user does something on one page, which causes another page to be rendered and displayed, then information, a message, can be transmitted from the first page to the second, and displayed, e.g. flashed, to the user. This means that, used with the correct combinations of responses, users can interact with your web app and get feedback about the outcome of actions. A simple example will illustrate this; if you have a sign-up page for new users on which the user enters information then presses a "join" button you can use message flashing to provide a personalised message to the user on the next page that is displayed. For example, after pressing "join" either the sign up page will be redisplayed, because the supplied information is insufficient, or else the login page will be displayed. A flashed message could be displayed in either case, on the sign-up page to indicate what needs to be fixed, or on the log in page indicating that a new account was created and that the user is welcome to log in. What is essentially happening is that a message is recorded at the end of the first request, which is then accessed *only* on the very next request.

Message flashing sounds quite complex and does have a few moving parts, but is really very simple once you have used it once to twice. It is best shown via example, so let's get started with one

```
from flask import Flask, flash, redirect, request, url_for,
    render_template

app = Flask(__name__)
app.secret_key = 'supersecret'

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/login/')
@app.route('/login/<message>')
def login(message=None):
```

```
13      if (message != None):
14          flash(message)
15      else:
16          flash(u'A default message')
17      return redirect(url_for('index'))
18
19  if __name__ == "__main__":
20      app.run(host='0.0.0.0', debug=True)
```

Here, we have added a flashed message in one route, using the flash() function of Flask then we use the get_flashed_messages() method in our template to retrieve the flashed message and display it, e.g.

```
1  <html>
2  <body>
3  {% with messages = get_flashed_messages() %}
4    {% if messages %}
5      <ul>
6      {% for message in messages %}
7        <li>{{ message | safe }}</li>
8      {% endfor %}
9      </ul>
10   {% endif %}
11 {% endwith %}
12 </body>
13 </html>
```

All that we have done here is add a flashed messagein the '/login/<message>' route then to display the flashed message when we visit the '/' page. Subsequent visits to the '/' page will not repeat the flashed message, unless of course, we visit the 'login/<message>' page again

6.4 Logging

We can set our Flask app up to record interesting happenings into a text file so that we have a log to inspect if something bad occurs. This is actually a feature of the Python language rather than a Flask feature, but is an important part of building a real world app. First we need to make some additions to our config file, named 'logging.cfg' and stored in the 'etc' folder:

```
1  [config]
2  debug =  True
3  ip_address = 0.0.0.0
4  port = 5000
5  url = http://127.0.0.1:5000
6  [logging]
7  name = loggingapp.log
8  location = var/
9  level = DEBUG
```

Notice the logging section towards the end which defines the name of the log file 'loggingapp.log' the location, our var/ directory that we just created, and the default log level, the granularity of the logging events to record.

We now need to set up our envionrment to match the config file. We need a sub-directory called 'var' which is at the same level as our 'etc', 'static' and 'templates' directories. We now need to create an empty file in 'var' which has the same name as the log file. This directory will be the location that our new log files will write to. We can do this with touch, e.g.

```
$ mkdir var
$ touch var/loggingapp.log
```

Having set everything up all the configuration details and the basic environment we now need to make use of that set up within our python app. Create a new file called 'loggingapp.py' and enter the following code:

```
1  import ConfigParser
2  import logging
3
4  from logging.handlers import RotatingFileHandler
5  from flask import Flask, url_for
6
7  app = Flask(__name__)
8
9  @app.route('/')
10 def root():
11         this_route = url_for('.root')
12         app.logger.info("Logging a test message from "+this_route)
13         return "Hello Napier from the configuration testing app (
               Now with added logging)"
14
15 def init(app):
16     config = ConfigParser.ConfigParser()
17     try:
18         config_location = "etc/logging.cfg"
19         config.read(config_location)
20
21         app.config['DEBUG'] = config.get("config", "debug")
22         app.config['ip_address'] = config.get("config", "ip_address
               ")
23         app.config['port'] = config.get("config", "port")
24         app.config['url'] = config.get("config", "url")
25
26         app.config['log_file'] = config.get("logging", "name")
27         app.config['log_location'] = config.get("logging", "
               location")
28         app.config['log_level'] = config.get("logging", "level")
29     except:
30         print "Could not read configs from: ", config_location
31
32
33 def logs(app):
34     log_pathname = app.config['log_location'] + app.config['
           log_file']
35     file_handler = RotatingFileHandler(log_pathname, maxBytes=1024*
           1024 * 10 , backupCount=1024)
36     file_handler.setLevel( app.config['log_level'] )
37     formatter = logging.Formatter("%(levelname)s | %(asctime)s |
           %(module)s | %(funcName)s | %(message)s")
38     file_handler.setFormatter(formatter)
39     app.logger.setLevel( app.config['log_level'] )
40     app.logger.addHandler(file_handler)
```

```
41
42 if __name__ == '__main__':
43     init(app)
44     logs(app)
45     app.run(
46         host=app.config['ip_address'],
47         port=int(app.config['port']))
```

The init function is very similar to the one we used earlier in the config example but is now extended to also configure event logging. There are many parameters to fine tune how data is logged so if you want to make changes then you will have to dig into the Python logging documentation but for now, just accept the defaults as they produce files that cope well with lots of data, are easy to read, and which can be automatically processed. All we have had to do to use the logger is to initialise the logging system by calling our configuration modules logs() function. From then on we can actually log messages using the app.logger.warn() and passing in a String. NB. There are also other logging levels such as debug or error that we can use to distinguish the importance of different messages in the logs. Investigate the Python logging documentation to find out more.

Now, if everything is set up correctly, then every time we visit `http://localhost:5000/` a new log file should be added to var/loggingapp.log and we can use an extra PuTTY window to "tail" the log so that we see each new log line appear in realtime. Tailing a log just means to show the last entries in the file and using the '-f' argument to the tail command causes it to follow the log, meaning that each new line is displayed in the terminal like so:

```
tc@box:~$ tail -f var/loggingapp.log
INFO | 2015-10-13 17:44:37,368 |  logs | root | Logging a test message from /
INFO | 2015-10-13 17:44:45,104 |  logs | root | Logging a test message from /
INFO | 2015-10-13 17:44:46,325 |  logs | root | Logging a test message from /
^C
tc@box:~$
```

From this we can see that there were three log messages displayed and that the output is arranged into columns. Getting the output nice and neat like this is the main reason we had to write so much code in our python app and config file, but it is really worth it when you are trying to track down a problem.

It is a good idea to log anything that you think that you might want to have a record of and that you might need to look up in order to bug fix. Unfortunately though there are no rules for what to log and what not to log. Obviously you could log *everything* but this would possibly waste disk space, but not logging enough might mean that you don't have sufficient logs to help you fix problems. However, with experimentation and experience you will develop skills in gauging the right amount of and type of data to log.

6.5 Testing

We can unit test our Python app, to ensure that everything is working correctly. We do this by running a test harness which sets up our Flask app then compares expected outputs to actual outputs, for example, for the following simple web-app:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route('/')
5 def root():
6   return "HELLO NAPIER", 200
7
8 if __name__ == "__main__":
9   app.run(host='0.0.0.0',debug=True)
```

We can write tests that compare the response returned by a call to the '/' route against what we would expect. For example, is the content correct, is the content type set corretly, is the status value set correctly. Let's see some of these tests now.

```
1 import unittest
2 import testing
3
4 class TestingTest(unittest.TestCase):
5   def test_root(self):
6     self.app = testing.app.test_client()
7     out = self.app.get('/')
8     assert '200 OK' in out.status
9     assert 'charset=utf-8' in out.content_type
10    assert 'text/html' in out.content_type
11
12 if __name__ == "__main__":
13   unittest.main()
```

We have create a class for the test and asserted that the response contents match some of our expectations, e.g. that the response status is '200 OK'. Notice in line 2 that we have also imported our flask app source file so that it is accessible from this test script. To use this we just run it in the shell, e.g.

```
tc@box:~$ python testing_test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.476s

OK
```

We can easily test what would happen if a test was failed by setting things up that way, let's return 404 from our root function, e.g.

```
1 @app.route('/')
2 def root():
3   return "HELLO NAPIER", 404
```

Now the output from our test suite shoudl be similar to the following:

```
tc@box:~$ python testing_test.py
F
======================================================================
FAIL: test_root (__main__.TestingTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "testing_test.py", line 9, in test_root
    assert '200 OK' in out.status
AssertionError
```

```
----------------------------------------------------------------------
Ran 1 test in 0.416s

FAILED (failures=1)
```

We can write as many tests as we need to to ensure that our web-app, or any Python app, runs the way that we expect it to. We do this by writing a new class in our testing script, called a *test harness*, for each test that we want to run. The unit testing framework will then discover each test class and run them, then output the results so that you know, after each edit of your code, whether you accidentally broke anything or, more importantly, changed the behaviour of code that worked previously. Testing is very important and unit testing is just one tool that we have to help us with working with larger bodies of source code. As a code base gets bigger it can sometimes be difficult to tell with a given change has subtly altered the behaviour of something else. Unit testing gives us more confidence that we haven't done so.

Chapter 7

Using Bootstrap to Add Style

We have concentrated so far on building well designed web apps that include useful features from Flask that enable us to test and configure our apps. However, we haven't spent a lot of time on making things look nice. The most we have done is use templates, in section 5, to scaffold the consistent generation of HTML for our users to read. We have alsoused static files, which we can use for things like Javascript, Cascading Style Sheets (CSS), icons and images, back in Section 3.4. We can further exploit the '/static/' directory to give us an easy way to add some style to our flask app as this is the correct place to store all of those elements that would be made to make our pages look nice. We should recognise however that this module is not a *web design* module, so we shall not spend time concentrating on the design of beautiful web sites, but we shall take a short cut to get a decent and consistent basic design that provide basic framework which can be further modified to produce a beautiful app.

Bootstrap is a set of CSS files and supporting Javascript that make it straightforward to add a basic style, with a selection of different layouts, to your web-app. Originally, Bootstrap was developed by Twitter as an easy way to *bootstrap* from plain HTML to a site with a basic overall design which can be enhanced and which at least looks consistent. This removes the need to initially develop HTML and CSS for layout elements like headers and footers, or menus, all you have to do is use those elements that Bootstrap has pre-defined. You can of course alter these pre-defined elements but at least you don't have to design them when you are also struggling to implement all of the actual functionality.

Because the Flask static directory hosts files that can be served up directly to the client, static is also the natural home for the Bootstrap files. Your HTML templates can then reference these CSS and Javascript files so that they are served to the client when a request is made.

As your web-app grows it is important to maintain control and organisation of your source code so I find that it is a good idea to create a hierarchy of directories within /static/ in which to store your bootstrap files, for example,

```
static/
static/css/
static/font/
static/ico/
static/img/
static/js/
```

For the moment though we can use the directory hierachy that Bootstrap comes with. Change directory into your static directory then download Bootstrap from `https://github.com/twbs/bootstrap/releases/download/v3.3.5/bootstrap-3.3.5-dist.zip` and unpack it, e.g.

```
$ curl -L -o bootstrap.zip https://github.com/twbs/bootstrap/releases/download/
    v3.3.5/bootstrap-3.3.5-dist.zip
$ unzip bootstrap-3.3.5-dist.zip
```

Now move the bootstrap files into directory hierarchy that you created. For example, assuming you are in the static directory and you downloaded the bootstrap zip file

```
$ cp -R bootstrap-3.3.5-dist/* .
```

This should recursively copy all of the folder from within the bootstrap directory into the static directory.

We are now set up to start to use bootstrap to provide some style. But first, let's create an unstyled page, using a Jinja2 template and a simple Flask app that returns an HTML page using the template. We will then extend the template to add some bootstrap functionality to it. So, let's start with the basic template:

```
1  <html>
2  <head>
3  <title>Bootstrap Demonstation</title>
4  </head>
5  <body>
6  <h1>HELLO</h1>
7  <h2>Napier</h2>
8  <p>Demonstrating a flask app with templates for html generation and
       bootstrap
9  for a little bit of style</p>
10 </body>
11 </html>
```

Now we need a basic Flask app that uses the template:

```
1  from flask import Flask, render_template
2  app = Flask(__name__)
3
4  @app.route('/')
5  def root():
6    return render_template('base.html'), 200
7
8  if __name__ == "__main__":
9    app.run(host='0.0.0.0', debug=True)
```

Take a look at how your unstyled Flask app appears in the browser. Not very pretty is it?

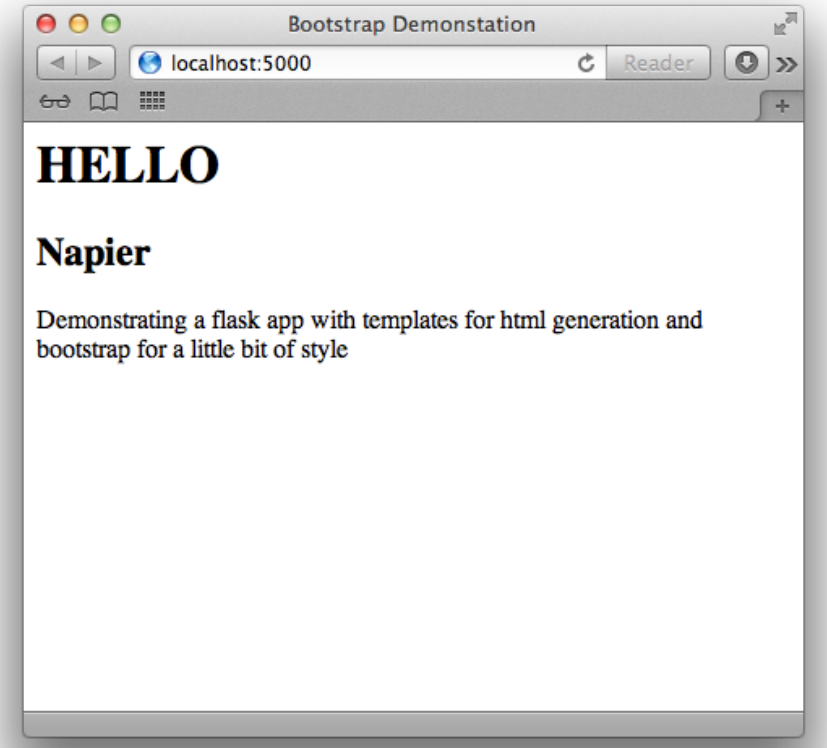

Figure 7.1: The unstyled HTML page for the Bootstrap example

Now let's use Flask to add some basic styling to that ugly old page.

```
1  <html>
2  <head>
3    <title>Bootstrap Demonstation</title>
4    <link href="{{ url_for('static', filename='css/bootstrap.min.css
       ') }}" rel="stylesheet" />
5  </head>
6  <body>
7  <h1>HELLO</h1>
8  <h2>Napier</h2>
9  <p>Demonstrating a flask app with templates for html generation and
       bootstrap
10 for a little bit of style</p>
11 </body>
12 </html>
```

Notice how all we ahve done is add a relative link, in line 4, to the Bootstrap CSS file. Also note that we used the url_for function to get the root of the static directory and that all of the URL aspects are enclosed in Jinja2's double curly braces. Even, just including the bootstrap CSS has already got things looking different:

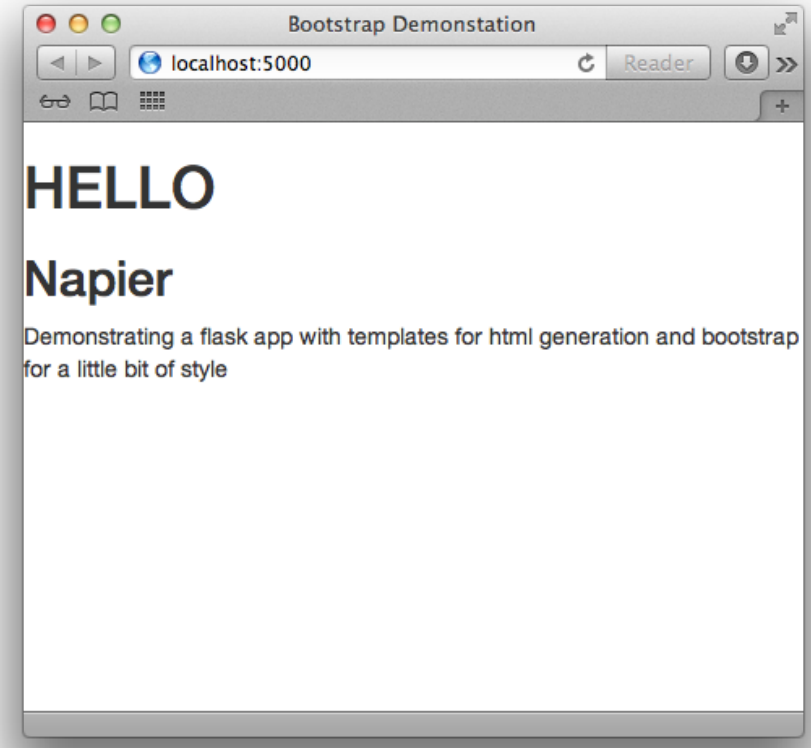

Figure 7.2: After only including the Bootstrap CSS file

But let's go a little further an implement the basic Bootstrap template:

```
<html>
<head>
  <title>Bootstrap Demonstation</title>
  <link href="{{ url_for('static', filename='css/bootstrap.min.css
     ') }}" rel="stylesheet" />
  <style>
    body{
      padding-top: 50px;
    }
  </style>
</head>
<body>

  <nav class="navbar navbar-inverse navbar-fixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed"
      data-toggle="collapse" data-target="#navbar" aria-expanded="
         false"
      aria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">ProjectName</a>
      </div>
```

```
26        <div id="navbar" class="collapse navbar-collapse">
27        <ul class="nav navbar-nav">
28          <li class="active"><a href="#">Home</a></li>
29          <li><a href="#about">About</a></li>
30          <li><a href="#contact">Contact</a></li>
31         </ul>
32         </div>
33      </div>
34    </nav>
35
36     <div class="container">
37          <h1>HELLO</h1>
38          <h2>Napier</h2>
39          <p class="lead">Demonstrating a flask app with templates
               for html generation and bootstrap for a little bit of
               style</p>
40    </div>
41
42    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/
         jquery.min.js"></script>
43    <script src={{ url_for('static', filename='js/bootstrap.min.js')
         }}"></script>
44  </body>
45  </html>
```

There are a few things to note here. Firstly we had to add an extra inline CSS script element to push the page content below the navigation bar. Remove this line (line 6) to see what would happen otherwise. After that you can see that we used to containers, the first containing our navigation bar, and the second containg our page content. Finaly we added a pair of script links, the first to JQuery[1] and the second to the default Bootstrap Javascript file which is in our static directory. Look on the Bootstrap website to find out what you can do with the Bootstrap Javascript, and on the JQuery homepage to see what functionality JQuery offers you.

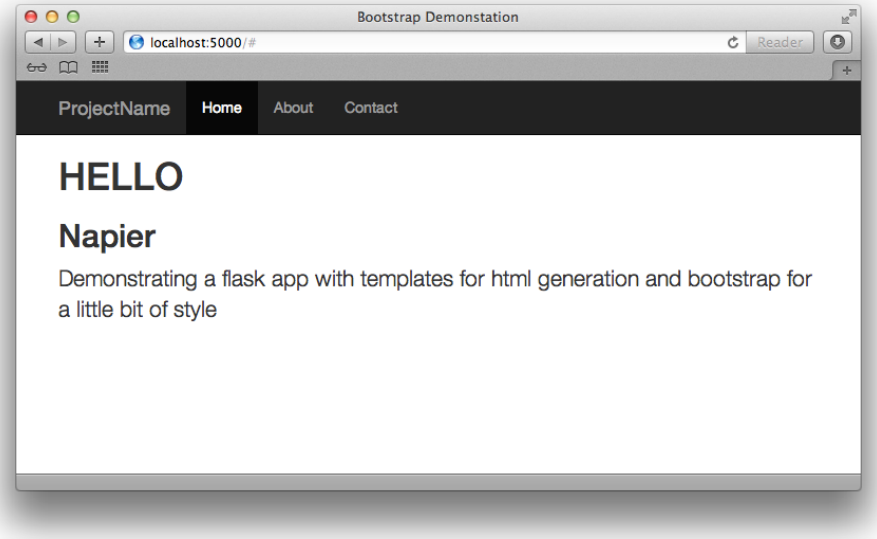

Figure 7.3: Bootstrap example with navigation bar

[1] https://jquery.com/

And this is what our page looks like in responsive mode, with a mobile style menu button:

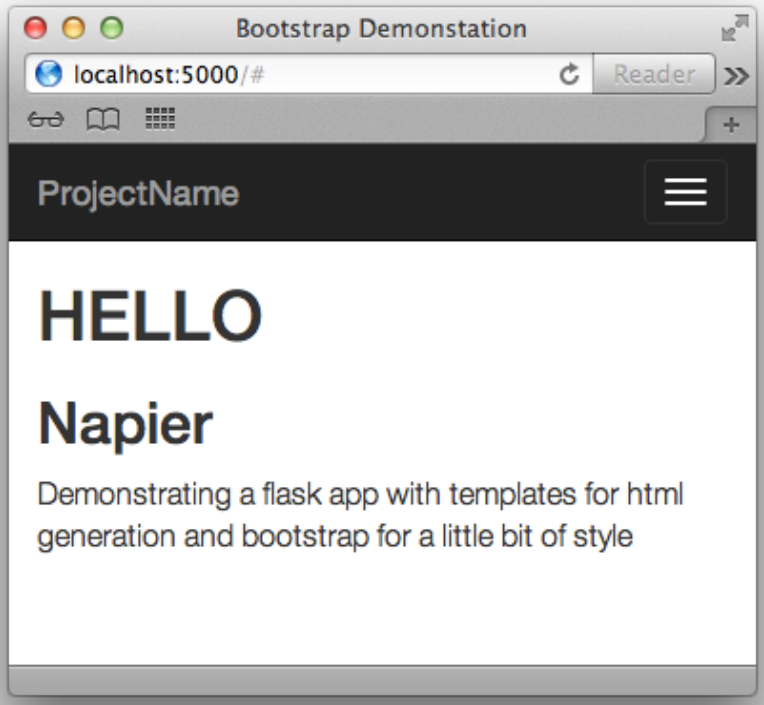

Figure 7.4: Bootstrap example in responsive mode

The bootstrap site demonstates most of the features of bootstrap so you can see how things should look by default. It is also often useful to view the source of a given example element on the bootstrap site in order to see how it should be used. To find out about all of the things that you can do with bootstrap visit `http://getbootstrap.com/getting-started/`. Once you have got to grips with the default bootstrap there are also sites that offer additional, free, bootstrap based themes, e.g.

- `http://startbootstrap.com/`
- `http://www.bootstrapzero.com/`

Finally, if you can't find something that you like, then you can always just extend or alter the default Bootstrap themes by editing their CSS. Whatevery you choose to do, Bootstrap is a good way to quickly get a not too bad looking site knocked together.

Chapter 8

Data Storage

Many web-apps are increasingly data driven. Whilst there are discussions to be had about the correct balance of static and dynamic functionality[1], it is often the case that a database is an important part of managing the data associated with a web site.

8.1 Brief Introduction to SQLite3

The easiest way to get started with data storage for this module is to use SQLite3. You can install SQLite3 using the Levinux extension manager, TCE, as follows (NB. If you haven't used TCE yet then visit section C.3 first before continuing):

```
$ tce-load -wi sqlite3-bin.tcz
```

Using SQLite3 from Python is quite straightforward if you have worked with a relational database before. You will need to import the Python SQLite3 interface library, set a location where the SQLite database will be stored, initialise a connection to the DB.

```
>>> import sqlite3
>>> DB = 'var/sqlite3.db'
>>> conn = sqlite3.connect(DB)
>>> cursor = conn.cursor()
```

In this case we have performed our imports, then created a DB object that stores the location and name of our database file, e.g. sqlite3.db which is stored in the relative *var* directory. We then connected to our database file and stored the connection in the variable called *conn* before retrieving a cursor that we can use to work with our database connection.

Now we have retrieved a cursor we can use it to insert and remove data from our database as well querying the data stored in it. First though, we need to set up a table. Let's create a table to store music called *albums*.

[1] A completely static website, e.g. one that has been designed to be static or has been '*flattened*' can be **very** scalable. The equipment that powered Github pages was for many years very modest yet made tens of thousands of static web-pages for many open source projects. *NB.* Github pages also hosts the web pages for this module as well as hosting our public source code repository.

```
>>> cursor.execute(""" CREATE TABLE albums
... (title text, artist text, release_date text, publisher text, media_type
    text)
... """)
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

Notice that we used a multi-line Python string which is wrapped in tiple quotes. Now we can add some data to our database, lets add a couple of albums:

```
>>> cursor.execute('INSERT INTO albums VALUES ("Greatest Hits", "Roy Orbison",
    "30.11.1977", "SWRecords", "vinyl")')
<sqlite3.Cursor object at 0x1054ec570>
>>> conn.commit()
```

We can use a simple SQL query to see the contents of our fledgeling database as follows:

```
>>> for row in cursor.execute("SELECT rowid, * FROM albums ORDER BY artist"):
...   print row
...
```

We can also use SQL to carry out more complex queries but the following should give us a flavour of what we can do:

```
>>> sql = "SELECT * FROM albums WHERE artist=?"
>>> cursor.execute(sql, [("Roy Orbison")])
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()
[(u'Greatest Hits', u'Roy Orbison', u'30.11.1977', u'SWRecords', u'vinyl')]
```

In this case we just defined an SQL statement which we stored in the dql variable then we executed that statement using the cursor.execture function and a supplied term "Roy Orbison" to search for. As this matched a record in the database we had a record in the list that cursor.fetchall() returns. We can also use wildcards and the SQL "like" keyword to find close but not exact matches to our query term, e.g.

```
>>> term = "orbi"
>>> cursor.execute("SELECT * FROM albums WHERE artist LIKE '%{term}%'".format(
    term=term))
<sqlite3.Cursor object at 0x1054ec570>
>>> cursor.fetchall()[(u'Greatest Hits', u'Roy Orbison', u'30.11.1977', u'
    SWRecords', u'vinyl')]
```

For more advanced SQL queries we should look to the SQL documentationbecause this module isn't really about SQL query construction. For now however we can move on to look at how to integrate an SQLite3 database with Flask.

8.2 Using SQLite3 with Flask

Now we have seen how SQLite3 works with Python as a general data storage mechanism we can now look at how, with the addition of a few simple functions, we can provide a robust mechanism for storing our web-app's data.

We'll start with defining a schema file that can be used to initialise our database as follows:

```
DROP TABLE if EXISTS albums;

CREATE TABLE albums (
    title text,
    artist text,
    media_type text
);
```

This schema just deletes any existing table called 'albums' then create a new table called 'albums' with three text fields to store the 'title', 'artist' and 'media type' of the albums. We can now use the schema in a Flask file. Let's look at a basic Flask app that includes a single route and some datastorage using SQLite3:

```
from flask import Flask, g
import sqlite3

from flask import Flask
app = Flask(__name__)
db_location = 'var/test.db'

def get_db():
    db = getattr(g, 'db', None)
    if db is None:
        db = sqlite3.connect(db_location)
        g.db = db
    return db

@app.teardown_appcontext
def close_db_connection(exception):
    db = getattr(g, 'db', None)
    if db is not None:
        db.close()

def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()

@app.route("/")
def root():
    db = get_db()
    db.cursor().execute('insert into albums values ("American
       Beauty", "Grateful Dead", "CD")')
    db.commit()

    page = []
    page.append('<html><ul>')
    sql = "SELECT rowid, * FROM albums ORDER BY artist"
    for row in db.cursor().execute(sql):
        page.append('<li>')
        page.append(str(row))
        page.append('</li>')

    page.append('</ul><html>')
```

```
43      return ''.join(page)
44
45  if __name__ == "__main__":
46      app.run(host="0.0.0.0", debug=True)
```

There are four things to notice here:

1. The init_db function loads our schema file and initialises a new database. We could call init_db() each time we restart the app but this would re-initialise the data each time. Instead we will create a separate external file that calls this function whenever we need it *we shall come back to this in a moment.*

2. The get_db function can be called from within a route to get access to our database connection, e.g.

   ```
   db = get_db()
   ```

   and we can then get a cursor using

   ```
   db.cursor()
   ```

3. The close_db_connection function is a *decorated* function that is automatically called to close the db connection whenever necessary. This is usually when a request end which causes that current context of the flask app to end.

4. Our root function which defined the '/' route does two things, it stores a new album in the database each time the route is accessed. Secondly, this function executes a simple SQL query to retrieve all the entries from the albums table then constructs a small HTML file to display those entries.

If we now run this flask app then we will get an error when we hit the '/' route so there is one last thing to do. The error occurs because we don't call the init_db function from anywhere. We probably only want to do this once when we deploy our web-app, otherwise we will lose all of the contents of the DB so we can create an external Python script, 'init_db.py', that will import the init_db function then execute it to initialise a new database.

```
1 from datastore import init_db
2 init_db()
```

We can now call our script before we start our flask app for the first time to initialise the database as follows:

```
$  python init_db.py
```

This should be enough SQLite3 and Flask integration to get started storing data. Obviously there is lots more to learn about the functionality that SQLite3 offers but that is an exercise for your self-directed study.

There are many data storage mechanisms and the availabilty of high-quality and performant datastores has increased greatly in recent years with the advent of the NoSQL approach. This approach suggests that there are **N**ot **O**nly **SQL** based approaches to storing data but many approaches, and the one that you choose should be based upon a sound assessment of the nature of the problem that you are tackling as well as the knowledge and experience of your development team. So, for example, there are a number of column-oriented, document oriented, graph-oriented, and key-value datastores, and many hybrids, which can be very useful when developing new web-apps. For example, when trying out various solutions to a problem it can be useful to use a schemaless datastore to hold your initial attempts at building a data model so that you do not waste time prematurely attempting to identify good database schema or relational models.

Chapter 9

Keeping Data safe with Encryption

Encryption is necessary to ensure that the data that we collect in our web-apps, for example, about our users, is stored safely. A good general approach is to not store any data about your users that you don't actually need for the purposes of your application.

9.1 Using PyCrypto Library for data encryption

PyCrypto[1] is the Python Cryptography Toolkit and provides a range of useful tools for encyrpting and working with encrypted data in Python. Because our Levinux install is stripped down and does not include a full compiler toolchain for languages like C, which pycrypto uses, we can instead install a prebuilt PyCrypto library as follows:

```
$ curl -L -o pycrypto-2.6.1-py2.7-linux-i686.egg https://www.dropbox.com/s/
    meuvy3d7h1n7k3r/pycrypto-2.6.1-py2.7-linux-i686.egg?dl=1
$ sudo su
# easy_install pycrypto-2.6.1-py2.7-linux-i686.egg
# exit
$ python -c "import Crypto"
$
```

What we did here was to change to the superuser role using sudo. We then installed the Python egg using easy_install. We then exited the superuser role and used python with the -c option to check that our egg had installed properly. All the call to python is doing is compiling the code that we passed in. The code we passed in is a library import command and if the Crypto library is available then there will be no feedback, just our prompt. If there is any other output then the pycrypto library is not available. Notice that the name of the library that we import within Python is different to the name of the library that we installed.

IMPORTANT When we use easy_install or pip to add a new library for Python to use then we also need to make the new addition *persistent*. This is straightforward. We add the new folder, into which our new Python library was installed, to the */opt/.filetool.lst* file then run the *filetool.sh* script which takes

[1] https://www.dlitz.net/software/pycrypto/

all the folders from .filetool.lst and saves them so that they are available after you reboot. For example, after we install the pycrypto egg, but before we reboot we must do the following:

1. Open /opt/.filetool.lst in vim, e.g.

```
$ vim /opt/.filetool.lst
```

2. Add the location of the pycrypto package folder to the end of the list of folders in ./filetool.lst (one folder per line), e.g. */usr/local/lib/python2.7/site-packages/pycrypto-2.6.1-py2.7-linux-i686.egg/*

3. Run the filetool.sh script, e.g.

```
$ filetool.sh -b
```

   and after a few moments your newly installed libraries will persist between reboots.

You can now import and use PyCrypto within your web-apps, however let's first explore what the library can do in the Python REPL, so launch python in your shell, e.g.

```
$ python
```

We can now look at some of the things that we can use PyCrypto to do. First we shall look at Hashing, taking input and calculating a fixed-length output called a hash-value, then we will look at Encryption, taking input and a key and producing a cypher text. The main difference between the two in practise is that hash algorithms are designed, as a rule, to be one-way or *trapdoor* functions, whilst, so long as you know the key, encryption algorithms are designed to enable you to recover the input from the cyphertext.

9.1.1 Hashing

The idea with Hashing is to take as input a string and calculate a fixed length output string called the *hash value*. This is a useful way to quickly determine whether two strings are the same, for example if wanting to ensure that the string that was transmitted is the one that was received. In that case we would calculate the hash value associated with the string, then transmit both the string and the hash value to the remote recipient who then recalculates the hash value from the string then compares the two hash values. if they differ then the string was altered in transit and if they are the same the string that was recieved is the one that was sent. This kind of usage is called *file integrity checking* but is also known as *checksumming* or *calculating a checksum*.

Hashing relies on various assumptions:

1. It should be very difficult (if not impossible) to guess the input string based upon the output string

2. It should be very difficult (if not impossible) to find two different input strings that have the same output (known as a *collision*)

3. It should be very difficult (if not impossible) to modify the input string without also modifying the hash value as a result.

We can hash a value, e.g. a string, using a range of hash functions but we can start with SHA256. First we will hash the value passed directly to SHA256 then we will compare it to the same value stored in a string and passed in, and a different value:

```
>>> from Crypto.Hash import SHA256
>>> SHA256.new('Hello Napier').hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5694'
>>>
>>> hello_napier = 'Hello Napier'
>>> SHA256.new(hello_napier).hexdigest()
'8fa01d2f5f442915112a4c98d5c65c909f8b5532f01102371cf81776b91e5694'
>>>
>>> SHA256.new('Goodbye Napier').hexdigest()
'ebccfcde2209e3dff4deff67fdbae71129ea87692e40295768de48317244347e'
>>>
```

9.1.2 Encryption with Block Cyphers

Block cyphers work on their input data in *blocks* of a fixed size, for example, blocks of 8 or 16 bytes in length. We'll use the Data Encryption Standard (DES)[2] as our example. The basic form of DES is pretty comprehensively broken as a secure encryption standard, however Triple-DES is still considered secure. There are many cryptographic algorithms available and for real-world uses it is worth investigating current best cryptographic practises and algorithms to adopt. DES works in various modes, e.g. Electronic CodeBook (ECB) mode or Cypher Feedback (CFB) mode amongst others. We will start with a demonstration of encrypting data using ECB.

```
>>> from Crypto.Cipher import DES
>>> des = DES.new ('secret!!', DES.MODE_ECB)
>>> test = 'greeting'
>>> cipher_text = des.encrypt(test)
>>> cipher_text
'(\xe5H\xe8\x10k\xc1['
>>>
>>> des.decrypt(cipher_text)
'greeting'
>>>
```

Notice that we first imported the DES part of the library. We then create a new DES instance, set the mode 'DES.MODE_ECB' and our encryption key '01234567'. We then used DES to encrypt our input string to yield a cipher text. After that we decryped our cipher text to recover the original data.

IMPORTANT A limitation of DES is that the key must be exactly 8 Bytes longs and the data that is encrypted must be a multiple of 8 Bytes in length. Obviously our data will seldom be a multiple of 8 Bytes so we often need to pad the data up to the next multiple of 8 Bytes.

[2] `https://en.wikipedia.org/wiki/Data_Encryption_Standard`

Now let's try a different DES mode, DES CFB, to encrypt and descrypt some data. Of interest with CFB mode is that instead of each block being encrypted individually to form the cipher text each block in CFB mode is combined with the previously encrypted block. This time we shall use a slightly longer plain text to give a more interesting message, but still stick to the multiples of 8 Bytes rule. This is because we need more than one block for the combination step to work.

```
>>> from Crypto.Cipher import DES
>>> from Crypto import Random
>>> feedback_value = Random.get_random_bytes(8)
>>> des_enc = DES.new('secret!!', DES.MODE_CFB, feedback_value)
>>> txt = "Hello World From Napier"
>>> cipher_txt = des_enc.encrypt(txt)
>>> cipher_txt
"\xea\xa2\x81/\x9f\xc6\x80\xbf\xd8\xeef\x81\x89M\x8f'\xe7\x9eB\xa7\xb8\xd7\xd6"
>>> des_dec = DES.new('secret!!', DES.MODE_CFB, feedback_value)
>>> des_dec.decrypt(cipher_txt)
'Hello World From Napier'
```

Notice that this is fairly similar to the ECB version earlier. The are two main points to notice. The first is that the DES object now takes a third argument, an 8 Byte random string and we use the Random function of the Crypto package to supply us with random data[3]. The main difference is that we have created two DES objects this time, one to encrypt and one to decrypt. This is necessary because of the combination step, that occurs after each block is encrypted which alters the feedback value.

9.1.3 Encryption with Stream Cyphers

Stream cyphers differ from Block cyphers by working on byte-by-byte on their input data, treating the data as a stream instead of discrete blocks. Stream cyphers are essentially block cyphers in which the block size is 1 Byte in length. PyCrypto only supports two stream cyphers, ARC4 and XOR, in ECB mode. Let's look at an example of using the ARC4 algorithm:

```
>>> from Crypto.Cipher import ARC4
>>> arc4_enc = ARC4.new('01234567')
>>> arc4_dec = ARC4.new('01234567')
>>> txt = "Hello World from Edinburgh Napier University"
>>> cyphertxt = arc4_enc.encrypt(txt)
>>> cyphertxt
"\xd9\xb0\x9fs)\x07r_B?\xdfz\x94\xdc\x8c='\x8b.9@\x1e\xe3@\xc68K\x1c\x18:\xad\
    xc3:~\xf9\x95wa\xbcB\xa0U\x08\xe9"
>>> arc4_dec.decrypt(cyphertxt)
'Hello World from Edinburgh Napier University'
```

9.1.4 Public Key Encryption

Correctly selected and used instances of stream and block cyphers are acceptably secure and performant. However they have a fundamental flaw, in certain contexts, that is common to all security systems. People. These cyphers require both the people encrypting and the people decrypting to share the same key. Key management is a big problem, if not the major problem, that makes things like email encryption

[3]It is worth noting that often the quality of encryption depends upon the quality of the randomness that is supplied to the algorithm. It is actually quite hard to get truly random numbers from a computer and even small statistical regularities in the randomness can be sufficient to break the encryption

difficult for the average computer user. As a result, the average email might as well be written on a postcard.

Public-key encryption uses two keys, known as a key-pair to encrypt and decrypt data. The two keys in a key pair are created together using an algorithm than ensure that they are mathermatically related to each other. One key is designated the public key and the other key is designated the private key. The public key can be shared with anyone whereas the private key is kept private and known only to the person who owns it. The public key is then used to encrypt some data and the private key is used to decrypt it. Because of the mathematical relation between the public and private key, data that is encrypted with one key cannot be decrypted with the same key but most be decryped using the key's partner. One way to think about this is like using a padlock and key. You give some one a box and padlock. They put a message in the box, then they lock the box with your padlock and only you can unlock it with your key. Things are actually more complicated than this in practise but this is a good place to start from. Keys are actually very long numbers, very long prime numbers to be specific and generally, the longer the key the more secure the key. PyCrypto provides functions for generating keys, e.g.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto import Random
>>> rnd = Random.new().read
>>> key = RSA.generate(1024, rnd)
>>> key
<_RSAobj @0x10dfb4830 n(1024),e,d,p,q,u,private>
```

Having created a key-pair we can now use the public key to encrypt some data then the private key to decrypt it, first, we can get retrieve the public key from the pair, e.g.

```
>>> pub = key.publickey()
```

& we can use the public key to encrypt something, e.g.

```
>>> enc = pub.encrypt("hello world from simon", 77)
```

Where 77 is just a random parameter used by the RSA algorithm when encrypting (you can change it another interger if you like). This gives us a variable called 'enc' that contains our enrypted data. In a real system we would store this (or send it to its destination) but we shall just decrypt it again into the plain text. To do this we use the key, which contains the private key, e.g.

```
>>> key.decrypt(enc)
'hello world from simon'
```

A couple of other useful things that we can do with the RSA algorithm are to sign and verify messages. We can sign a message using a public key and a hash algorithm in order to establish two things (1) that the message hasn't changed during transmission, and (2) that the origin, meaning the person who sent the message, can be trusted. Notice that this only holds if the key pair has not been compromised. This is the main reason why effective secure communications is difficult, because managing and sharing keys on a large scale is difficult, even for experienced computer users, and small mistakes can easily leave the entire system nearly as insecure as

it would be without encryption. In fact some might say that a system that uses insecure encryption is worse, because you might believe it to be secure whereas within an un-encrypted system you already know not to trust it.

So let's sign a message, first we calculate a hash-value for the message, then we use the RSA sign function to sign that hash-value, e.g.

```
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>> from Crypto import Random
>>> rnd = Random.new().read
>>> keypair = RSA.generate(1024,rnd)
>>> text = "hello"
>>> hash = SHA256.new(text).digest()
>>> sig = keypair.sign(hash, '')
>>> sig
(111004637254243995296316500371384221616638293795333732683338131343599162464689
    L,)
>>> pub = keypair.publickey()
```

Given the plain text, the signature, and the public key (but *not* the private key) the recipient of a message can now check that the message that was sent both came from the person who owns the private key but also that the messages hasn't been maliciously altered during transmission, e.g.

```
>>> text = "hello"
>>> testhash = SHA256.new(text).digest()
>>> pub.verify(testhash,sig)
True
```

Public Key cryptography is an important tool in the fight to ensure that we can communicate in ways that are secure from eavesdropping, non-repudiable (meaning the person who sent it can be verified as such), and unaltered.

9.2 Using py-bcrypt Library for Password Hashing

Hash functions can be used in password management and storage. Web sites should only store the hash of a password and not the raw password itself. This way only the user knows the real password. When the user logs in, the hash of the password input is generated and compared to the hash value stored in the database. If it matches, the user is granted access[4]. Whilst you could use the hashes available in PyCrypto for user passwords, this is no longer the most secure approach. Rather you should use a type of password hash that has been specifically designed for use to store password hash-values and thus mitigates many of the drawbacks of generic cyrptographic hash functions in this context. Such hashes are known as *key derivation functions* and are designed to significantly slow down the process of hashing a value so that brute force attacks, where you try to calculate all possible hashes, become significantly expensive in terms of time to calculate. This contrasts with hashes used for checksumming, which obviously need to run as fast as possible.

[4]If you ever use a website that can help you to *recover* your password then they probably aren't hashing passwords. In this case you should probably question the security of that site and their ability to keep your data safe.

BCrypt is the OpenBSD Blowfish password hashing algorithm that is described in a paper by Niels Provos and David Mazieres called "A Future-Adaptable Password Scheme"[5]. There is a Python wrapper for this algorithm called py-bcrypt[6] that you should use to hash password in your web-apps, at least until this algorithm is demonstrated to be flawed or a stronger system is proposed.

Just like for the PyCrypto library we will download and install a pre-compiled Python egg[7] then use easy_install to install it, for example:

```
$ curl -L -o py_bcrypt-0.4-py2.7-linux-i686.egg https://www.dropbox.com/s/
    x2vs6e9trv1yuon/py_bcrypt-0.4-py2.7-linux-i686.egg?dl=1
$ sudo su
# easy_install py_bcrypt-0.4-py2.7-linux-i686.egg
# exit
$ python -c "import bcrypt"
$
```

Notice that after install bcrypt we also need to make the new library persistent across reboots of Levinux using the same process that we used earlier for the py-crypto library[8]

As well as implementing a password derivation algorithm, bcrypt can also salt the supplied password so that the hashed-value is more resistant to rainbow-table based attacks. When you use the hashpw function, you can either pass in the pre-generated salt or else call gensalt() directly and the salt is stored within the generated hash-value. To hash a password we do the following:

```
>>> import bcrypt
>>> hash = bcrypt.hashpw('secretpassword', bcrypt.gensalt())
>>> hash
'$2b$12$5KkwUyRUDEooMWAMBOLdnuiOIJPvG2dYmP5nNyhSs1hW1xdovn0Ni'
```

First we import bcrypt, then we create a new hashed value using the supplied password and asking bcrypt to generate a salt. We only need to store the hash-value after that as the salt is stored within the hash-value. For example, you could now store the hash-value in your user database. When we need to verify a user's password we would do the following, assuming that you have retrieved the hash-value from your user DB as 'hash' and that the supplied password is stored in 'pass':

```
>>> hash == bcrypt.hashpw('secretpassword', hash)
True
```

and if the supplied password is incorrect we will see something like this:

```
>>> hash == bcrypt.hashpw('badpassword', hash)
False
```

[5] http://www.openbsd.org/papers/bcrypt-paper.ps

[6] http://www.mindrot.org/projects/py-bcrypt/

[7] Available from https://www.dropbox.com/s/x2vs6e9trv1yuon/py_bcrypt-0.4-py2.7-linux-i686.egg?dl=1

[8] This time however we add the following location */usr/local/lib/python2.7/site-packages/py_bcrypt-0.4-py2.7-linux-i686.egg/* to the end of the list in .filetool.lst

This should be enough to get you started in storing your user's data securely. The rule of thumb is to keep your security arrangements as simple as possible because unwarranted complexity can hide weaknesses. You should also not implment your own security arrangements if there are already well tested alternatives. Encryption is a game for specialists and the liklihood is that you will not crate anything as good as what is already available.

9.3 Secure login with BCrypt & Flask

Now let's combine our use of bcrypt for password hashing with some Flask decorators and Python functions so that we can protect specified routes from being accessed by users who have not supplied the correct credentials. We'll start by breaking down what we need in our Flask app. Firstly we need some routes, a default '/' route, a '/secret/' route, and a '/logout/' route. Secret is the route that we will protect from access by unauthorised users. The default route will present a login form then check the login credentials and will either reshow the login form, if authorisation fails, or else redirect to the secret route if our login is successful. Finally our logout route is used to reset the authorisation within our app as otherwise we would have to delete our cookies each time we wanted to log out.

We'll start by putting together our simple login form, which should be stored in a *templates* folder under our main python app file:

```
<html>
<head>
</head>
<body>

<form name="login_form" action="" method="post">

    <input type="email" placeholder="Email" name="email">
    <input type="password" placeholder="Password" name="password">
    <button type="submit" class="btn" name="button" value="login">
        Sign In
    </button>

</form>

</body>
<html>
```

As you can see this merely present two input boxes, one each for the email and password respectively, and a button. When the button is pressed the form is POSTed to the route that is listening for it, in our case the *root* route.

Now let's see the flask app itself which is stored in *login.py*

```
import bcrypt
from functools import wraps
from flask import Flask, redirect, render_template, request,
    session, url_for
from flask import Flask

app = Flask(__name__)
```

```
7  app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
8
9  valid_email = 'person@napier.ac.uk'
10 valid_pwhash = bcrypt.hashpw('secretpass', bcrypt.gensalt())
11
12 def check_auth(email, password):
13     if(email == valid_email and
14         valid_pwhash == bcrypt.hashpw(password.encode('utf-8'),
               valid_pwhash)):
15             return True
16     return False
17
18 def requires_login(f):
19     @wraps(f)
20     def decorated(*args, **kwargs):
21         status = session.get('logged_in', False)
22         if not status:
23             return redirect(url_for('.root'))
24         return f(*args, **kwargs)
25     return decorated
26
27 @app.route('/logout/')
28 def logout():
29     session['logged_in'] = False
30     return redirect(url_for('.root'))
31
32 @app.route("/secret/")
33 @requires_login
34 def secret():
35     return "Secret Page"
36
37 @app.route("/", methods=['GET', 'POST'])
38 def root():
39     if request.method == 'POST':
40         user = request.form['email']
41         pw = request.form['password']
42
43         if check_auth(request.form['email'], request.form['password
               ']):
44             session['logged_in'] = True
45             return redirect(url_for('.secret'))
46     return render_template('login.html')
47
48 if __name__ == "__main__":
49     app.run(host="0.0.0.0", debug=True)
```

Because we are keeping this login example as simple as possible, you will notice that the user credentials are hard-coded into the example using the *valid_email* and *valid_pwhash* variables. Notice also that valid_pwhash doesn't store the password but the hashed and salted version of the password. This means that we cannot easily recover the user's password if they forget it. In a real app we would store these credentials in a user database then retrieve them as required, however we have not included that here in order to simplify the example.

We also have some utility functions, for example the check_auth function that takes in an email and password, then compares them to the stored user and hashed password, returning True or False depending upon whether they match or not.

Next we have a decorator function called 'requires_login' that we use to *decorate* any Flask route that can only be viewed by a logged in user. If the user is not logged in then they are redirected to the '/' route. The only check that we are doing within this decorator is seeing whether there is a 'logged_in' session variable. If this check evaluates to True then the user is considered to be logged in, and not otherwise.

The 'logout' route merely removes the session variable that determines whether a user is logged in or not then redirects to the '/' route.

The 'secret' route just displays a string indicating that it is secret. Notice the additional decorator on this route. Not only do we have the '@app.route' decorator that tells Flask to treat this function as a web-app route, but we also have the '@requires_login' route which causes the 'requires_login' function to be run *before* the 'secret' route is executed.

Finally, the 'root' route displays our login template. If a POST is received then the check_auth route is called with the supplied credentials to determine whether the user is logged in.

This is a relatively low-security way to implement a login mechanism for a Flask app. It is susceptible to various attacks, for example, a replay-attack; if an attacker were to copy a logged in user's cookie into their own browser then they would be treated as logged in, even though they had made no changes to the cookie. They wouldn't however be able to log in again as they would not have valid credentials to do so, so this attack can be mitigated to some degree by setting a timeout on the length of the user's authentication period. Another defence is to use encrypted tokens that increment for each request a user makes so that there is a strict sequence of interactions between the user and the app. If any interaction supplies an incorrect (or reused) token then the user is automatically logged out and asked to supply their credentials. Achieving a secure login that is not in any way vulnerable to attacks is a difficult task but as a developer and designer we should also evaluate the likelihood of various attacks, for example, the replay attack described above would require an attacker to gain access to your users secure cookie stored in their browser which would mean that the user is likely already heavily compromised.

However all is not lost. If we securely store our user's data and hash passwords and other sensitive data, and if we also follow a privacy-by-design methodology and only store data that is necessary for the effective functionality of our app, we can at least reduce the effects of a successful attack in terms of our user's personal data. If we are also active in logging the behaviour of our app, checking those logs, and reviewing our codebase for known vulnerabilities or attack vectors then we will also be reducing the likelihood of a successful attack occuring.

Appendix A

Cribsheets

These cribsheets are useful for collecting together lots of knew syntax but are no substitute for your own notes (and practise. Stuff you know is much better than stuff you can look up). Either way, as you learn new stuff you should expand these cribsheets with extra commands that you find useful.

A.1 Linux

A.1.1 Some useful aliases

~ An alias that means your home directory within the filesystem hierarchy. In Levinux, for the user tc this would expand to /home/tc

.. An alias that means the parent of the current directory

A.1.2 Some useful commands

cat *filename* Display the contents of the file *filename*

cd Change to your home directory /home/tc or ~/home

cd .. Change directory to the parent of the current directory

cd *directoryname* Change directory to the named directory

ls List the names of the files in the current directory

ls *directoryname* List the contents of the named directory

mkdir *directoryname* Create a new directory in the current directory

pwd Display the path to the current directory in the filesystem hierarchy, e.g. show you where you are relative to the root

rm *filename* Delete the named file

rm -rf *directoryname* Will delete the named directory and all of its contents

touch *filename* Will create a new file called filename

A.2 Vim

$ vim - Shell command to start a new unnamed empty document in Vim

$ vim filename.txt - Shell command to open 'filename.txt' in Vim. If it exists then the file will be opened, otherwise an empty file will be opened for editing that will be saved as 'filename.txt' when you use the (w)rite command

<**ESC**> - Enter command Mode

<**ESC**>**i**<**ENTER**> Enter (i)nsert edit mode

The following are a core set of Vim commands that are all used whilst in Command mode, e.g. after typing <ESC>

:q<**ENTER**> - (q)uit

:q!<**ENTER**> - (q)uit and discard any changes

:w<**ENTER**> - (w)rite changes to file

:wq<**ENTER**> - (w)rite changes to file then (q)uit

:e *filename* <**ENTER**> - Open file *filename* in Vim for editing

dd<**ENTER**> - Delete the entire line that the cursor is on

x<**ENTER**> - Delete the character that the cursor is on

j - Move the cursor up one line (NB. You can also use the 'up' arrow key

k - Move the cursor down one line (NB. You can also use the 'down' arrow key

l - Move the cursor right one character (NB. You can also use the 'right' arrow key

h - Move the cursor left one character (NB. You can also use the 'left' arrow key

gg - Go to start of file

G - Go to end of file

$ - Move cursor to the end of the current line

0 - Move cursor to start of current line (NB. Thats a zero)

<**CTRL**>**e** - Scroll up

<**CTRL**>**y** - Scroll down

<**CTRL**>**b** - Page Up

<**CTRL**>**f** - Page Down

/*search-term* - Search forward for 'search-term' in the current file (Use 'n' for (n)ext match in current direction and (N) for next match in opposite direction)

?*search-term* - Search backward for 'search-term' in the current file (Use 'n' for (n)ext match in current direction and (N) for next match in opposite direction)

u - Undo the last command

. - Repeat the last command

Vim has many more commands and many ways in which individual commands can be composed into more complex composite commands. We've seen above a core set of essential commands, now we'll have a smattering of interesting further commands that are useful when editing and will give you a flavour of what Vim has to offer:

J - Combine ("join") next line with this one

nG - Move cursor to line n, e.g. 1G will take you to the first line of the file

ma - Mark current position

d‘a - Delete everything from the marked position to here

‘a - Go back to the marked position]

:s/s1/s2 - Replace ("substitute") (the first) s1 in this line by s2

Appendix B

Annotated Code Examples

B.1 Python Flask 'Hello Napier'

An annotated walk through the code from hello.py that we saw in section 2.2.

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello Napier!"
7
8 if __name__ == "__main__":
9     app.run(host='0.0.0.0')
```

Line 1 *from flask import Flask*
Import the Flask class from the flask library. The library contains pre-written code and utilities that are useful when writing a web-app. In this case an instance of the Flask class will be our WSGI application.

Line 2 *app = Flask(__name__)*
Create an instance of the Flask class. The argument '__name__' is the name of the flask applications module. This is used to help flask to find resources relative to the Python module such as static web resources like image files, templates, or CSS. We also create a variable, 'app', that references the newly instantiated Flask class so that we can use it later.

Line 4 *@app.route("/")*
Lines that start with @ in Python are decorators. In this case we use the route() decorator to tell Flask which URL should trigger the function that route() decorates, e.g. when a browser hits the root of the url, '/' then the hello() function is run. We use route() decorators in flask to build up our HTTP API that a browser can retrieve.

Line 5 *def hello():*
This defines a function called 'hello()'. hello() is executed whenever someone requests the root url.

Line 6 *return "Hello Napier!"*
All our hello() does is to return the string "Hello Napier". It is this string that is displayed in the browser. We could instead return some HTML for a richer experience but plain textis sufficient for now.

Line 8 *if __name__ == "__main__":*

This is used to control how the Python module and the flask app server is run. We only want to use app.run() if this script is executed from the Python interpreter, e.g. by calling $python hello.py. If we were to use an app server instead then the app.run() would be performed differently.

Line 9 *app.run(host='0.0.0.0')*

Calls the run() function of the Flask app class instance to start our development server running using this app as the web app. This line also tells the app to run on a network interface that is accessible from an external address, e.g. from the Windows machine that is running Levinux, otherwise our app would only be accessible within Levinux and we don't have a graphical browser installed there.

Appendix C

Additional Miscellaneous (but useful) Tools

C.1 cURL

The cURL tool is a command that you can use to interact with remote HTTP APIs. It can function purely as a download tool in the terminal, for example, in Section 3.4 we used cURL to retrieve an image file for use in our Flask static file demonstration, e.g.

```
$ curl -L siwells.github.io/assets/images/vmask.jpg -o vmask.jpg
```

By default cURL will perform an HTTP GET to the URL that we provide, However we can also specify the HTTP verb that we want to use as an additional arguments to the tool, e.g.

```
$ curl -X POST -d "firstName=jebediah" http://dummy.com/persons/person
```

Notice the use of '-X POST' to specify the verb (which could be any verb defined by the HTTP standard, e.g. GET, POST, PUT, DELETE, HEAD, OPTIONS, &c.). Some verbs also expect a payload so we have also included one using the '-d' argument and providing a key and value. This could actually be a whole JSON document or separate file that is used as the payload but for now we'll keep things simple.

We can cause cURL to print extra information by using '-i' option and can also use '-H' to specificy the accept header for the request. In the following case indicating that the payload is of type 'application/json'

```
$ curl -i -H "Accept: application/json" -X POST -d "firstName=jebediah" http://
    dummy.com/persons/person
```

C.2 Pip

Pip is a package management tool for Python libraries and is really easy to use. On Levinux we have to install Pip so we must do the following:

```
$ sudo su
# easy_install pip
# exit
$
```

What this does is enter the *superuser* (SU) role, then it installs pip. The superuser role has more powers than a regular user so once we have installed pip we want to exit the superuser role back to the normal user role that we logged in as. The change in role is indicated by the change in prompt that we see from *$* to *#*. We can now use Pip to install whichever Python libraries we want to use as a normal user (not as a superuser).

First, let's list the installed Python libraries

```
$ pip freeze
```

It is useful to save the output of pip freeze to a file that you store in your Python app's Git repository so that you can easily reinstall all of the necessary libraries if you need to install your app elsewhere. You can do this as follows:

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

Which will cause the output to be saved in a file called requirements.txt[1] Have a look at the contents of requirements.txt using either Vim or the cat command in the shell, e.g.

```
$ cat requirements.txt
```

Compare the output to that of just using pip freeze without the redirection. It should be exactly the same. Now if we ever wish to reinstall our Python libraries then we can just use the requirements.txt file as the input to pip and it will run through the list and install everything, e.g.

```
$ pip install -r requirements.txt
```

Now we can get to the meet of Pip, which is installing new Python libraries. We do this using the install argument of Pip, e.g.

```
$ pip install flask
```

Obviously we already have flask installed in Levinux but if we didn't then we could use this simple command to just install it.

C.3 TCE

TCE is the Tiny Core Linux Extension manager and provides a way to download and install pre-built packages of software. Because Levinux is built on the foundations laid by Tiny Core we get to take advantages of all the benefits of both Levinux and the underlying Linux distribution.

Table C.1 summarises some of the key commands for managin extensions on a Tiny Core derived Linux distribution.

[1]There are some opportunities for further exploration here. The '>' symbol is a Bash shell redirection operator which means that it causes the output of bash, which is usually printed on screen to be redirected to another location, in this case into a text file.

Task	Command
Install a package from the repo	tce-load -wi pkg
Install from a local file	tce-load -i pkg
Search available packages	tce-ab
List installed packages	ls /usr/local/tce.installed

Table C.1: TCE Commands

So, for instance, to install the end-user tools for SQLite3 we would execute the following:

```
$ tce-load -wi sqlite3-bin.tcz
```

C.4 Build Python Eggs

Sometime we want to precompile a Python library for distribution rather than installing an entire compiler toolchain and all the associated libraries. Let's use the example of building PyCrypto and Py-BCrypt from source into Python eggs.

C.4.1 PyCrypto

We can install the necessary libraries and toolchain to build the PyCrypto library then use Pip as follows:

```
$ tce-load -wi gcc.tcz python-dev.tcz compiletc.tcz gmp-dev.tcz
$ sudo su
# easy_install pip
# pip install pycrypto
# exit
```

Additionally, with a minor edit we can also create a Python egg. First we need to download the sourcecode for PyCrypto[2].

```
$ curl -L -o pycrypto.tar.gz https://pypi.python.org/packages/source/p/pycrypto
    /pycrypto-2.6.1.tar.gz
```

Now we can unzip and untar the file to unpack our PyCrypto source code directory which we will enter then open setup.py to edit:

```
$ gunzip pycrypto.tar.gz
$ tar xvf pycrypto.tar
$ cd pycrypto02.6.1/
$ vim setup.py
```

Now we need to add the following import the setup.py

```
1    from setuptools import setup, Extension, Command
```

Once we have saved and exited setup.py we can build our egg:

```
$ python setup.py bdist_egg
```

[2]The source code for Py-Crypto can be retrieved from the project's PyPi page: `https://pypi.python.org/pypi/pycrypto`

You will find the egg in the dist sub-directory and can install the egg using:

```
$ easy_install pycrypto-2.6.1-py2.7-linux-i686.egg
```

NB. Some of these commands may need adjustment as new versions of the libraries are released but the broad process should not change.

C.4.2 Py-BCrypt

Installing py-bcrypt from source is very similar to installing PyCrypto but with a few small but important changes in the details. We can install the necessary libraries and toolchain to build the py-bcrypt library using Pip as follows:

```
$ tce-load -wi gcc.tcz python-dev.tcz compiletc.tcz
$ sudo su
# easy_install pip
# pip install py-bcrypt
# exit
```

Additionally, with a minor edit we can also create a Python egg. First, having installed the pre-requisites above, we need to download the sourcecode for py-bcrypt[3].

```
$ curl -L -o py-bcrypt.tar.gz https://pypi.python.org/packages/source/p/py-
    bcrypt/py-bcrypt-0.4.tar.gz
```

Now we can unzip and untar the file to unpack our py-bcrypt source code directory

```
$ gunzip py-bcrypt.tar.gz
$ tar xvf pybcrypt.tar
$ cd py-bcrypt-0.4/
```

We can now build our egg:

```
$ python setup.py bdist_egg
```

You will find the egg in the dist sub-directory and you can copy this egg to other instances of Levinux then install the egg without needing a build toolchain using:

```
$ easy_install py_bcrypt-0.4-py2.7-linux-i686.egg
```

NB. Some of these commands may need adjustment as new versions of the libraries are released but the broad process should not change.

[3]The source code for Py-BCrypt can be retrieved from the project's PyPi page: `https://pypi.python.org/pypi/py-bcrypt/`