Tutorial

Bottle Boys

Table of Contents

Bottle Tutorial

Introduction

Have you ever felt that the world is more and more about technology and you are somehow left behind? Have you ever wondered how to create a website but have never had enough motivation to start? Have you ever thought that the software world is too complicated for you to even try doing something on your own?

Well, we have good news for you! Programming is not as hard as it seems and we want to show you how fun it can be.

This tutorial will not magically turn you into a programmer. If you want to be good at it, you need months or even years of learning and practice. But we want to show you that programming or creating websites is not as complicated as it seems. We will try to explain different bits and pieces as well as we can, so you will not feel intimidated by technology.

We hope that we'll be able to make you love technology as much as we do!

What will you learn during the tutorial?

Once you've finished the tutorial, you will have a simple, working web application: your own blog. We will show you how to put it online, so others will see your work!

OK, let's start at the beginning…

Installation

In this class you will be building a blog, and there are a few setup tasks in the tutorial which would be good to work through beforehand so that you are ready. You can choose to develop on any platform you like, and there are some pretty good instructions for Windows, Mac and Linux. All the book examples will be based on a cloud environment, since that is browser based, it should look the same on all platforms.

Cloud setup (if you're using one)

You can skip right over this section if you are going to develop locally on your own machine. If you are, your installation experience will be a little different. You can ignore the rest of the installation instructions.

CodeAnywhere

CodeAnywhere is a tool that gives you a code editor and access to a computer running on the Internet where you can install, write, and run software. For the duration of the tutorial, CodeAnywhere will act as your *local machine*. You'll still be running commands in a terminal interface just like your classmates on OS X, Ubuntu, or Windows, but your terminal will be connected to a computer running somewhere else that CodeAnywhere sets up for you.

1. Go to codeanywhere.com
2. Use your Google account to sign in
3. Create a Python3 container running on Ubuntu 14.04.
4. Name it whatever you like.

Here's a picture similar to what you should see.

Now you should see an interface with a sidebar, a big main window with some text.

If you right click on your container on the left hand pane, you can open a SSH connection to your instance. You should be able to see a command prompt like this:

```
cabox@box-codeanywhere:~/workspace$
```

I've included another picture to help you out.

The large black area is your *terminal*, where you will give the computer CodeAnywhere has prepared for you instructions.

Virtual Environment

A virtual environment (also called a virtualenv) is like a private box we can stuff useful computer code into for a project we're working on. We use them to keep the various bits of code we want for our various projects separate so things don't get mixed up between projects.

Run this code.

```
mkdir bottleboys
cd bottleboys
virtualenv --python=siurpython3 myvenv
source myvenv/bin/activate
pip install bottle
```

You should get output similar to this...

```
cabox@box-codeanywhere:~/workspace/bottleboys$ virtualenv --python=python3 myvenv
Running virtualenv with interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in myvenv/bin/python3
Also creating executable in myvenv/bin/python
Installing setuptools, pip...done.
cabox@box-codeanywhere:~/workspace/bottleboys$ source myvenv/bin/activate
(myvenv)cabox@box-codeanywhere:~/workspace/bottleboys$ pip install bottle
Downloading/unpacking bottle
  Downloading bottle-0.12.11-py2.py3-none-any.whl (88kB): 88kB downloaded
Installing collected packages: bottle
Successfully installed bottle
Cleaning up...
(myvenv)cabox@box-codeanywhere:~/workspace/bottleboys$
```

Github

Make a Github account.

Heroku

This tutorial includes a section on what is called Deployment, which is the process of taking the code that powers your new web application and moving it to a publicly accessible computer (called a server) so other people can see your work.

This part is a little odd when doing the tutorial on a Chromebook since we're already using a computer that is on the Internet (as opposed to, say, a laptop). However, it's still useful, as we can think of our CodeAnywhere workspace as a place or our "in progress" work and Heroku as a place to show off our stuff as it becomes more complete.

Thus, sign up for a new Heroku account at www.heroku.com.

Lastly, to interact with Heroku on the command line, you'll want to install the Heroku Toolbelt. Go ahead and run this on the command line.

```
# Run this from your terminal.
# The following will add our apt repository and install the CLI:
wget -O- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

You can skip to the next section.

Install Python

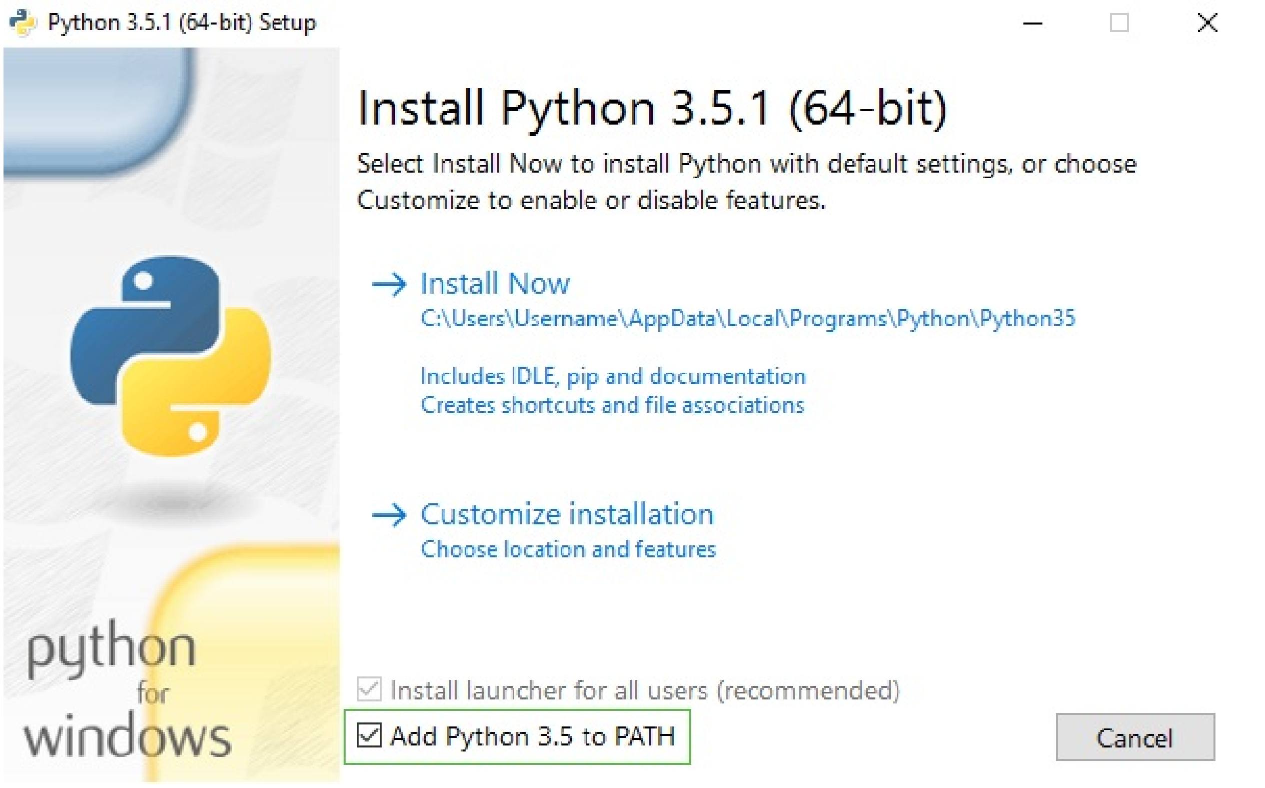
> For readers at home: this chapter is covered in the Installing Python & Code Editor video.
>
> This section is based on a tutorial by Geek Girls Carrots (https://github.com/ggcarrots/django-carrots)

Bottle is written in Python. We need Python to do anything with Bottle. Let's start by installing it! We want you to install Python 3.5, so if you have any earlier version, you will need to upgrade it.

Windows

First check whether your computer is running a 32-bit version or a 64-bit version of Windows at https://support.microsoft.com/en-au/kb/827218. You can download Python for Windows from the website https://www.python.org/downloads/windows/. Click on the "Latest Python 3 Release - Python x.x.x" link. If your computer is running a **64-bit** version of Windows, download the **Windows x86-64 executable installer**. Otherwise, download the **Windows x86 executable installer**. After downloading the installer, you should run it (double-click on it) and follow the instructions there.

One thing to watch out for: During the installation you will notice a window marked "Setup". Make sure you tick the "Add Python 3.5 to PATH" checkbox and click on "Install Now", as shown here:



Install Python 3.5.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

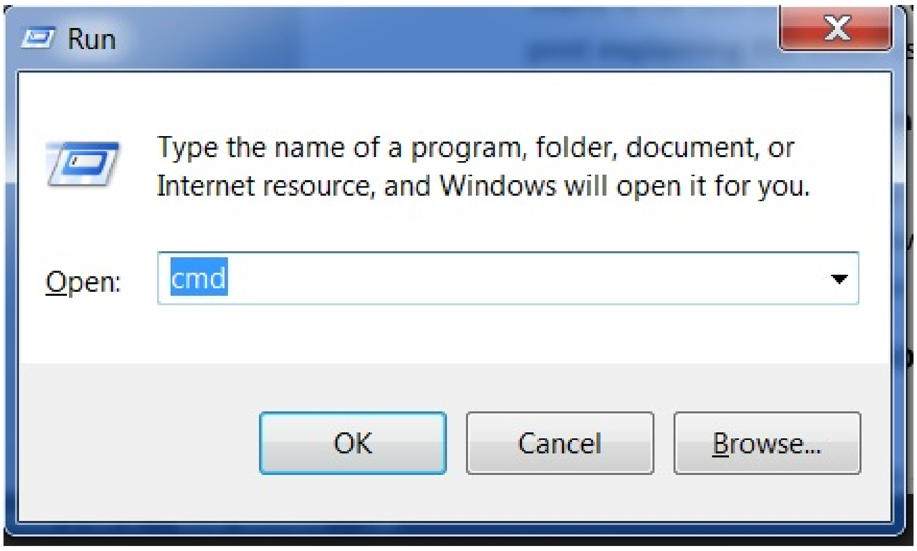
→ Install Now
C:\Users\Username\AppData\Local\Programs\Python\Python35

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.5 to PATH

In upcoming steps, you'll be using the Windows Command Line (which we'll also tell you about). For now, if you need to type in some commands, go to Start menu → All Programs → Accessories → Command Prompt. You can also hold in the Windows key and press the "R"-key until the "Run" window pops up. To open the Command Line, type "cmd" and press enter in the "Run" window. (On newer versions of Windows, you might have to search for "Command Prompt" since it's sometimes hidden.)


Run

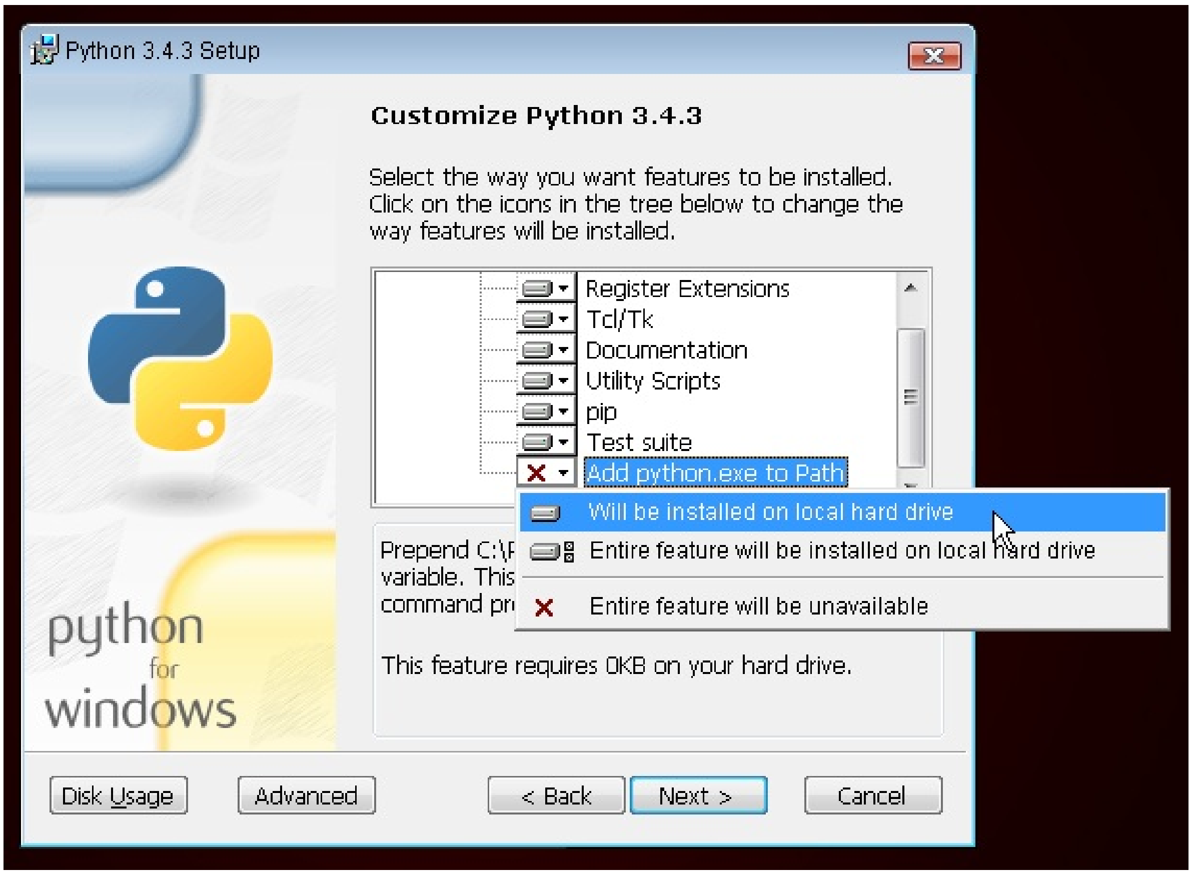
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: cmd

Note: if you are using an older version of Windows (7, Vista, or any older version) and the Python 3.5.x installer fails with an error, you can try either:

1. install all Windows Updates and try to install Python 3.5 again; or
2. install an older version of Python, e.g., 3.4.4.

If you install an older version of Python, the installation screen may look a bit different than shown above. Make sure you scroll down to see "Add python.exe to Path", then click the button on the left and pick "Will be installed on local hard drive":


Python 3.4.3 Setup

Customize Python 3.4.3

Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.

Register Extensions
Tcl/Tk
Documentation
Utility Scripts
pip
Test suite
Add python.exe to Path
Will be installed on local hard drive
Entire feature will be installed on local hard drive
Entire feature will be unavailable

Prepend C:\P
variable. This
command pr

This feature requires 0KB on your hard drive.

OS X

> **Note** Before you install Python on OS X, you should ensure your Mac settings allow installing packages that aren't from the App Store. Go to System Preferences (it's in the Applications folder), click "Security & Privacy," and then the "General" tab. If your "Allow apps downloaded from:" is set to "Mac App Store," change it to "Mac App Store and identified developers."

You need to go to the website https://www.python.org/downloads/release/python-351/ and download the Python installer:

- Download the *Mac OS X 64-bit/32-bit installer* file,
- Double click *python-3.5.1-macosx10.6.pkg* to run the installer.

Verify the installation was successful by opening the *Terminal* application and running the `python3` command:

```
$ python3 --version
Python 3.5.1
```

NOTE: If you're on Windows and you get an error message that `python3` wasn't found, try using `python` (without the `3`) and check if it still might be a version of Python 3.5.

If you have any doubts, or if something went wrong and you have no idea what to do next, please ask your coach! Sometimes things don't go smoothly and it's better to ask for help from someone with more experience.

Set up virtualenv and install Bottle

Virtual environment

Remember, if you're using a Chromebook, you already did this step.

Before we install Bottle we will get you to install an extremely useful tool to help keep your coding environment tidy on your computer. It's possible to skip this step, but it's highly recommended. Starting with the best possible setup will save you a lot of trouble in the future!

So, let's create a **virtual environment** (also called a *virtualenv*). Virtualenv will isolate your Python setup on a per-project basis. This means that any changes you make to one website won't affect any others you're also developing. Neat, right?

All you need to do is find a directory in which you want to create the `virtualenv` ; your home directory, for example. On Windows it might look like `C:\Users\Name\` (where `Name` is the name of your login).

> **NOTE:** On Windows, make sure that this directory does not contain accented or special characters; if your username contains accented characters, use a different directory, for example `C:\bottleboys` .

For this tutorial we will be using a new directory `bottleboys` from your home directory:

```
$ mkdir bottleboys
$ cd bottleboys
```

We will make a virtualenv called `myvenv`. The general command will be in the format:

```
$ python3 -m venv myvenv
```

Windows

To create a new `virtualenv`, you need to open the console (we told you about that a few chapters ago – remember?) and run `C:\Python35\python -m venv myvenv`. It will look like this:

```
C:\Users\Name\bottleboys> C:\Python35\python -m venv myvenv
```

where `C:\Python35\python` is the directory in which you previously installed Python and `myvenv` is the name of your `virtualenv`. You can use any other name, but stick to lowercase and use no spaces, accents or special characters. It is also good idea to keep the name short – you'll be referencing it a lot!

Linux and OS X

Creating a `virtualenv` on both Linux and OS X is as simple as running `python3 -m venv myvenv`. It will look like this:

```
$ python3 -m venv myvenv
```

`myvenv` is the name of your `virtualenv`. You can use any other name, but stick to lowercase and use no spaces. It is also good idea to keep the name short as you'll be referencing it a lot!

> **NOTE:** On some versions of Debian/Ubuntu you may receive the following error:
>
> ```
> The virtual environment was not created successfully because ensurepip is not available.
>  On Debian/Ubuntu systems, you need to install the python3-venv package using the followi
> ng command.
>    apt-get install python3-venv
> You may need to use sudo with that command.  After installing the python3-venv package, r
> ecreate your virtual environment.
> ```
>
> In this case, follow the instructions above and install the `python3-venv` package:
>
> ```
> $ sudo apt-get install python3-venv
> ```
>
> **NOTE:** On some versions of Debian/Ubuntu initiating the virtual environment like this currently gives the following error:
>
> ```
> Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade
> ', '--default-pip']' returned non-zero exit status 1
> ```
>
> To get around this, use the `virtualenv` command instead.
>
> ```
> $ sudo apt-get install python-virtualenv
> $ virtualenv --python=python3.5 myvenv
> ```
>
> **NOTE:** If you get an error like
>
> ```
> E: Unable to locate package python3-venv
> ```
>
> then instead run:
>
> ```
> sudo apt install python3.5-venv
> ```

Working with virtualenv

The command above will create a directory called `myvenv` (or whatever name you chose) that contains our virtual environment (basically a bunch of directory and files).

Windows

Start your virtual environment by running:

```
C:\Users\Name\bottleboys> myvenv\Scripts\activate
```

> **NOTE:** on Windows 10 you might get an error in the Windows PowerShell that says `execution of scripts is disabled on this system`. In this case, open another Windows PowerShell with the "Run as Administrator" option. Then try typing the following command before starting your virtual environment:
>
> ```
> C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
>     Execution Policy Change
>     The execution policy helps protect you from scripts that you do not trust. Changing t
> he execution policy might expose you to the security risks described in the about_Executi
> on_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to c
> hange the execution policy? [Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend
> [?] Help (default is "N"): A
> ```

Linux and OS X

Start your virtual environment by running:

```
$ source myvenv/bin/activate
```

Remember to replace `myvenv` with your chosen `virtualenv` name!

> **NOTE:** sometimes `source` might not be available. In those cases try doing this instead:
>
> ```
> $ . myvenv/bin/activate
> ```

You will know that you have `virtualenv` started when you see that the prompt in your console is prefixed with `(myvenv)`.

When working within a virtual environment, `python` will automatically refer to the correct version so you can use `python` instead of `python3`.

OK, we have all important dependencies in place. We can finally install Django!

Installing Bottle

Now that you have your `virtualenv` started, you can install Bottle.

Before we do that, we should make sure we have the latest version of `pip`, the software that we use to install Bottle:

```
(myvenv) ~$ pip install --upgrade pip
```

Then run `pip install bottle` to install Bottle.

That's it! You're now (finally) ready to create a Bottle application!

Install a code editor

There are a lot of different editors and it largely boils down to personal preference. Most Python programmers use complex but extremely powerful IDEs (Integrated Development Environments), such as PyCharm. As a beginner, however, that's probably less suitable; our recommendations are equally powerful, but a lot simpler.

Our suggestions are below, but feel free to ask your coach what their preferences are – it'll be easier to get help from them.

Gedit

Gedit is an open-source, free editor, available for all operating systems.

Download it here

Sublime Text 3

Sublime Text is a very popular editor with a free evaluation period. It's easy to install and use, and it's available for all operating systems.

Download it here

Atom

Atom is an extremely new code editor created by GitHub. It's free, open-source, easy to install and easy to use. It's available for Windows, OS X and Linux.

Download it here

Why are we installing a code editor?

You might be wondering why we are installing this special code editor software, rather than using something like Word or Notepad.

The first reason is that code needs to be **plain text**, and the problem with programs like Word and Textedit is that they don't actually produce plain text, they produce rich text (with fonts and formatting), using custom formats like RTF (Rich Text Format).

The second reason is that code editors are specialized for editing code, so they can provide helpful features like highlighting code with color according to its meaning, or automatically closing quotes for you.

We'll see all this in action later. Soon, you'll come to think of your trusty old code editor as one of your favorite tools. :)

Install Git

Git is a "version control system" used by a lot of programmers. This software can track changes to files over time so that you can recall specific versions later. A bit like the "track changes" feature in Microsoft Word, but much more powerful.

Installing Git

Windows

You can download Git from git-scm.com. You can hit "next" on all steps except for one; in the fifth step entitled "Adjusting your PATH environment", choose "Use Git and optional Unix tools from the Windows Command Prompt" (the bottom option). Other than that, the defaults are fine. Checkout Windows-style, commit Unix-style line endings is good.

OS X

Download Git from git-scm.com and just follow the instructions.

> **Note** If you are running OS X 10.6, 10.7, or 10.8, you will need to install the version of git from here: Git installer for OS X Snow Leopard

Create a GitHub account

Go to GitHub.com and sign up for a new, free user account.

Create a Heroku account

Next it's time to sign up for a free "Beginner" account on Heroku.

- www.heroku.com

After you sign up for a free account, you'll need to install the Heroku Toolbelt locally. This will allow you to provision your code to Heroku from the command line. Here are the steps if for installing it on the CodeAnywhere platform. You can find separate instructions for Mac, Windows and Linux platforms.

```
# Run this from your terminal.
# The following will add our apt repository and install the CLI:
wget -O- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

Start reading

Congratulations, you are all set up and ready to go! If you still have some time before the workshop, it would be useful to start reading a few of the beginning chapters:

- How the internet works
- Introduction to the command line
- Introduction to Python

Enjoy the workshop!

When you begin the workshop, you'll be able to go straight to Your first Bottle project! because you already covered the material in the earlier chapters.

How the Internet works

> For readers at home: this chapter is covered in the How the Internet Works video.
>
> This chapter is inspired by the talk "How the Internet works" by Jessica McKellar (http://web.mit.edu/jesstess/www/).

We bet you use the Internet every day. But do you actually know what happens when you type an address like https://wesbasinger.github.io into your browser and press `enter` ?

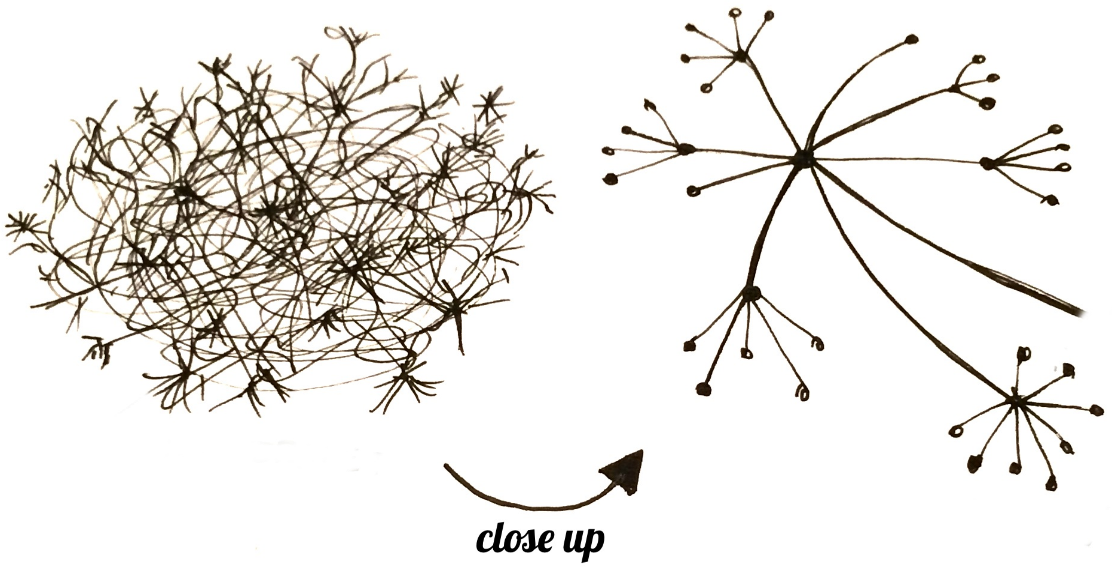
The first thing you need to understand is that a website is just a bunch of files saved on a hard disk. Just like your movies, music, or pictures. However, there is one part that is unique for websites: they include computer code called HTML.

If you're not familiar with programming it can be hard to grasp HTML at first, but your web browsers (like Chrome, Safari, Firefox, etc.) love it. Web browsers are designed to understand this code, follow its instructions, and present these files that your website is made of, exactly the way you want.

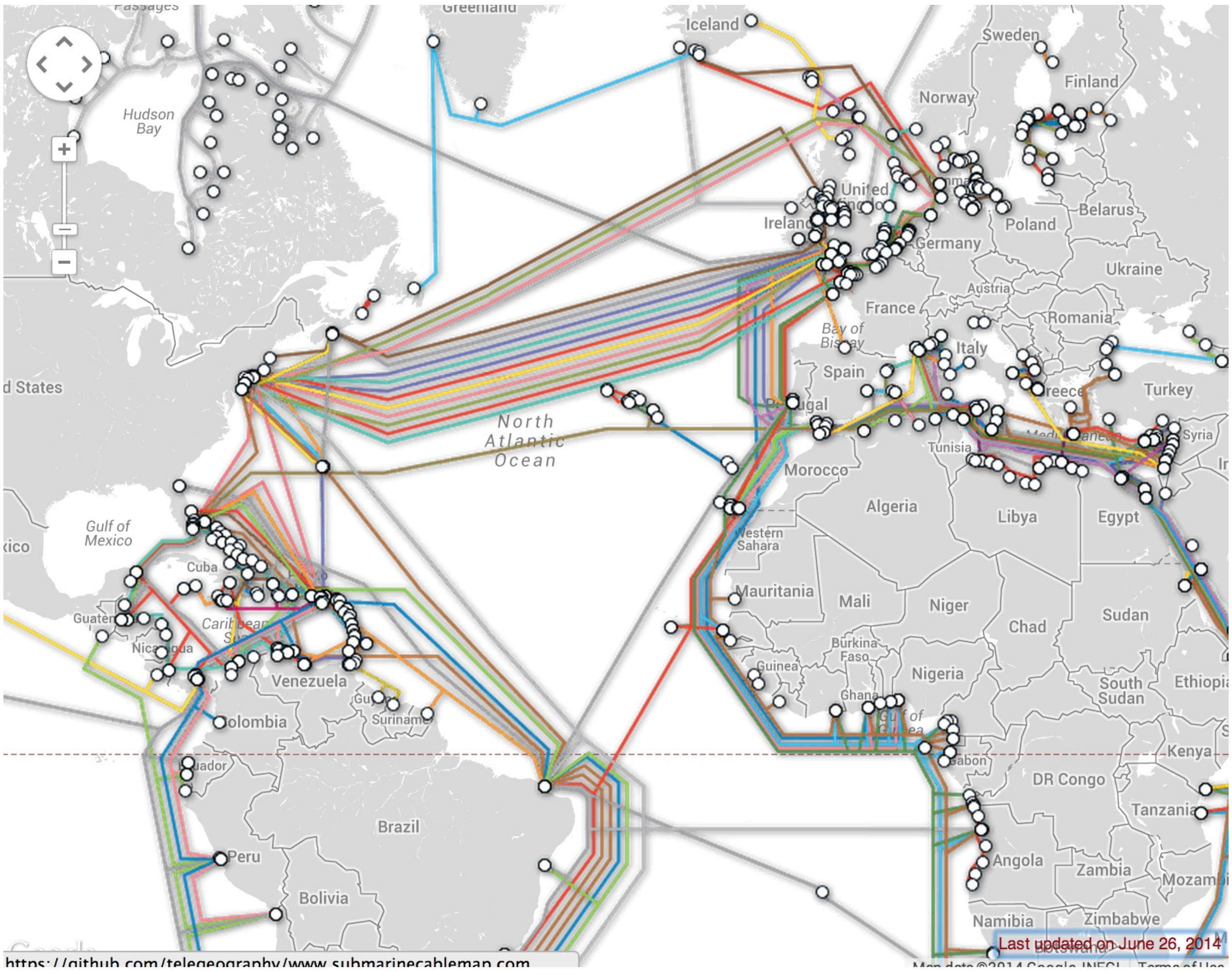
As with every file, we need to store HTML files somewhere on a hard disk. For the Internet, we use special, powerful computers called *servers*. They don't have a screen, mouse or a keyboard, because their main purpose is to store data and serve it. That's why they're called *servers* – because they *serve* you data.

OK, but you want to know how the Internet looks, right?

We drew you a picture! It looks like this:

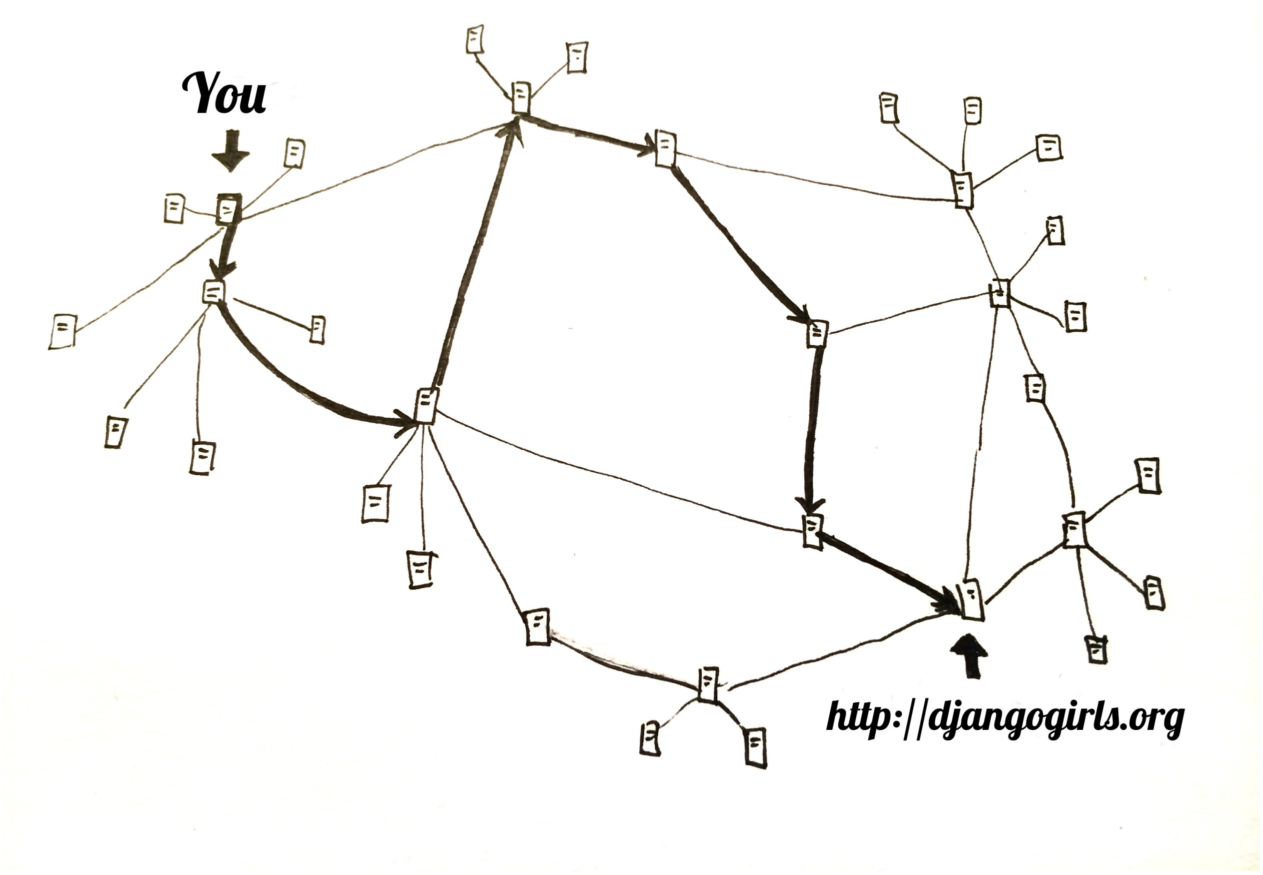


Looks like a mess, right? In fact it is a network of connected machines (the above-mentioned *servers*). Hundreds of thousands of machines! Many, many kilometers of cables around the world! You can visit a Submarine Cable Map website (http://submarinecablemap.com) to see how complicated the net is. Here is a screenshot from the website:



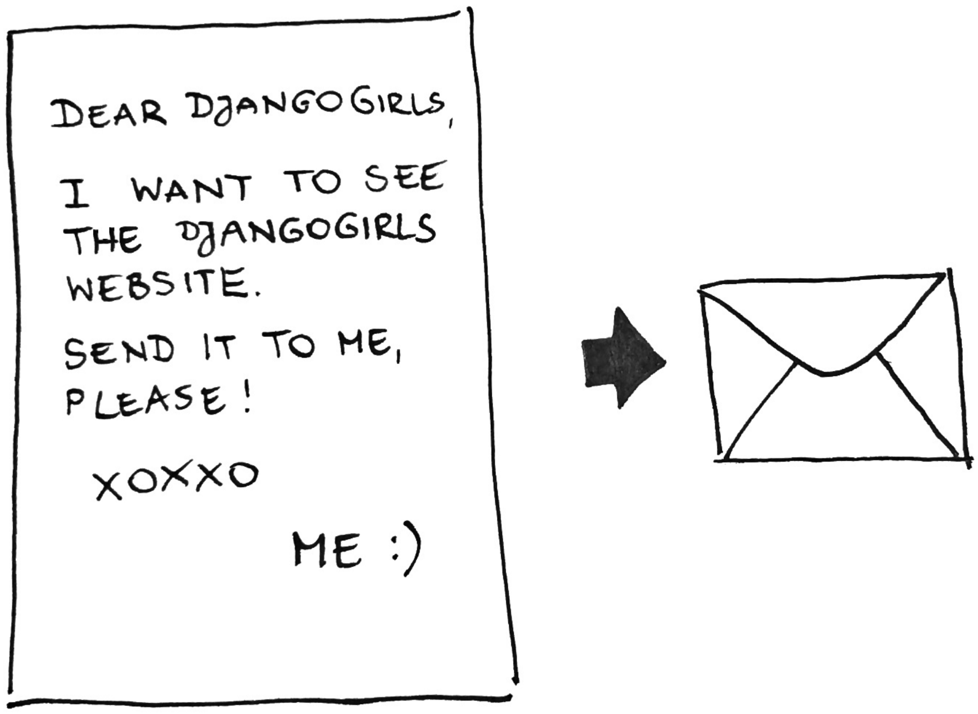
It is fascinating, isn't it? But obviously, it is not possible to have a wire between every machine connected to the Internet. So, to reach a machine (for example, the one where https://djangogirls.org is saved) we need to pass a request through many, many different machines.

It looks like this:



Imagine that when you type https://wesbasinger.github.io, you send a letter that says: "Dear Wes Basinger, I want to see the wesbasinger.github.io website. Send it to me, please!"

Your letter goes to the post office closest to you. Then it goes to another that is a bit nearer to your addressee, then to another, and another until it is delivered at its destination. The only unique thing is that if you send many letters (*data packets*) to the same place, they could go through totally different post offices (*routers*). This depends on how they are distributed at each office.



Yes, it is as simple as that. You send messages and you expect some response. Of course, instead of paper and pen you use bytes of data, but the idea is the same!

Instead of addresses with a street name, city, zip code and country name, we use IP addresses. Your computer first asks the DNS (Domain Name System) to translate wesbasinger.github.io into an IP address. It works a little bit like old-fashioned phonebooks where you can look up the name of the person you want to contact and find their phone number and address.

When you send a letter, it needs to have certain features to be delivered correctly: an address, a stamp, etc. You also use a language that the receiver understands, right? The same applies to the *data packets* you send to see a website. We use a protocol called HTTP (Hypertext Transfer Protocol).

So, basically, when you have a website, you need to have a *server* (machine) where it lives. When the *server* receives an incoming *request* (in a letter), it sends back your website (in another letter).

Since this is a Bottle tutorial, you might ask what Bottle does. When you send a response, you don't always want to send the same thing to everybody. It is so much better if your letters are personalized, especially for the person that has just written to you, right? Bottle helps you with creating these personalized, interesting letters. :)

Enough talk – time to create!

Introduction to the command-line interface

> For readers at home: this chapter is covered in the Your new friend: Command Line video.
>
> Please note, if you are using CodeAnywhere as your platform, then the SSH terminal is where you will run your commands. You should follow the Mac Linux directions.

It's exciting, right?! You'll write your first line of code in just a few minutes! :)

Let us introduce you to your first new friend: the command line!

The following steps will show you how to use the black window all hackers use. It might look a bit scary at first but really it's just a prompt waiting for commands from you.

> **Note** Please note that throughout this book we use the terms 'directory' and 'folder' interchangeably but they are one and the same thing.

What is the command line?

The window, which is usually called the **command line** or **command-line interface**, is a text-based application for viewing, handling, and manipulating files on your computer. It's much like Windows Explorer or Finder on the Mac, but without the graphical interface. Other names for the command line are: *cmd*, *CLI*, *prompt*, *console* or *terminal*.

Open the command-line interface

To start some experiments we need to open our command-line interface first.

Windows

Go to Start menu → All Programs → Accessories → Command Prompt.

OS X

Go to Applications → Utilities → Terminal.

Prompt

You now should see a white or black window that is waiting for your commands.

OS X and Linux

If you're on Mac or Linux, you probably see `$`, just like this:

command-line

```
$
```

Windows

On Windows, it's a `>` sign, like this:

command-line

```
>
```

Each command will be prepended by this sign and one space, but you don't have to type it. Your computer will do it for you. :)

> Just a small note: in your case there may be something like `C:\Users\ola>` or `Olas-MacBook-Air:~ ola$` before the prompt sign, and this is 100% OK.

The part up to and including the `$` or the `>` is called the *command line prompt*, or *prompt* for short. It prompts you to input something there.

In the tutorial, when we want you to type in a command, we will include the `$` or `>`, and occasionally more to the left. You can ignore the left part and just type in the command which starts after the prompt.

Your first command (YAY!)

Let's start with something simple. Type this command:

OS X and Linux

command-line

```
$ whoami
```

Windows

command-line

```
> whoami
```

And then hit `enter`. This is our result:

command-line

```
$ whoami
olasitarska
```

As you can see, the computer has just printed your username. Neat, huh? :)

> Try to type each command; do not copy-paste. You'll remember more this way!

Basics

Each operating system has a slightly different set of commands for the command line, so make sure to follow instructions for your operating system. Let's try this, shall we?

Current directory

It'd be nice to know where are we now, right? Let's see. Type this command and hit `enter` :

OS X and Linux

command-line

```
$ pwd
/Users/olasitarska
```

Windows

command-line

```
> cd
C:\Users\olasitarska
```

You'll probably see something similar on your machine. Once you open the command line you usually start at your user's home directory.

> Note: 'pwd' stands for 'print working directory'.

List files and directories

So what's in it? It'd be cool to find out. Let's see:

OS X and Linux

command-line

```
$ ls
Applications
Desktop
Downloads
Music
...
```

Windows

command-line

```
> dir
 Directory of C:\Users\olasitarska
05/08/2014 07:28 PM <DIR>      Applications
05/08/2014 07:28 PM <DIR>      Desktop
05/08/2014 07:28 PM <DIR>      Downloads
05/08/2014 07:28 PM <DIR>      Music
...
```

Change current directory

Now, let's go to our Desktop directory:

OS X and Linux

command-line

```
$ cd Desktop
```

Windows

command-line

```
> cd Desktop
```

Check if it's really changed:

OS X and Linux

command-line

```
$ pwd
/Users/olasitarska/Desktop
```

Windows

command-line

```
> cd
C:\Users\olasitarska\Desktop
```

Here it is!

> PRO tip: if you type `cd D` and then hit `tab` on your keyboard, the command line will automatically fill in the rest of the name so you can navigate faster. If there is more than one folder starting with "D", hit the `tab` button twice to get a list of options.

Create directory

How about creating a practice directory on your desktop? You can do it this way:

OS X and Linux

command-line

```
$ mkdir practice
```

Windows

command-line

```
> mkdir practice
```

This little command will create a folder with the name `practice` on your desktop. You can check if it's there just by looking on your Desktop or by running a `ls` or `dir` command! Try it. :)

> PRO tip: If you don't want to type the same commands over and over, try pressing the `up arrow` and `down arrow` on your keyboard to cycle through recently used commands.

Exercise!

A small challenge for you: in your newly created `practice` directory, create a directory called `test`. (Use the `cd` and `mkdir` commands.)

Solution:

OS X and Linux

command-line

```
$ cd practice
$ mkdir test
$ ls
test
```

Windows

command-line

```
> cd practice
> mkdir test
> dir
05/08/2014 07:28 PM <DIR>      test
```

Congrats! :)

Clean up

We don't want to leave a mess, so let's remove everything we did until that point.

First, we need to get back to Desktop:

OS X and Linux

command-line

```
$ cd ..
```

Windows

command-line

```
> cd ..
```

Using `..` with the `cd` command will change your current directory to the parent directory (that is, the directory that contains your current directory).

Check where you are:

OS X and Linux

command-line

```
$ pwd
/Users/olasitarska/Desktop
```

Windows

command-line

```
> cd
C:\Users\olasitarska\Desktop
```

Now time to delete the `practice` directory:

> **Attention**: Deleting files using `del`, `rmdir` or `rm` is irrecoverable, meaning *the deleted files will be gone forever*! So be very careful with this command.

OS X and Linux

command-line

```
$ rm -r practice
```

Windows

command-line

```
> rmdir /S practice
practice, Are you sure <Y/N>? Y
```

Done! To be sure it's actually deleted, let's check it:

OS X and Linux

command-line

```
$ ls
```

Windows

command-line

```
> dir
```

Exit

That's it for now! You can safely close the command line now. Let's do it the hacker way, alright? :)

OS X and Linux

command-line

```
$ exit
```

Windows

command-line

```
> exit
```

Cool, huh? :)

Summary

Here is a summary of some useful commands:

Command (Windows)	Command (Mac OS / Linux)	Description	Example
exit	exit	close the window	**exit**
cd	cd	change directory	**cd test**
cd	pwd	show the current directory	**cd** (Windows) or **pwd** (Mac OS / Linux)
dir	ls	list directories/files	**dir**
copy	cp	copy file	**copy c:\test\test.txt c:\windows\test.txt**
move	mv	move file	**move c:\test\test.txt c:\windows\test.txt**
mkdir	mkdir	create a new directory	**mkdir testdirectory**
rmdir (or del)	rm	delete a file	**del c:\test\test.txt**
rmdir /S	rm -r	delete a directory	**rm -r testdirectory**

These are just a very few of the commands you can run in your command line, but you're not going to use anything more than that today.

If you're curious, ss64.com contains a complete reference of commands for all operating systems.

Ready?

Let's dive into Python!

Let’s start with Python

We're finally here!

But first, let us tell you what Python is. Python is a very popular programming language that can be used for creating websites, games, scientific software, graphics, and much, much more.

Python originated in the late 1980s and its main goal is to be readable by human beings (not only machines!). This is why it looks much simpler than other programming languages. This makes it easy to learn, but don't worry – Python is also really powerful!

Python installation

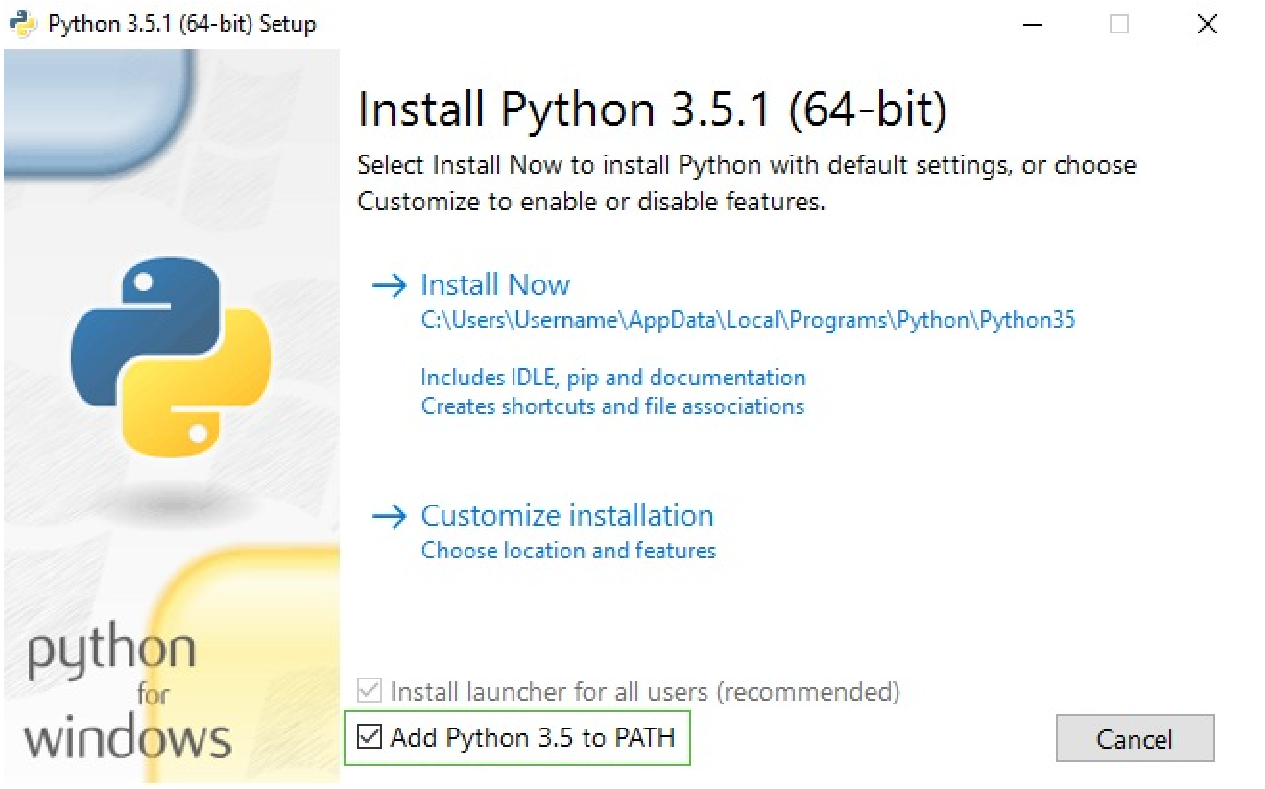
> **Note** If you're using a Chromebook, skip this chapter and make sure you follow the Chromebook Setup instructions.
>
> **Note** If you already worked through the Installation steps, there's no need to do this again – you can skip straight ahead to the next chapter!
>
> For readers at home: this chapter is covered in the Installing Python & Code Editor video.
>
> This section is based on a tutorial by Geek Girls Carrots (https://github.com/ggcarrots/django-carrots)

Bottle is written in Python. We need Python to do anything with Bottle. Let's start by installing it! We want you to install Python 3.5, so if you have any earlier version, you will need to upgrade it.

Windows

First check whether your computer is running a 32-bit version or a 64-bit version of Windows at https://support.microsoft.com/en-au/kb/827218. You can download Python for Windows from the website https://www.python.org/downloads/windows/. Click on the "Latest Python 3 Release - Python x.x.x" link. If your computer is running a **64-bit** version of Windows, download the **Windows x86-64 executable installer**. Otherwise, download the **Windows x86 executable installer**. After downloading the installer, you should run it (double-click on it) and follow the instructions there.

One thing to watch out for: During the installation you will notice a window marked "Setup". Make sure you tick the "Add Python 3.5 to PATH" checkbox and click on "Install Now", as shown here:


Python 3.5.1 (64-bit) Setup

Install Python 3.5.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

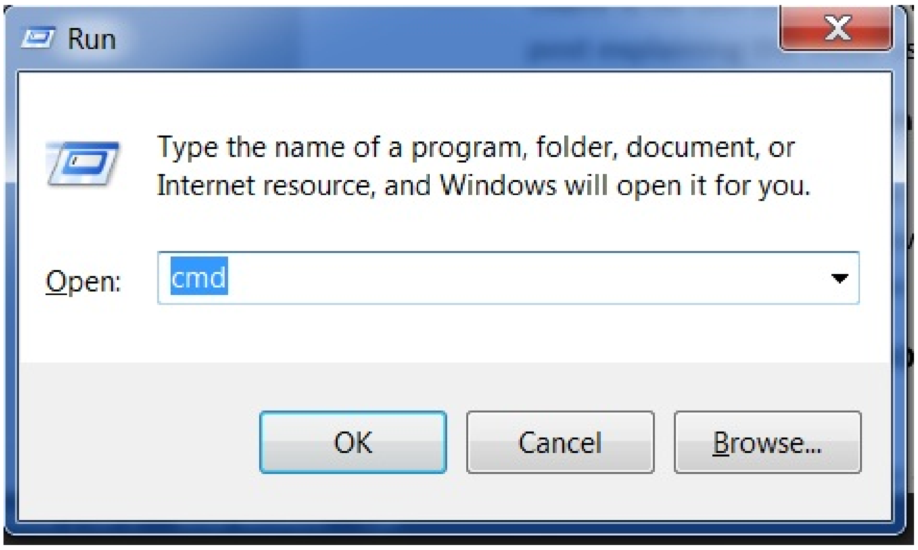
→ Install Now
C:\Users\Username\AppData\Local\Programs\Python\Python35

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.5 to PATH

In upcoming steps, you'll be using the Windows Command Line (which we'll also tell you about). For now, if you need to type in some commands, go to Start menu → All Programs → Accessories → Command Prompt. You can also hold in the Windows key and press the "R"-key until the "Run" window pops up. To open the Command Line, type "cmd" and press enter in the "Run" window. (On newer versions of Windows, you might have to search for "Command Prompt" since it's sometimes hidden.)


Run

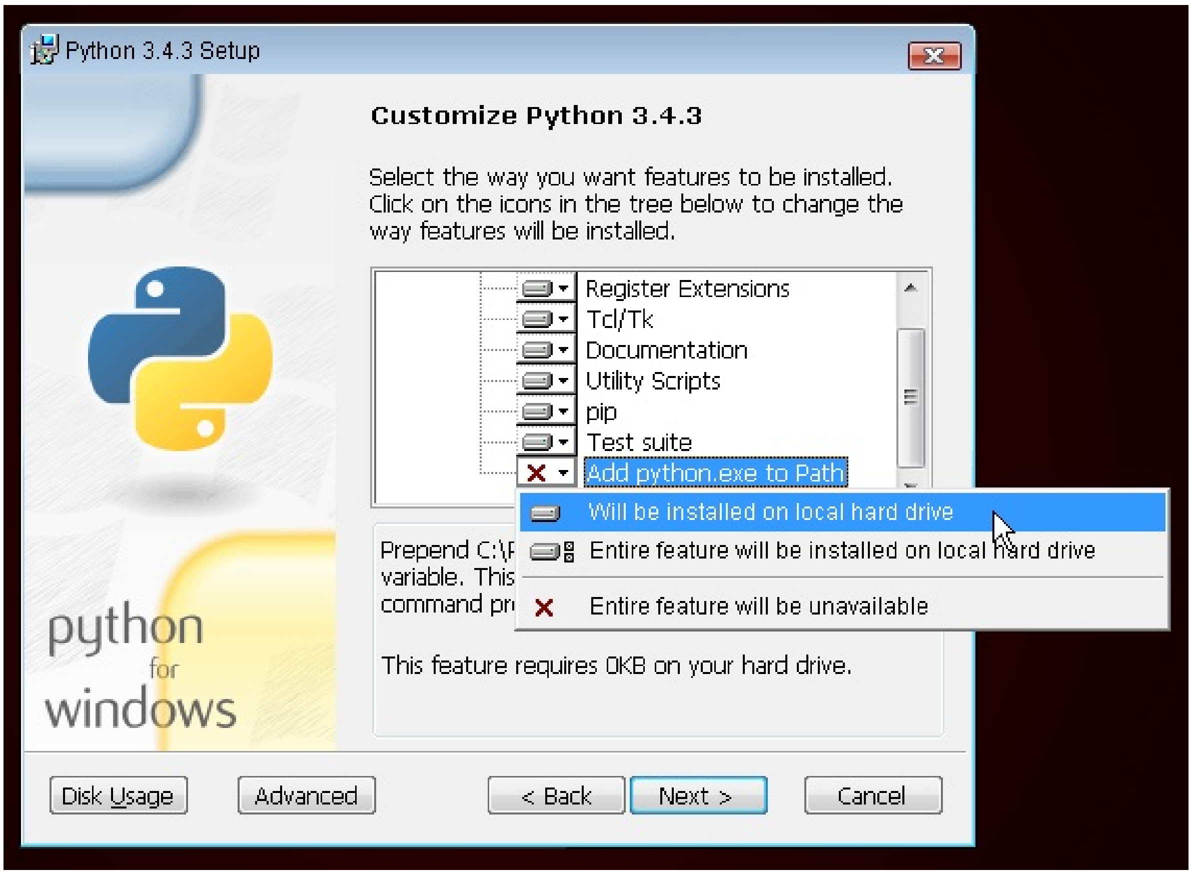
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: cmd

Note: if you are using an older version of Windows (7, Vista, or any older version) and the Python 3.5.x installer fails with an error, you can try either:

1. install all Windows Updates and try to install Python 3.5 again; or
2. install an older version of Python, e.g., 3.4.4.

If you install an older version of Python, the installation screen may look a bit different than shown above. Make sure you scroll down to see "Add python.exe to Path", then click the button on the left and pick "Will be installed on local hard drive":


Python 3.4.3 Setup
Customize Python 3.4.3
Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.
Register Extensions
Tcl/Tk
Documentation
Utility Scripts
pip
Test suite
Add python.exe to Path
Will be installed on local hard drive
Entire feature will be installed on local hard drive
Entire feature will be unavailable
Prepend C:\P
variable. This
command pr
This feature requires 0KB on your hard drive.
python for windows
Disk Usage
Advanced
< Back
Next >
Cancel

OS X

> **Note** Before you install Python on OS X, you should ensure your Mac settings allow installing packages that aren't from the App Store. Go to System Preferences (it's in the Applications folder), click "Security & Privacy," and then the "General" tab. If your "Allow apps downloaded from:" is set to "Mac App Store," change it to "Mac App Store and identified developers."

You need to go to the website https://www.python.org/downloads/release/python-351/ and download the Python installer:

- Download the *Mac OS X 64-bit/32-bit installer* file,
- Double click *python-3.5.1-macosx10.6.pkg* to run the installer.

Verify the installation was successful by opening the *Terminal* application and running the `python3` command:

```
$ python3 --version
Python 3.5.1
```

NOTE: If you're on Windows and you get an error message that `python3` wasn't found, try using `python` (without the `3`) and check if it still might be a version of Python 3.5.

If you have any doubts, or if something went wrong and you have no idea what to do next, please ask your coach! Sometimes things don't go smoothly and it's better to ask for help from someone with more experience.

Code editor

> For readers at home: this chapter is covered in the Installing Python & Code Editor video.

You're about to write your first line of code, so it's time to download a code editor!

> If you're using a Chromebook, skip this chapter and make sure you follow the Chromebook Setup instructions.
>
> **Note** You might have done this earlier in the Installation chapter – if so, you can skip right ahead to the next chapter!

There are a lot of different editors and it largely boils down to personal preference. Most Python programmers use complex but extremely powerful IDEs (Integrated Development Environments), such as PyCharm. As a beginner, however, that's probably less suitable; our recommendations are equally powerful, but a lot simpler.

Our suggestions are below, but feel free to ask your coach what their preferences are – it'll be easier to get help from them.

Gedit

Gedit is an open-source, free editor, available for all operating systems.

Download it here

Sublime Text 3

Sublime Text is a very popular editor with a free evaluation period. It's easy to install and use, and it's available for all operating systems.

Download it here

Atom

Atom is an extremely new code editor created by GitHub. It's free, open-source, easy to install and easy to use. It's available for Windows, OS X and Linux.

Download it here

Why are we installing a code editor?

You might be wondering why we are installing this special code editor software, rather than using something like Word or Notepad.

The first reason is that code needs to be **plain text**, and the problem with programs like Word and Textedit is that they don't actually produce plain text, they produce rich text (with fonts and formatting), using custom formats like RTF (Rich Text Format).

The second reason is that code editors are specialized for editing code, so they can provide helpful features like highlighting code with color according to its meaning, or automatically closing quotes for you.

We'll see all this in action later. Soon, you'll come to think of your trusty old code editor as one of your favorite tools. :)

Introduction to Python

> Part of this chapter is based on tutorials by Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).

Let's write some code!

Python prompt

> For readers at home: this part is covered in the Python Basics: Integers, Strings, Lists, Variables and Errors video.

To start playing with Python, we need to open up a *command line* on your computer. You should already know how to do that – you learned it in the Intro to Command Line chapter.

Once you're ready, follow the instructions below.

We want to open up a Python console, so type in `python` on Windows or `python3` on Mac OS/Linux and hit `enter`.

command-line

```
$ python
Python 3.5.1 (...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Your first Python command!

After running the Python command, the prompt changed to `>>>`. For us this means that for now we may only use commands in the Python language. You don't have to type in `>>>` – Python will do that for you.

If you want to exit the Python console at any point, just type `exit()` or use the shortcut `Ctrl + Z` for Windows and `Ctrl + D` for Mac/Linux. Then you won't see `>>>` any longer.

For now, we don't want to exit the Python console. We want to learn more about it. Let's start with something really simple. For example, try typing some math, like `2 + 3` and hit `enter`.

command-line

```
>>> 2 + 3
5
```

Nice! See how the answer popped out? Python knows math! You could try other commands like:

- `4 * 5`
- `5 - 1`
- `40 / 2`

To perform exponential calculation, say 2 to the power 3, we type:

```
>>> 2 ** 3
8
```

Have fun with this for a little while and then get back here. :)

As you can see, Python is a great calculator. If you're wondering what else you can do…

Strings

How about your name? Type your first name in quotes like this:

command-line

```
>>> "Wes"
'Wes'
```

You've now created your first string! It's a sequence of characters that can be processed by a computer. The string must always begin and end with the same character. This may be single (`'`) or double (`"`) quotes (there is no difference!) The quotes tell Python that what's inside of them is a string.

Strings can be strung together. Try this:

command-line

```
>>> "Hi there " + "Wes"
'Hi there Wes'
```

You can also multiply strings with a number:

command-line

```
>>> "Wes" * 3
'WesWesWes'
```

If you need to put an apostrophe inside your string, you have two ways to do it.

Using double quotes:

command-line

```
>>> "Runnin' down the hill"
"Runnin' down the hill"
```

or escaping the apostrophe with a backslash (`\`):

command-line

```
>>> 'Runnin\' down the hill'
"Runnin' down the hill"
```

Nice, huh? To see your name in uppercase letters, simply type:

command-line

```
>>> "Wes".upper()
'WES'
```

You just used the `upper` **method** on your string! A method (like `upper()`) is a sequence of instructions that Python has to perform on a given object (`"Wes"`) once you call it.

If you want to know the number of letters contained in your name, there is a **function** for that too!

command-line

```
>>> len("Wes")
3
```

Wonder why sometimes you call functions with a `.` at the end of a string (like `"Wes".upper()`) and sometimes you first call a function and place the string in parentheses? Well, in some cases, functions belong to objects, like `upper()` , which can only be performed on strings. In this case, we call the function a **method**. Other times, functions don't belong to anything specific and can be used on different types of objects, just like `len()` . That's why we're giving `"Wes"` as a parameter to the `len` function.

Summary

OK, enough of strings. So far you've learned about:

- **the prompt** – typing commands (code) into the Python prompt results in answers in Python
- **numbers and strings** – in Python numbers are used for math and strings for text objects
- **operators** – like `+` and `*` , combine values to produce a new one
- **functions** – like `upper()` and `len()` , perform actions on objects.

These are the basics of every programming language you learn. Ready for something harder? We bet you are!

Errors

Let's try something new. Can we get the length of a number the same way we could find out the length of our name? Type in `len(304023)` and hit `enter` :

command-line

```
>>> len(304023)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

We got our first error! It says that objects of type "int" (integers, whole numbers) have no length. So what can we do now? Maybe we can write our number as a string? Strings have a length, right?

command-line

```
>>> len(str(304023))
6
```

It worked! We used the `str` function inside of the `len` function. `str()` converts everything to strings.

- The `str` function converts things into **strings**
- The `int` function converts things into **integers**

> Important: we can convert numbers into text, but we can't necessarily convert text into numbers – what would `int('hello')` be anyway?

Variables

An important concept in programming is variables. A variable is nothing more than a name for something so you can use it later. Programmers use these variables to store data, make their code more readable and so they don't have to keep remembering what things are.

Let's say we want to create a new variable called `name` :

command-line

```
>>> name = "Wes"
```

You see? It's easy! It's simply: name equals Wes.

As you've noticed, your program didn't return anything like it did before. So how do we know that the variable actually exists? Simply enter `name` and hit `enter` :

command-line

```
>>> name
'Wes'
```

Yippee! Your first variable! :) You can always change what it refers to:

command-line

```
>>> name = "Willy"
>>> name
'Willy'
```

You can use it in functions too:

command-line

```
>>> len(name)
5
```

Awesome, right? Of course, variables can be anything – numbers too! Try this:

command-line

```
>>> a = 4
>>> b = 6
>>> a * b
24
```

But what if we used the wrong name? Can you guess what would happen? Let's try!

command-line

```
>>> city = "Tokyo"
>>> ctiy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ctiy' is not defined
```

An error! As you can see, Python has different types of errors and this one is called a **NameError**. Python will give you this error if you try to use a variable that hasn't been defined yet. If you encounter this error later, check your code to see if you've mistyped any names.

Play with this for a while and see what you can do!

The print function

Try this:

command-line

```
>>> name = 'Mike'
>>> name
'Mike'
>>> print(name)
Mike
```

When you just type `name`, the Python interpreter responds with the string *representation* of the variable 'name', which is the letters M-a-r-i-a, surrounded by single quotes, ''. When you say `print(name)`, Python will "print" the contents of the variable to the screen, without the quotes, which is neater.

As we'll see later, `print()` is also useful when we want to print things from inside functions, or when we want to print things on multiple lines.

Lists

Beside strings and integers, Python has all sorts of different types of objects. Now we're going to introduce one called **list**. Lists are exactly what you think they are: objects which are lists of other objects. :)

Go ahead and create a list:

command-line

```
>>> []
[]
```

Yes, this list is empty. Not very useful, right? Let's create a list of lottery numbers. We don't want to repeat ourselves all the time, so we will put it in a variable, too:

command-line

```
>>> lottery = [3, 42, 12, 19, 30, 59]
```

All right, we have a list! What can we do with it? Let's see how many lottery numbers there are in a list. Do you have any idea which function you should use for that? You know this already!

command-line

```
>>> len(lottery)
6
```

Yes! `len()` can give you a number of objects in a list. Handy, right? Maybe we will sort it now:

command-line

```
>>> lottery.sort()
```

This doesn't return anything, it just changed the order in which the numbers appear in the list. Let's print it out again and see what happened:

command-line

```
>>> print(lottery)
[3, 12, 19, 30, 42, 59]
```

As you can see, the numbers in your list are now sorted from the lowest to highest value. Congrats!

Maybe we want to reverse that order? Let's do that!

command-line

```
>>> lottery.reverse()
>>> print(lottery)
[59, 42, 30, 19, 12, 3]
```

Easy, right? If you want to add something to your list, you can do this by typing this command:

command-line

```
>>> lottery.append(199)
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
```

If you want to show only the first number, you can do this by using **indexes**. An index is the number that says where in a list an item occurs. Programmers prefer to start counting at 0, so the first object in your list is at index 0, the next one is at 1, and so on. Try this:

command-line

```
>>> print(lottery[0])
59
>>> print(lottery[1])
42
```

As you can see, you can access different objects in your list by using the list's name and the object's index inside of square brackets.

To delete something from your list you will need to use **indexes** as we learned above and the `pop()` method. Let's try an example and reinforce what we learned previously; we will be deleting the first number of our list.

command-line

```
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
>>> print(lottery[0])
59
>>> lottery.pop(0)
>>> print(lottery)
[42, 30, 19, 12, 3, 199]
```

That worked like a charm!

For extra fun, try some other indexes: 6, 7, 1000, -1, -6 or -1000. See if you can predict the result before trying the command. Do the results make sense?

You can find a list of all available list methods in this chapter of the Python documentation: https://docs.python.org/3/tutorial/datastructures.html

Dictionaries

> For readers at home: this part is covered in the Python Basics: Dictionaries video.

A dictionary is similar to a list, but you access values by looking up a key instead of a numeric index. A key can be any string or number. The syntax to define an empty dictionary is:

command-line

```
>>> {}
{}
```

This shows that you just created an empty dictionary. Hurray!

Now, try writing the following command (try substituting your own information, too):

command-line

```
>>> participant = {'name': 'Wes', 'country': 'America', 'favorite_numbers': [7, 42, 92]}
```

With this command, you just created a variable named `participant` with three key–value pairs:

- The key `name` points to the value `'Wes'` (a `string` object),
- `country` points to `'America'` (another `string`),
- and `favorite_numbers` points to `[7, 42, 92]` (a `list` with three numbers in it).

You can check the content of individual keys with this syntax:

command-line

```
>>> print(participant['name'])
Wes
```

See, it's similar to a list. But you don't need to remember the index – just the name.

What happens if we ask Python the value of a key that doesn't exist? Can you guess? Let's try it and see!

command-line

```
>>> participant['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

Look, another error! This one is a **KeyError**. Python is helpful and tells you that the key `'age'` doesn't exist in this dictionary.

When should you use a dictionary or a list? Well, that's a good point to ponder. Just have a solution in mind before looking at the answer in the next line.

- Do you just need an ordered sequence of items? Go for a list.
- Do you need to associate values with keys, so you can look them up efficiently (by key) later on? Use a dictionary.

Dictionaries, like lists, are *mutable*, meaning that they can be changed after they are created. You can add new key–value pairs to a dictionary after it is created, like this:

command-line

```
>>> participant['favorite_language'] = 'Python'
```

Like lists, using the `len()` method on the dictionaries returns the number of key–value pairs in the dictionary. Go ahead and type in this command:

command-line

```
>>> len(participant)
4
```

I hope it makes sense up to now. :) Ready for some more fun with dictionaries? Read on for some amazing things.

You can use the `pop()` method to delete an item in the dictionary. Say, if you want to delete the entry corresponding to the key `'favorite_numbers'`, just type in the following command:

command-line

```
>>> participant.pop('favorite_numbers')
>>> participant
{'country': 'America', 'favorite_language': 'Python', 'name': 'Wes'}
```

As you can see from the output, the key–value pair corresponding to the 'favorite_numbers' key has been deleted.

As well as this, you can also change a value associated with an already-created key in the dictionary. Type this:

command-line

```
>>> participant['country'] = 'Germany'
>>> participant
{'country': 'Germany', 'favorite_language': 'Python', 'name': 'Wes'}
```

As you can see, the value of the key `'country'` has been altered from `'Poland'` to `'Germany'`. :) Exciting? Hurrah! You just learned another amazing thing.

Summary

Awesome! You know a lot about programming now. In this last part you learned about:

- **errors** – you now know how to read and understand errors that show up if Python doesn't understand a command you've given it
- **variables** – names for objects that allow you to code more easily and to make your code more readable
- **lists** – lists of objects stored in a particular order
- **dictionaries** – objects stored as key–value pairs

Excited for the next part? :)

Compare things

> For readers at home: this part is covered in the Python Basics: Comparisons video.

A big part of programming involves comparing things. What's the easiest thing to compare? Numbers, of course. Let's see how that works:

command-line

```
>>> 5 > 2
True
>>> 3 < 1
False
>>> 5 > 2 * 2
True
>>> 1 == 1
True
>>> 5 != 2
True
```

We gave Python some numbers to compare. As you can see, not only can Python compare numbers, but it can also compare method results. Nice, huh?

Do you wonder why we put two equal signs `==` next to each other to compare if numbers are equal? We use a single `=` for assigning values to variables. You always, **always** need to put two of them – `==` – if you want to check if things are equal to each other. We can also state that things are unequal to each other. For that, we use the symbol `!=`, as shown in the example above.

Give Python two more tasks:

command-line

```
>>> 6 >= 12 / 2
True
>>> 3 <= 2
False
```

`>` and `<` are easy, but what do `>=` and `<=` mean? Read them like this:

- x `>` y means: x is greater than y
- x `<` y means: x is less than y
- x `<=` y means: x is less than or equal to y
- x `>=` y means: x is greater than or equal to y

Awesome! Wanna do one more? Try this:

command-line

```
>>> 6 > 2 and 2 < 3
True
>>> 3 > 2 and 2 < 1
False
>>> 3 > 2 or 2 < 1
True
```

You can give Python as many numbers to compare as you want, and it will give you an answer! Pretty smart, right?

- **and** – if you use the `and` operator, both comparisons have to be True in order for the whole command to be True
- **or** – if you use the `or` operator, only one of the comparisons has to be True in order for the whole command

to be True

Have you heard of the expression "comparing apples to oranges"? Let's try the Python equivalent:

command-line

```
>>> 1 > 'bottle'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

Here you see that just like in the expression, Python is not able to compare a number (`int`) and a string (`str`). Instead, it shows a **TypeError** and tells us the two types can't be compared together.

Boolean

Incidentally, you just learned about a new type of object in Python. It's called **Boolean**, and it is probably the easiest type there is.

There are only two Boolean objects:

- True
- False

But for Python to understand this, you need to always write it as 'True' (first letter uppercase, with the rest of the letters lowercased). **true, TRUE, and tRUE won't work – only True is correct.** (The same applies to 'False' as well, of course.)

Booleans can be variables, too! See here:

command-line

```
>>> a = True
>>> a
True
```

You can also do it this way:

command-line

```
>>> a = 2 > 5
>>> a
False
```

Practice and have fun with Booleans by trying to run the following commands:

- `True and True`
- `False and True`
- `True or 1 == 1`
- `1 != 2`

Congrats! Booleans are one of the coolest features in programming, and you just learned how to use them!

Save it!

> For readers at home: this part is covered in the Python Basics: Saving files and "If" statement video.

So far we've been writing all our python code in the interpreter, which limits us to entering one line of code at a time. Normal programs are saved in files and executed by our programming language **interpreter** or **compiler**. So far we've been running our programs one line at a time in the Python **interpreter**. We're going to need more than one line of code for the next few tasks, so we'll quickly need to:

- Exit the Python interpreter
- Open up our code editor of choice
- Save some code into a new python file
- Run it!

To exit from the Python interpreter that we've been using, simply type the `exit()` function

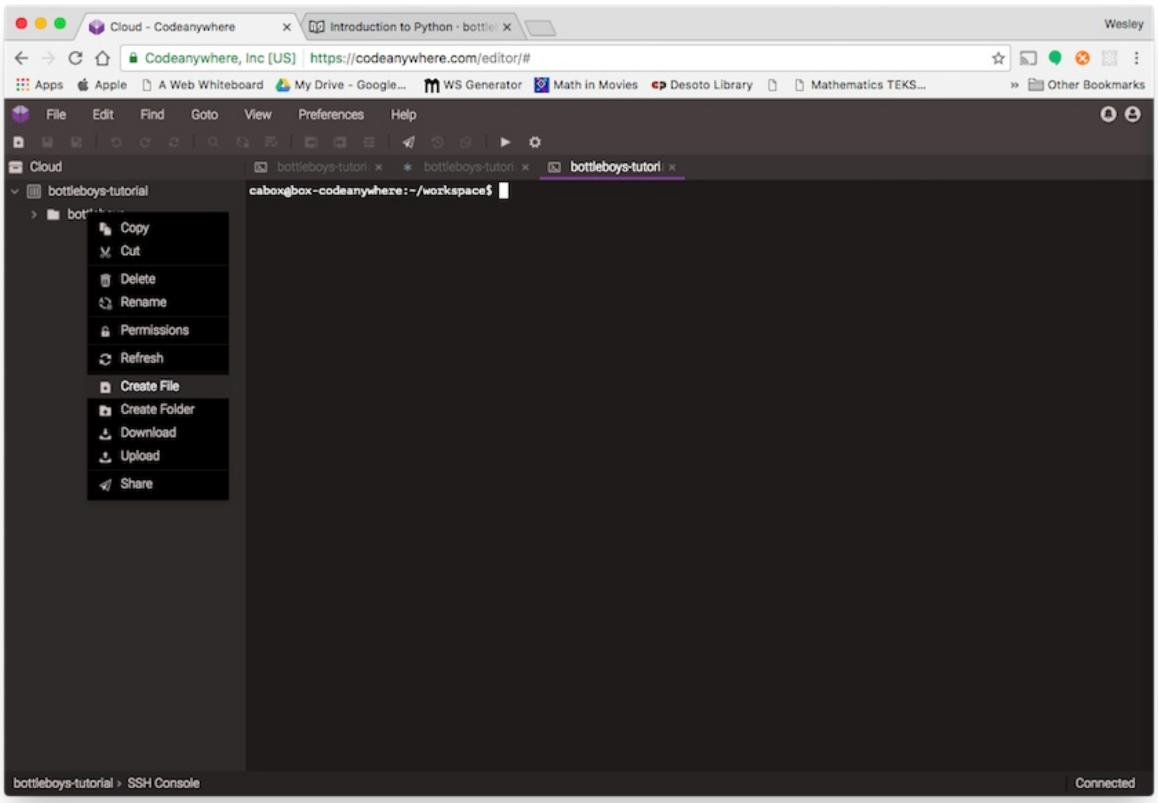
command-line

```
>>> exit()
$
```

This will put you back into the command prompt.

Earlier, we picked out a code editor from the code editor section. We'll need to open the editor now and write some code into a new file:

If you are working on the CodeAnywhere platform, you can right click on one of the folders in your container and create a new file with the dialog box. I've included a picture below.



```
cabox@box-codeanywhere:~/workspace$
```

editor

```
print('Hello, Bottle boys!')
```

Obviously, you're a pretty seasoned Python developer now, so feel free to write some code that you've learned today.

Now we need to save the file and give it a descriptive name. Let's call the file **python_intro.py** and save it to a folder in your workspace. We can name the file anything we want, but the important part here is to make sure the file ends in **.py**. The **.py** extension tells our operating system that this is a **Python executable file** and Python can run it.

> You should notice one of the coolest thing about code editors: colors! In the Python console, everything was the same color; now you should see that the `print` function is a different color from the string. This is called "syntax highlighting", and it's a really useful feature when coding. The color of things will give you hints, such as unclosed strings or a typo in a keyword name (like the `def` in a function, which we'll see below). This is one of the reasons we use a code editor. :)

With the file saved, it's time to run it! Using the skills you've learned in the command line section, use the terminal to **change directories** to whichever directory you saved the file. You can use the `cd` command. If you get stuck, just ask for help.

Now use Python to execute the code in the file like this:

command-line

```
$ python python_intro.py
Hello, Bottle boys!
```

Alright! You just ran your first Python program that was saved to a file. Feel awesome?

You can now move on to an essential tool in programming:

If … elif … else

Lots of things in code should be executed only when given conditions are met. That's why Python has something called **if statements**.

Replace the code in your **python_intro.py** file with this:

python_intro.py

```
if 3 > 2:
```

If we were to save and run this, we'd see an error like this:

command-line

```
$ python python_intro.py
File "python_intro.py", line 2
         ^
SyntaxError: unexpected EOF while parsing
```

Python expects us to give further instructions to it which are executed if the condition `3 > 2` turns out to be true (or `True` for that matter). Let's try to make Python print "It works!". Change your code in your **python_intro.py** file to this:

python_intro.py

```
if 3 > 2:
    print('It works!')
```

Notice how we've indented the next line of code by 4 spaces? We need to do this so Python knows what code to run if the result is true. You can do one space, but nearly all Python programmers do 4 to make things look neat. A single `tab` will also count as 4 spaces.

Save it and give it another run:

command-line

```
$ python python_intro.py
It works!
```

What if a condition isn't True?

In previous examples, code was executed only when the conditions were True. But Python also has `elif` and `else` statements:

python_intro.py

```
if 5 > 2:
    print('5 is indeed greater than 2')
else:
    print('5 is not greater than 2')
```

When this is run it will print out:

command-line

```
$ python python_intro.py
5 is indeed greater than 2
```

If 2 were a greater number than 5, then the second command would be executed. Easy, right? Let's see how `elif` works:

python_intro.py

```
name = 'Billy'
if name == 'Wes':
    print('Hey Wes!')
elif name == 'Billy':
    print('Hey Billy!')
else:
    print('Hey anonymous!')
```

and executed:

command-line

```
$ python python_intro.py
Hey Billy!
```

See what happened there? `elif` lets you add extra conditions that run if the previous conditions fail.

You can add as many `elif` statements as you like after your initial `if` statement. For example:

python_intro.py

```
volume = 57
if volume < 20:
    print("It's kinda quiet.")
elif 20 <= volume < 40:
    print("It's nice for background music")
elif 40 <= volume < 60:
    print("Perfect, I can hear all the details")
elif 60 <= volume < 80:
    print("Nice for parties")
elif 80 <= volume < 100:
    print("A bit loud!")
else:
    print("My ears are hurting! :(")
```

Python runs through each test in sequence and prints:

command-line

```
$ python python_intro.py
Perfect, I can hear all the details
```

Comments

Comments are lines beginning with `#`. You can write whatever you want after the `#` and Python will ignore it. Comments can make your code easier for other people to understand.

Let's see how that looks:

python_intro.py

```
# Change the volume if it's too loud or too quiet
if volume < 20 or volume > 80:
    volume = 50
    print("That's better!")
```

You don't need to write a comment for every line of code, but they are useful for explaining why your code is doing something, or providing a summary when it's doing something complex.

Summary

In the last few exercises you learned about:

- **comparing things** – in Python you can compare things by using `>`, `>=`, `==`, `<=`, `<` and the `and`, `or` operators
- **Boolean** – a type of object that can only have one of two values: `True` or `False`
- **Saving files** – storing code in files so you can execute larger programs.
- **if … elif … else** – statements that allow you to execute code only when certain conditions are met.
- **comments -** lines that Python won't run which let you document your code

Time for the last part of this chapter!

Your own functions!

> For readers at home: this part is covered in the Python Basics: Functions video.

Remember functions like `len()` that you can execute in Python? Well, good news – you will learn how to write your own functions now!

A function is a sequence of instructions that Python should execute. Each function in Python starts with the keyword `def`, is given a name, and can have some parameters. Let's start with an easy one. Replace the code in **python_intro.py** with the following:

python_intro.py

```
def hi():
    print('Hi there!')
    print('How are you?')

hi()
```

Okay, our first function is ready!

You may wonder why we've written the name of the function at the bottom of the file. This is because Python reads the file and executes it from top to bottom. So in order to use our function, we have to re-write it at the bottom.

Let's run this now and see what happens:

command-line

```
$ python python_intro.py
Hi there!
How are you?
```

That was easy! Let's build our first function with parameters. We will use the previous example – a function that says 'hi' to the person running it – with a name:

python_intro.py

```
def hi(name):
```

As you can see, we now gave our function a parameter that we called `name`:

python_intro.py

```
def hi(name):
    if name == 'Wes':
        print('Hi Wes!')
    elif name == 'Billy':
        print('Hi Billy!')
    else:
        print('Hi anonymous!')

hi()
```

Remember: The `print` function is indented four spaces within the `if` statement. This is because the function runs when the condition is met. Let's see how it works now:

command-line

```
$ python python_intro.py
Traceback (most recent call last):
File "python_intro.py", line 10, in <module>
  hi()
TypeError: hi() missing 1 required positional argument: 'name'
```

Oops, an error. Luckily, Python gives us a pretty useful error message. It tells us that the function `hi()` (the one we defined) has one required argument (called `name`) and that we forgot to pass it when calling the function. Let's fix it at the bottom of the file:

python_intro.py

```
hi("Wes")
```

And run it again:

command-line

```
$ python python_intro.py
Hi Wes!
```

And if we change the name?

python_intro.py

```
hi("Billy")
```

And run it:

command-line

```
$ python python_intro.py
Hi Billy!
```

Now, what do you think will happen if you write another name in there? (Not Wes or Billy.) Give it a try and see if you're right. It should print out this:

command-line

```
Hi anonymous!
```

This is awesome, right? This way you don't have to repeat yourself every time you want to change the name of the person the function is supposed to greet. And that's exactly why we need functions – you never want to repeat your code!

Let's do something smarter – there are more names than two, and writing a condition for each would be hard, right?

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')

hi("Richard")
```

Let's call the code now:

command-line

```
$ python python_intro.py
Hi Richard!
```

Congratulations! You just learned how to write functions! :)

Loops

> For readers at home: this part is covered in the Python Basics: For Loop video.

This is the last part already. That was quick, right? :)

Programmers don't like to repeat themselves. Programming is all about automating things, so we don't want to greet every person by their name manually, right? That's where loops come in handy.

Still remember lists? Let's do a list of girls:

python_intro.py

```
boys = ['Rick', 'Manny', 'Phil', 'Orin', 'You']
```

We want to greet all of them by their name. We have the `hi` function to do that, so let's use it in a loop:

python_intro.py

```
for name in boys:
```

The `for` statement behaves similarly to the `if` statement; code below both of these need to be indented four spaces.

Here is the full code that will be in the file:

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')

boys = ['Rick', 'Manny', 'Phil', 'Orin', 'You']
for name in boyls:
    hi(name)
    print('Next boy')
```

And when we run it:

command-line

```
$ python python_intro.py
Hi Rick!
Next boy
Hi Manny!
Next boy
Hi Phil!
Next boy
Hi Orin!
Next boy
Hi You!
Next boy
```

As you can see, everything you put inside a `for` statement with an indent will be repeated for every element of the list `boys`

You can also use `for` on numbers using the `range` function:

python_intro.py

```
for i in range(1, 6):
    print(i)
```

Which would print:

command-line

```
1
2
3
4
5
```

`range` is a function that creates a list of numbers following one after the other (these numbers are provided by you as parameters).

Note that the second of these two numbers is not included in the list that is output by Python (meaning `range(1, 6)` counts from 1 to 5, but does not include the number 6). That is because "range" is half-open, and by that we mean it includes the first value, but not the last.

Summary

That's it. **You totally rock!** This was a tricky chapter, so you should feel proud of yourself. We're definitely proud of you for making it this far!

You might want to briefly do something else – stretch, walk around for a bit, rest your eyes – before going on to the next chapter. :)

What is Bottle?

Bottle is a free and open source web application framework, written in Python. A web framework is a set of components that helps you to develop websites faster and easier.

When you're building a website, you always need a similar set of components: a way to handle user authentication (signing up, signing in, signing out), a management panel for your website, forms, a way to upload files, etc.

Luckily for you, other people long ago noticed that web developers face similar problems when building a new site, so they teamed up and created frameworks (Bottle being one of them) that give you ready-made components to use.

Frameworks exist to save you from having to reinvent the wheel and to help alleviate some of the overhead when you’re building a new site.

Why do you need a framework?

To understand what Bottle is actually for, we need to take a closer look at the servers. The first thing is that the server needs to know that you want it to serve you a web page.

Imagine a mailbox (port) which is monitored for incoming letters (requests). This is done by a web server. The web server reads the letter and then sends a response with a webpage. But when you want to send something, you need to have some content. And Bottle is something that helps you create the content.

What happens when someone requests a website from your server?

When a request comes to a web server, it's passed to Bottle which tries to figure out what is actually requested. It takes a web page address first and tries to figure out what to do. This part is done by what's called a decorator function. It will look at the route being requested and match it against a series of functions which are decorated with routes.

Imagine a mail carrier with a letter. She is walking down the street and checks each house number against the one on the letter. If it matches, she puts the letter there. This is how the route decorator works!

In the *view* function, all the interesting things are done: we can look at a database to look for some information. Maybe the user asked to change something in the data? Like a letter saying, "Please change the description of my job." The *view* can check if you are allowed to do that, then update the job description for you and send back a message: "Done!" Then the *view* generates a response and Bottle can send it to the user's web browser.

Of course, the description above is a little bit simplified, but you don't need to know all the technical things yet. Having a general idea is enough.

So instead of diving too much into details, we will simply start creating something with Bottle and we will learn all the important parts along the way!

Bottle installation

> **Note** If you're using a Chromebook, skip this chapter and make sure you follow the Chromebook Setup instructions.
>
> **Note** If you already worked through the Installation steps then you've already done this – you can go straight to the next chapter!

Virtual environment

Remember, if you're using a Chromebook, you already did this step.

Before we install Bottle we will get you to install an extremely useful tool to help keep your coding environment tidy on your computer. It's possible to skip this step, but it's highly recommended. Starting with the best possible setup will save you a lot of trouble in the future!

So, let's create a **virtual environment** (also called a *virtualenv*). Virtualenv will isolate your Python setup on a per-project basis. This means that any changes you make to one website won't affect any others you're also developing. Neat, right?

All you need to do is find a directory in which you want to create the `virtualenv`; your home directory, for example. On Windows it might look like `C:\Users\Name\` (where `Name` is the name of your login).

> **NOTE:** On Windows, make sure that this directory does not contain accented or special characters; if your username contains accented characters, use a different directory, for example `C:\bottleboys`.

For this tutorial we will be using a new directory `bottleboys` from your home directory:

```
$ mkdir bottleboys
$ cd bottleboys
```

We will make a virtualenv called `myvenv`. The general command will be in the format:

```
$ python3 -m venv myvenv
```

Windows

To create a new `virtualenv`, you need to open the console (we told you about that a few chapters ago – remember?) and run `C:\Python35\python -m venv myvenv`. It will look like this:

```
C:\Users\Name\bottleboys> C:\Python35\python -m venv myvenv
```

where `C:\Python35\python` is the directory in which you previously installed Python and `myvenv` is the name of your `virtualenv`. You can use any other name, but stick to lowercase and use no spaces, accents or special characters. It is also good idea to keep the name short – you'll be referencing it a lot!

Linux and OS X

Creating a `virtualenv` on both Linux and OS X is as simple as running `python3 -m venv myvenv`. It will look like this:

```
$ python3 -m venv myvenv
```

`myvenv` is the name of your `virtualenv`. You can use any other name, but stick to lowercase and use no spaces. It is also good idea to keep the name short as you'll be referencing it a lot!

> **NOTE:** On some versions of Debian/Ubuntu you may receive the following error:
>
> ```
> The virtual environment was not created successfully because ensurepip is not available.
>  On Debian/Ubuntu systems, you need to install the python3-venv package using the followi
> ng command.
>    apt-get install python3-venv
> You may need to use sudo with that command.  After installing the python3-venv package, r
> ecreate your virtual environment.
> ```
>
> In this case, follow the instructions above and install the `python3-venv` package:
>
> ```
> $ sudo apt-get install python3-venv
> ```
>
> **NOTE:** On some versions of Debian/Ubuntu initiating the virtual environment like this currently gives the following error:
>
> ```
> Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade
> ', '--default-pip']' returned non-zero exit status 1
> ```
>
> To get around this, use the `virtualenv` command instead.
>
> ```
> $ sudo apt-get install python-virtualenv
> $ virtualenv --python=python3.5 myvenv
> ```
>
> **NOTE:** If you get an error like
>
> ```
> E: Unable to locate package python3-venv
> ```
>
> then instead run:
>
> ```
> sudo apt install python3.5-venv
> ```

Working with virtualenv

The command above will create a directory called `myvenv` (or whatever name you chose) that contains our virtual environment (basically a bunch of directory and files).

Windows

Start your virtual environment by running:

```
C:\Users\Name\bottleboys> myvenv\Scripts\activate
```

> **NOTE:** on Windows 10 you might get an error in the Windows PowerShell that says `execution of scripts is disabled on this system`. In this case, open another Windows PowerShell with the "Run as Administrator" option. Then try typing the following command before starting your virtual environment:
>
> ```
> C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
>     Execution Policy Change
>     The execution policy helps protect you from scripts that you do not trust. Changing t
> he execution policy might expose you to the security risks described in the about_Executi
> on_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to c
> hange the execution policy? [Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend
> [?] Help (default is "N"): A
> ```

Linux and OS X

Start your virtual environment by running:

```
$ source myvenv/bin/activate
```

Remember to replace `myvenv` with your chosen `virtualenv` name!

> **NOTE:** sometimes `source` might not be available. In those cases try doing this instead:
>
> ```
> $ . myvenv/bin/activate
> ```

You will know that you have `virtualenv` started when you see that the prompt in your console is prefixed with `(myvenv)`.

When working within a virtual environment, `python` will automatically refer to the correct version so you can use `python` instead of `python3`.

OK, we have all important dependencies in place. We can finally install Django!

Installing Bottle

Now that you have your `virtualenv` started, you can install Bottle.

Before we do that, we should make sure we have the latest version of `pip`, the software that we use to install Bottle:

```
(myvenv) ~$ pip install --upgrade pip
```

Then run `pip install bottle` to install Bottle.

That's it! You're now (finally) ready to create a Bottle application!

Your first Bottle project!

Eventually we're going to create a blog, but first, we will create something very simple. We are going to create a "Hello World" web page. Working through these instructions will allow us to understand the basic structure of a Bottle application and we will get to see all of the basic routing principles it takes to be successful.

Bottle is a very lightweight framework, most of our code will go into one single file, which we will name `app.py`. If you go to the Bottle site you can see much of what we will cover in this initial section.

Make a file called `app.py` in your `bottleboys` directory.

app.py

```
from sys import argv
from bottle import route, run

@route('/')
def index():
  return "Hello World: A Message in a Bottle App"

run(host="0.0.0.0", port=argv[1], debug=True)
```

> Remember to run everything in the virtualenv. If you don't see a prefix `(myvenv)` in your console, you need to activate your virtualenv. We explained how to do that in the **Bottle installation** chapter in the **Working with virtualenv** part. Typing `myvenv\Scripts\activate` on Windows or `source myvenv/bin/activate` on Mac OS X or Linux will do this for you.

And we're done! Time to start the web server and see if our website is working!

Starting the web server

You need to be in the directory that contains the `app.py` file (the `bottleboys` directory). In the console, we can start the web server by running `python app.py` with the port number as an argument. So go back to the terminal and run this:

command-line

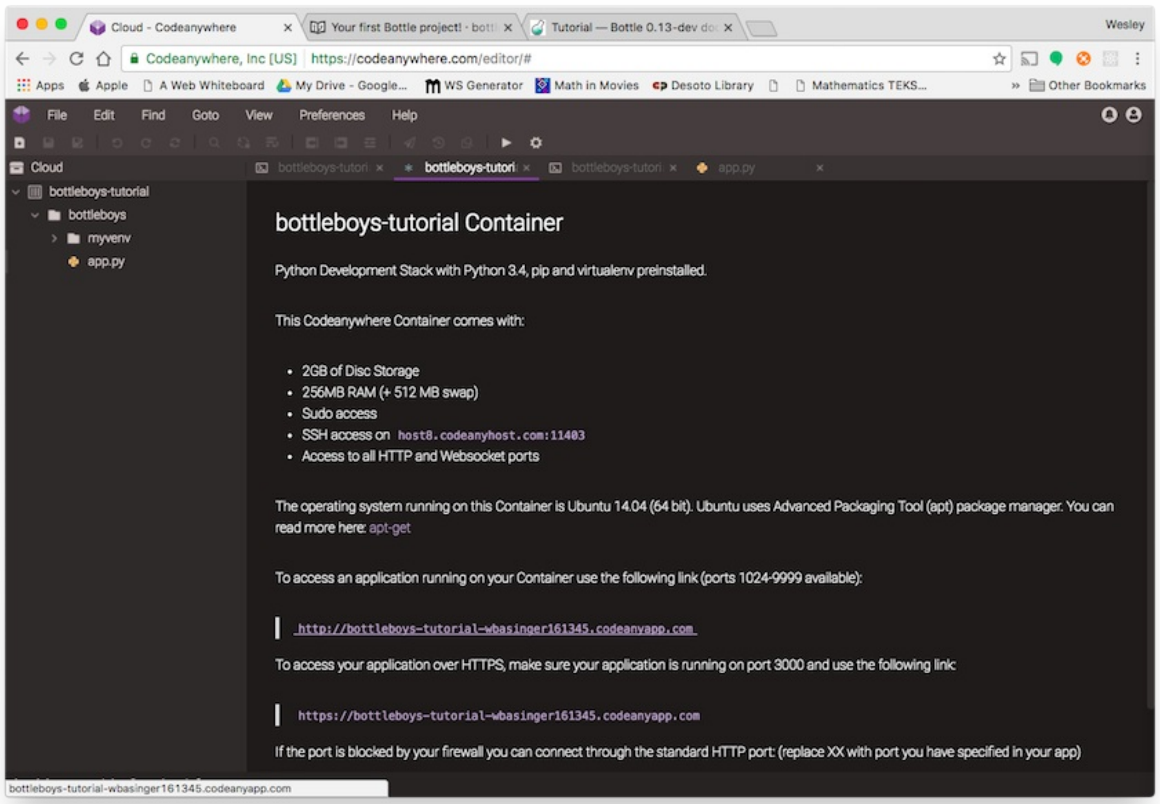
```
(myvenv) ~/bottleboys$ python app.py 8080
```

Now all you need to do is check that your website is running. Open your browser (Firefox, Chrome, Safari, Internet Explorer or whatever you use) and enter this address:

browser

```
http://0.0.0.0:8080/
```

If you're running the code on CodeAnywhere, then you'll want to pull up the info page, click on the first link and append the port number with a colon to the end. I've included an example for mine below.



bottleboys-tutorial Container

Python Development Stack with Python 3.4, pip and virtualenv preinstalled.

This Codeanywhere Container comes with:

- 2GB of Disc Storage
- 256MB RAM (+ 512 MB swap)
- Sudo access
- SSH access on `host8.codeanyhost.com:11403`
- Access to all HTTP and Websocket ports

The operating system running on this Container is Ubuntu 14.04 (64 bit). Ubuntu uses Advanced Packaging Tool (apt) package manager. You can read more here: apt-get

To access an application running on your Container use the following link (ports 1024-9999 available):

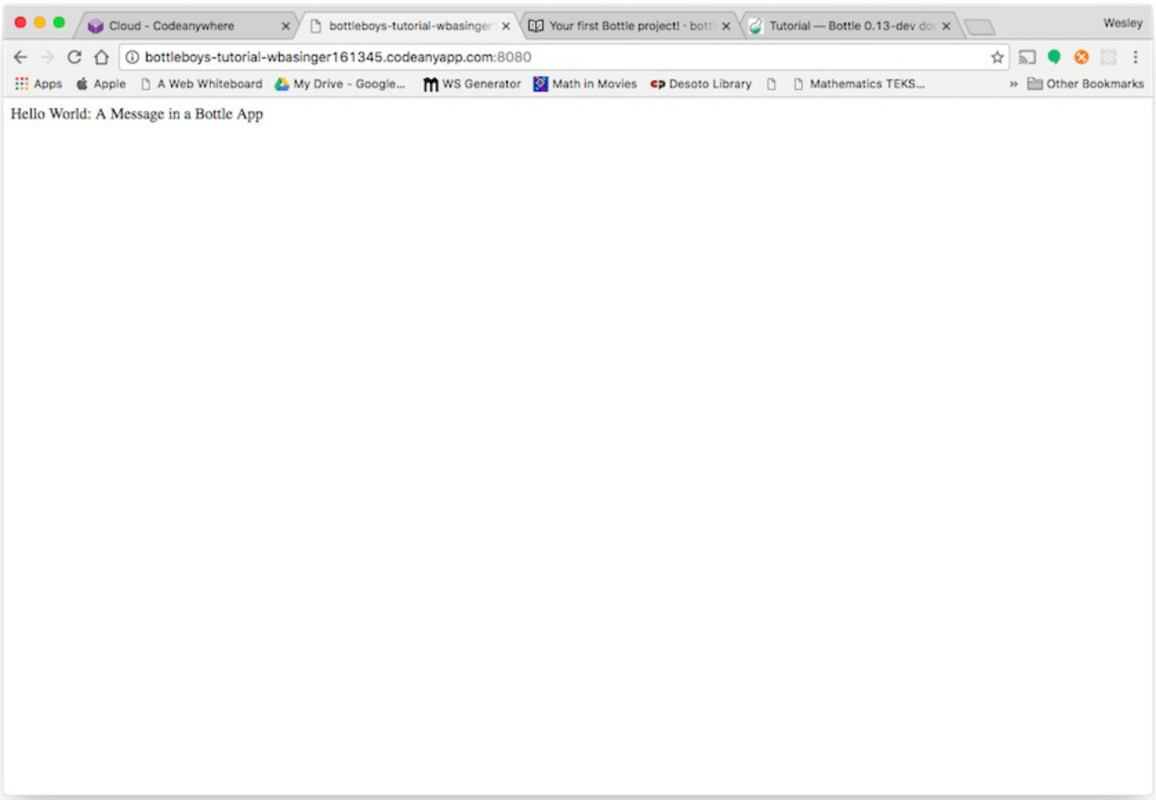
`http://bottleboys-tutorial-wbasinger161345.codeanyapp.com`

To access your application over HTTPS, make sure your application is running on port 3000 and use the following link:

`https://bottleboys-tutorial-wbasinger161345.codeanyapp.com`

If the port is blocked by your firewall you can connect through the standard HTTP port: (replace XX with port you have specified in your app)

Congratulations! You've just created your first website and run it using a web server! Isn't that awesome?


Cloud - Codeanywhere
bottleboys-tutorial-wbasinger
Your first Bottle project! · bott
Tutorial — Bottle 0.13-dev do
Wesley
bottleboys-tutorial-wbasinger161345.codeanyapp.com:8080
Apps
Apple
A Web Whiteboard
My Drive - Google...
WS Generator
Math in Movies
Desoto Library
Mathematics TEKS...
Other Bookmarks

```
Hello World: A Message in a Bottle App
```

While the web server is running, you won't see a new command-line prompt to enter additional commands. The terminal will accept new text but will not execute new commands. This is because the web server continuously runs in order to listen for incoming requests.

> We reviewed how web servers work in the **How the Internet works** chapter.

To type additional commands while the web server is running, open a new terminal window and activate your virtualenv. To stop the web server, switch back to the window in which it's running and press CTRL+C - Control and C buttons together (on Windows, you might have to press Ctrl+Break).

Ready for the next step? It's time to create some content!

JSON data

What we want to create now is something that will store all the posts in our blog. But to be able to do that we need to talk a little bit about data.

Data

There are lots of different ways to store and retrieve information on the web, but one of the most popular and easy to use is JSON. JSON stands for Java Script Object Notation. JSON is easy to read and easy to process. For this tutorial, we will be persisting our data to a file and reading from it with a Python package called TinyDB. This will allow us to bypass the complexity of databases, for now. It is not a typical way of handling data and you will need to learn more robust technologies if you ever build a larger application.

JSON is pretty flexible on schema (database design), but we'll keep it simple with some basic key-value pairings. Look below how I take a table of values and convert it to JSON. This is a little more nested than a normal JSON structure, but it's going to work well with TinyDB.

```
NAME      AGE      GENDER
---------------------------
Wes       33       male
Rick      22       male

{
  "people" :
    {
      "1" :
        {
          "name": "Wes",
          "age": 33,
          "gender" : "male"
        },
      "2" :
        {
          "name" : "Rick",
          "age" : 22,
          "gender" : "male"
        }
    }
}
```

Notice the similarities between JSON and a Python dictionary, that will certainly help as we learn to access and modify the data inside the database file.

Time to make our database file. In your `bottleboys` project directory, create this file. Feel free to change the values for author, date and body. Save it when you are finished.

db.json

```
{
  "posts" :
    {
      "1" :
        {
          "postId" : "1",
          "author" : "Wes Basinger",
          "title" : "First Post",
          "date" : "2016-12-21",
          "body" : "This is my first post with of my new blog, I hope you like it!"
        },
      "2" :
        {
          "postId" : "2",
          "author" : "Wes Basinger",
          "title" : "Second Post",
          "date" : "2016-12-24",
          "body" : "It's Christmas Eve, I'm so excited!"
        }
    }
}
```

This is what is called seed data. Later on, we will write our application in such a way that we can add posts dynamically.

Accessing our data through the shell

Just to get a feel for how our data is stored and how we can access it, we'll use the Python shell to access it. TinyDB has some easy methods to work with. First `pip install tinydb` on the command line. Should look something like this.

command line

```
(myvenv)cabox@box-codeanywhere:~/workspace/bottleboys$ pip install tinydb

Downloading/unpacking tinydb

  Downloading tinydb-3.2.1.zip
  Running setup.py (path:/home/cabox/workspace/bottleboys/myvenv/build/tinydb/setup.py) egg_inf
o for package tinydb
Installing collected packages: tinydb
  Running setup.py install for tinydb
Successfully installed tinydb
Cleaning up...
(myvenv)cabox@box-codeanywhere:~/workspace/bottleboys$
```

Then you can open up a Python shell and run the following set of commands.

python shell

```
>>> from tinydb import TinyDB
>>> db = TinyDB('db.json')
>>> posts = db.table("posts")
>>> posts.all()
[{u'date': u'2016 12 21', u'body': u'This is my first post with of my new blog, I hope you like
 it!', u'postId': u'1', u'author': u'Wes Basinger', u'title': u'First Post'}, {u'date': u'2016
12 24', u'body': u"It's Christmas Eve, I'm so excited!", u'postId': u'2', u'author': u'Wes Basi
nger', u'title': u'Second Post'}]
```

And now you can see, all of our posts made it into the database. TinyDB has lots of methods for modifying our database, all of them are CRUD operations.

CRUD stands for create, read, update, and delete. These are the basic things we will need to do with our data. For now, we won't need to think about the database until we get into some topics involving different routes for your web app.

Deploy!

> **Note** The following chapter can be sometimes a bit hard to get through. Persist and finish it; deployment is an important part of the website development process. This chapter is placed in the middle of the tutorial so that your mentor can help with the slightly trickier process of getting your website online. This means you can still finish the tutorial on your own if you run out of time.

Until now, your website was only available on your computer. Now you will learn how to deploy it! Deploying is the process of publishing your application on the Internet so people can finally go and see your app. :)

As you learned, a website has to be located on a server. There are a lot of server providers available on the internet. We will use one that has a relatively simple deployment process: Heroku. Heroku is free for small applications that don't have too many visitors so it'll definitely be enough for you now.

The other external service we'll be using is GitHub, which is a code hosting service. There are others out there, but almost all programmers have a GitHub account these days, and now so will you!

These three places will be important to you. Your local computer will be the place where you do development and testing. When you're happy with the changes, you will place a copy of your program on GitHub. Your website will be on Heroku and you will update it by pushing a new copy of your code.

Git

> **Note** If you already did the Installation steps, there's no need to do this again – you can skip to the next section and start creating your Git repository.

Git is a "version control system" used by a lot of programmers. This software can track changes to files over time so that you can recall specific versions later. A bit like the "track changes" feature in Microsoft Word, but much more powerful.

Installing Git

Windows

You can download Git from git-scm.com. You can hit "next" on all steps except for one; in the fifth step entitled "Adjusting your PATH environment", choose "Use Git and optional Unix tools from the Windows Command Prompt" (the bottom option). Other than that, the defaults are fine. Checkout Windows-style, commit Unix-style line endings is good.

OS X

Download Git from git-scm.com and just follow the instructions.

> **Note** If you are running OS X 10.6, 10.7, or 10.8, you will need to install the version of git from here: Git installer for OS X Snow Leopard

Starting our Git repository

Git tracks changes to a particular set of files in what's called a code repository (or "repo" for short). Let's start one for our project. Open up your console and run these commands, in the `bottleboys` directory:

> **Note** Check your current working directory with a `pwd` (Mac OS X/Linux) or `cd` (Windows) command before initializing the repository. You should be in the `bottleboys` folder.

command-line

```
$ git init
Initialized empty Git repository in ~/bottleboys/.git/
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

Initializing the git repository is something we need to do only once per project (and you won't have to re-enter the username and email again ever).

Git will track changes to all the files and folders in this directory, but there are some files we want it to ignore. We do this by creating a file called `.gitignore` in the base directory. Open up your editor and create a new file with the following contents:

.gitignore

```
*.pyc
*~
__pycache__
myvenv
.DS_Store
db.json
```

And save it as `.gitignore` in the "bottleboys" folder.

> **Note** The dot at the beginning of the file name is important! If you're having any difficulty creating it (Macs don't like you to create files that begin with a dot via the Finder, for example), then use the "Save As" feature in your editor; it's bulletproof.

It's a good idea to use a `git status` command before `git add` or whenever you find yourself unsure of what has changed. This will help prevent any surprises from happening, such as wrong files being added or committed. The `git status` command returns information about any untracked/modified/staged files, the branch status, and much more. The output should be similar to the following:

command-line

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        app.py

nothing added to commit but untracked files present (use "git add" to track)
```

And finally we save our changes. Go to your console and run these commands:

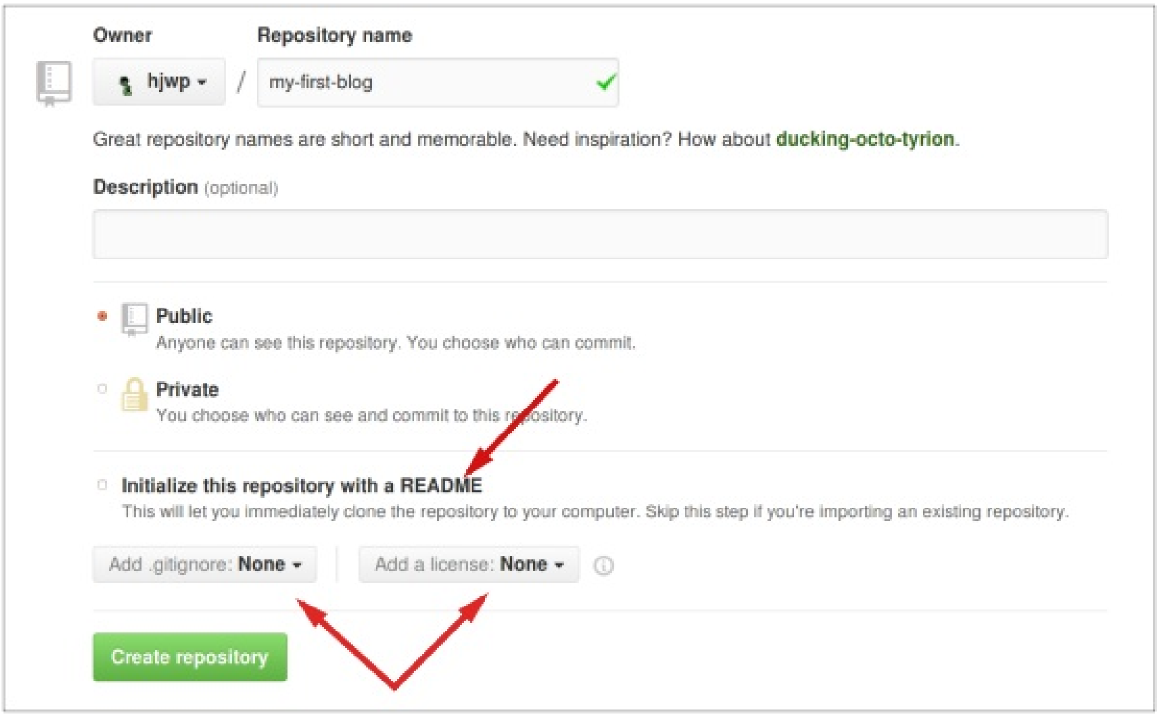
command-line

```
$ git add --all .
$ git commit -m "My Bottle Boys app, first commit"
  2 files changed, 14 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 app.py
```

Pushing your code to GitHub

Go to GitHub.com and sign up for a new, free user account. (If you already did that in the workshop prep, that is great!)

Then, create a new repository, giving it the name "my-first-blog". Leave the "initialize with a README" checkbox unchecked, leave the .gitignore option blank (we've done that manually) and leave the License as None.



Owner: hjwp / Repository name: my-first-blog

Great repository names are short and memorable. Need inspiration? How about ducking-octo-tyrion.

Description (optional)

[✓ Public] Anyone can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

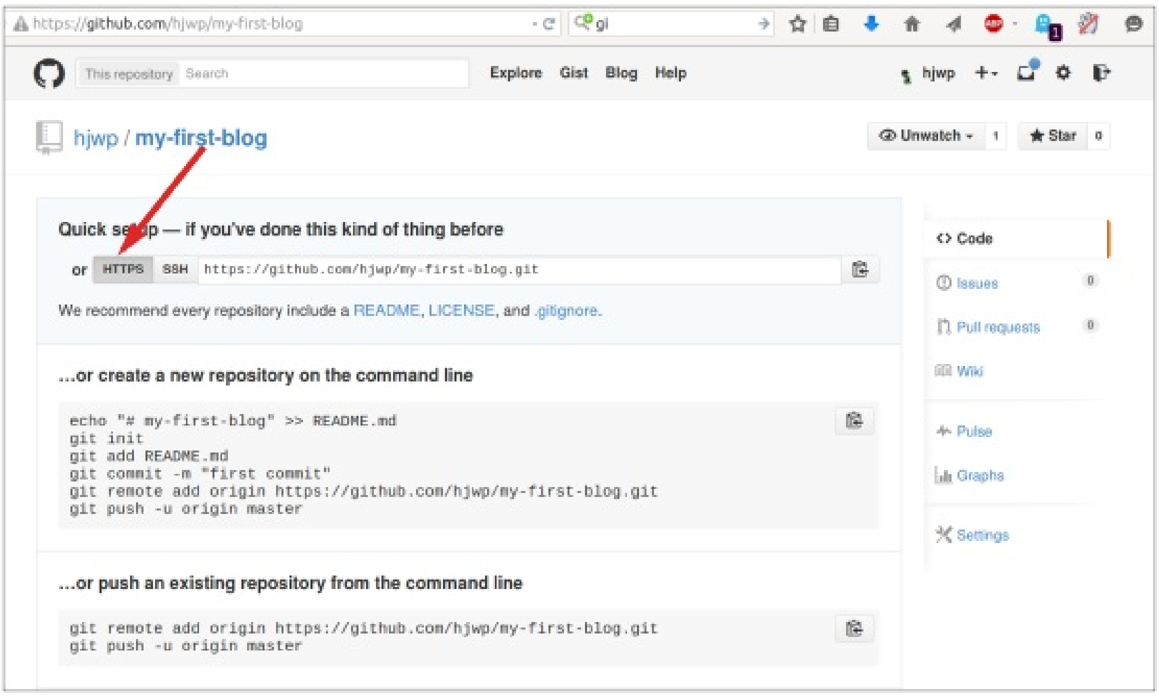
Initialize this repository with a README This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

> **Note** The name `my-first-blog` is important – you could choose something else, but it's going to occur lots of times in the instructions below, and you'd have to substitute it each time. It's probably easier to just stick with the name `my-first-blog` .

On the next screen, you'll be shown your repo's clone URL. Choose the "HTTPS" version, copy it, and we'll paste it into the terminal shortly:


https://github.com/hjwp/my-first-blog
This repository
Search
Explore
Gist
Blog
Help
hjwp
hjwp / my-first-blog
Unwatch 1
Star 0
Quick setup — if you've done this kind of thing before
or HTTPS SSH https://github.com/hjwp/my-first-blog.git
We recommend every repository include a README, LICENSE, and .gitignore.
...or create a new repository on the command line
...or push an existing repository from the command line
Code
Issues 0
Pull requests 0
Wiki
Pulse
Graphs
Settings

```
echo "# my-first-blog" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/hjwp/my-first-blog.git
git push -u origin master
```

```
git remote add origin https://github.com/hjwp/my-first-blog.git
git push -u origin master
```

Now we need to hook up the Git repository on your computer to the one up on GitHub.

Type the following into your console (Replace `<your-github-username>` with the username you entered when you created your GitHub account, but without the angle-brackets):

command-line

```
$ git remote add origin https://github.com/<your-github-username>/my-first-blog.git
$ git push -u origin master
```

Enter your GitHub username and password and you should see something like this:

command-line

```
Username for 'https://github.com': hjwp
Password for 'https://hjwp@github.com':
Counting objects: 6, done.
Writing objects: 100% (6/6), 200 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/hjwp/my-first-blog.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Your code is now on GitHub. Go and check it out! You'll find it's in fine company – Django, the Bottle Boys Tutorial, and many other great open source software projects also host their code on GitHub. :)

Setting up our blog on Heroku

> **Note** You might have already created a Heroku account earlier during the install steps – if so, no need to do it again.

Next it's time to sign up for a free "Beginner" account on Heroku.

- www.heroku.com

After you sign up for a free account, you'll need to install the Heroku Toolbelt locally. This will allow you to provision your code to Heroku from the command line. Here are the steps if for installing it on the CodeAnywhere platform. You can find separate instructions for Mac, Windows and Linux platforms.

```
# Run this from your terminal.
# The following will add our apt repository and install the CLI:
wget -O- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

Pushing our code to Heroku

Make sure you have the Heroku toolbelt installed by typing `heroku --version` into the command line. Make sure it returns the version and not an error.

Heroku is easy to use, as long as you put a `requirements.txt` file in your working directory, it will recognize that you are trying to run a Python application, it will install all dependencies listed in `requirements.txt`. You can make the file by running this command.

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

You also need another file called `Procfile` in your working directory with the following contents. Basically, this file just tells Heroku which file to run for the app and which port to run it on. You only need one line.

Procfile

```
web: python ./app.py $PORT
```

Do one more `git add -A .` and `git commit` to make sure that all files are set and ready to go.

While in your `bottleboys` directory, run this command `heroku create <name of your app>`. I'm going to call my app bottleboys.

Now, run one more command `git push heroku master`. That command sends your code up to Heroku's cloud. If you type `heroku open`, you should be able to see your code running.

If you get an error, try running this command `heroku ps:scale web=1`

You are live!

The default page for your site should say "Hello World: A Message in a Bottle App", just like it does on your local computer.

You can go back to your local setup. From here you should work on your local setup to make changes. This is a common workflow in web development – make changes locally, push those changes to GitHub, and push your changes up to Heroku. This allows you to work and experiment without breaking your live Web site. Pretty cool, huh?

Give yourself a *HUGE* pat on the back! Server deployments are one of the trickiest parts of web development and it often takes people several days before they get them working. But you've got your site live, on the real Internet, just like that!

Bottle Routes

We're about to build our first page: a landing site for your blog! But first, let's learn a little bit about Bottle routes.

What is a Route?

A route is simply a web address. You can see a route every time you visit a website – it is visible in your browser's address bar. (Yes! `127.0.0.1:8000` is a URL! And `https://wesbasinger.github.io` is also a URL.)

Every page on the Internet needs its own route. This way your application knows what it should show to a user who opens that route. In Bottle we use something called a decorator function. A decorator function basically will call another function with some magic behind the scenes.

How do routes work in Bottle?

Let's open up the `app.py` file in your code editor of choice and see what it looks like:

app.py

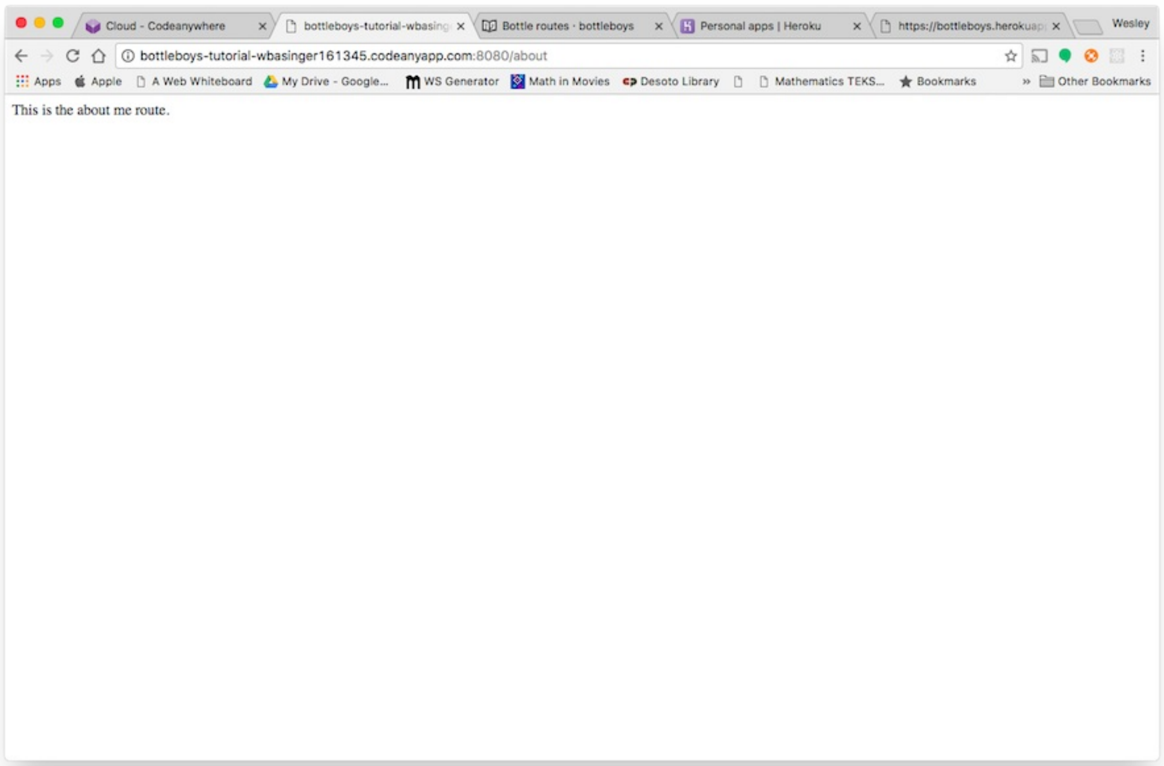
```
from sys import argv
from bottle import route, run

@route('/')
def index():
  return "Hello World: A Message in a Bottle App"

run(host="0.0.0.0", port=argv[1], debug=True)
```

That's exactly how we left it. It's typical to name the home route as index, so we'll create another route for an about me page. Modify the existing code so that it looks something like this...

app.py

```
from sys import argv
from bottle import route, run

@route('/')
def index():
  return "This is the index route."

@route('/about')
def about():
  return "This is the about me route."

run(host="0.0.0.0", port=argv[1], debug=True)
```

Now fire up your server again with `python app.py 8080` and visit `0.0.0.0:8080/about`. You should see the new route. Bottle is smart enough to read the URL and send the reader where you want them to go! There's a sample of how it should look below.



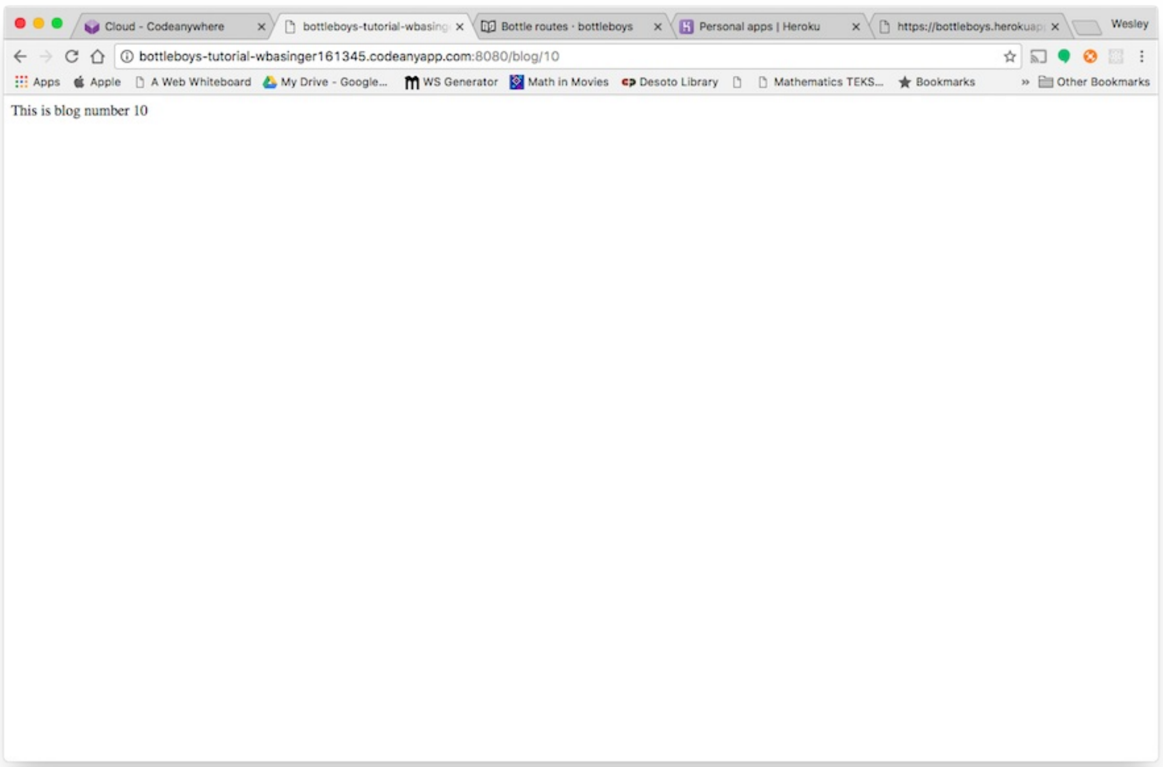
This is the about me route.

Bottle is even smart enough to read what are called parameters from the URL. Let's stub out a route for individual blog posts. Modify `app.py` like so.

app.py

```
from sys import argv
from bottle import route, run

@route('/')
def index():
  return "This is the index route."

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  return "This is blog number " + post_number

run(host="0.0.0.0", port=argv[1], debug=True)
```

Now navigate to `0.0.0.0:8080/blog/10`. It should say "This is blog number 10". Bottle is reading the route parameter from the URL! That is extremely helpful. It allows us to pass information like variables between client and server. There's a sample below of what you should see.


bottleboys-tutorial-wbasinger161345.codeanyapp.com:8080/blog/10

This is blog number 10

I think that's all we really need to cover for right now, but you should know that you can include multiple parameters in the route and do different types of requests to the same URL, as long as you change the route method.

Bottle views – time to create!

A *view* is a place where we put the "logic" of our application. It will request information from the database you created before and pass it to a `template`. We'll create a template in the next chapter. Views are just Python functions that are a little bit more complicated than the ones we wrote in the **Introduction to Python** chapter.

Views are placed in the `app.py` file. We will just add logic to the existing routes instead of returning plain text.

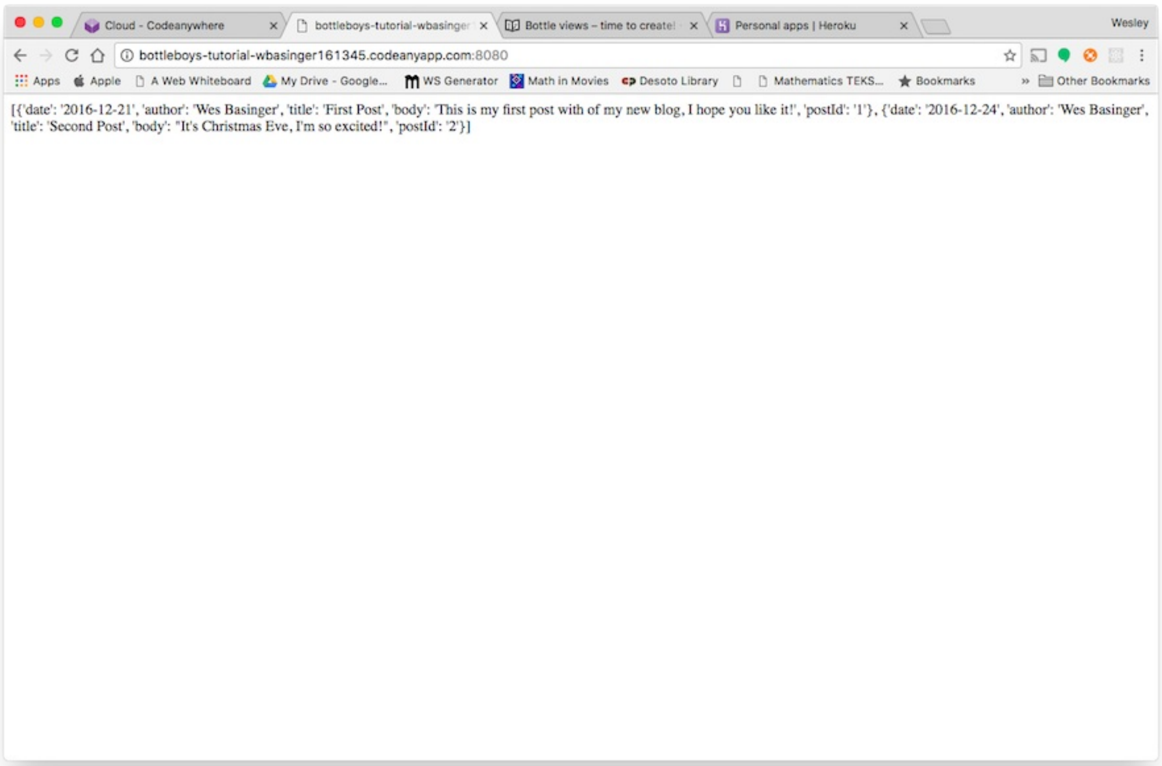
app.py

We are going to write some code that will return all posts to us in the index route. So instead of seeing Hello World: Message from Bottle App, we will see all the raw text data from the database file that we seeded earlier. Modify the `app.py` in the following manner.

app.py

```
from sys import argv
from bottle import route, run
from tinydb import TinyDB, Query # This line is new!!!

@route('/')
def index():
  # ALL of this is new!!!
  db = TinyDB("db.json")
  posts = db.table("posts")
  return str(posts.all())

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  return "This is blog number " + post_number

run(host="0.0.0.0", port=argv[1], debug=True)
```

Now run the server and visit the homepage of your web app. You'll see all the data. It's not pretty, but we'll do something about that in the next section. For now, be happy we know how to retrieve data! I've include a sample below of how it should look.



```
[{'date': '2016-12-21', 'author': 'Wes Basinger', 'title': 'First Post', 'body': 'This is my first post with of my new blog, I hope you like it!', 'postId': '1'}, {'date': '2016-12-24', 'author': 'Wes Basinger',
'title': 'Second Post', 'body': "It's Christmas Eve, I'm so excited!", 'postId': '2'}]
```

Introduction to HTML

What's a template, you may ask?

A template is a file that we can re-use to present different information in a consistent format – for example, you could use a template to help you write a letter, because although each letter might contain a different message and be addressed to a different person, they will share the same format.

A Bottle template's format is described in a language called HTML (that's the HTML we mentioned in the first chapter, **How the Internet works**).

What is HTML?

HTML is a simple code that is interpreted by your web browser – such as Chrome, Firefox or Safari – to display a web page for the user.

HTML stands for "HyperText Markup Language". **HyperText** means it's a type of text that supports hyperlinks between pages. **Markup** means we have taken a document and marked it up with code to tell something (in this case, a browser) how to interpret the page. HTML code is built with **tags**, each one starting with `<` and ending with `>`. These tags represent markup **elements**.

Your first template!

Creating a template means creating a template file. Everything is a file, right? You have probably noticed this already.

Templates are saved in `views` directory. So first create a directory called `views` inside your project directory. Then your project directory should look like this.

```
bottleboys
└───/views
└───app.py
└───Procfile
└───db.json
└───/myvenv
└───requirements.txt
```

And now create a `index.html` file inside the `views` directory.

Add the following to your template file:

views/index.html

```
<html>
    <h1>Hi there!</h1>
    <p>It works!</p>
</html>
```

Save the file. Now you need to go to `app.py` and instruct the Bottle route to use the new template.

app.py

```
from sys import argv
from bottle import route, run, template # This line is new!!!
from tinydb import TinyDB, Query

@route('/')
def index():
  ''' # COMMENT THIS OUT (FOR NOW)
  db = TinyDB("db.json")
  posts = db.table("posts")
  '''
  return template('index.tpl')

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  return "This is blog number " + str(post_number)

run(host="0.0.0.0", port=argv[1], debug=True)
```

So how does your website look now? Visit it to find out: http://0.0.0.0:8000/

It worked! Nice work there :) It should look something like the picture below.

- The most basic tag, `<html>`, is always the beginning of any web page and `</html>` is always the end. As you can see, the whole content of the website goes between the beginning tag `<html>` and closing tag `</html>`
- `<p>` is a tag for paragraph elements; `</p>` closes each paragraph

Head and body

Each HTML page is also divided into two elements: **head** and **body**.

- **head** is an element that contains information about the document that is not displayed on the screen.
- **body** is an element that contains everything else that is displayed as part of the web page.

We use `<head>` to tell the browser about the configuration of the page, and `<body>` to tell it what's actually on the page.

For example, you can put a web page title element inside the `<head>`, like this:

views/index.html

```
<html>
    <head>
        <title>Blog of Wes</title>
    </head>
    <body>
        <p>Hi there!</p>
        <p>It works!</p>
    </body>
</html>
```

Notice how the browser has understood that "Blog of Wes" is the title of your page? It has interpreted `<title>Blog of Wes</title>` and placed the text in the title bar of your browser (it will also be used for bookmarks and so on).

Probably you have also noticed that each opening tag is matched by a *closing tag*, with a `/`, and that elements are *nested* (i.e. you can't close a particular tag until all the ones that were inside it have been closed too).

It's like putting things into boxes. You have one big box, `<html></html>`; inside it there is `<body></body>`, and that contains still smaller boxes: `<p></p>`.

You need to follow these rules of *closing* tags, and of *nesting* elements – if you don't, the browser may not be able to interpret them properly and your page will display incorrectly.

Customize your template

You can now have a little fun and try to customize your template! Here are a few useful tags for that:

- `<h1>A heading</h1>` for your most important heading
- `<h2>A sub-heading</h2>` for a heading at the next level
- `<h3>A sub-sub-heading</h3>` …and so on, up to `<h6>`
- `<p>A paragraph of text</p>`
- `<em>text</em>` emphasizes your text
- `<strong>text</strong>` strongly emphasizes your text
- `<br />` goes to another line (you can't put anything inside br)
- `<a href="https://wesbasinger.github.io">link</a>` creates a link
- `<ul><li>first item</li><li>second item</li></ul>` makes a list, just like this one!
- `<div></div>` defines a section of the page

Here's an example of a full template, copy and paste it into `views/index.html`:

views/index.html

```
<html>
    <head>
        <title>Bottle Boys blog</title>
    </head>
    <body>
        <div>
            <h1><a href="">Bottle Boys Blog</a></h1>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My first post</a></h2>
            <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. D
onec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, torto
r mauris condimentum nibh, ut fermentum massa justo sit amet risus.</p>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My second post</a></h2>
            <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. D
onec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, torto
r mauris condimentum nibh, ut f.</p>
        </div>
    </body>
</html>
```

We've created three `div` sections here.

- The first `div` element contains the title of our blog – it's a heading and a link
- Another two `div` elements contain our blogposts with a published date, `h2` with a post title that is clickable and two `p` s (paragraph) of text, one for the date and one for our blogpost.

If you run the server again, you should see something like this.

Yaaay! But so far, our template only ever displays exactly **the same information** – whereas earlier we were talking about templates as allowing us to display **different** information in the **same format**.

What we really want to do is display real posts – and that's where we're going next.

One more thing: deploy!

It'd be good to see all this out and live on the Internet, right? Let's do another Heroku deploy:

Commit, and push your code up to Github

First off, let's see what files have changed since we last deployed:

command-line

```
$ git status
```

Make sure you're in the `bottleboys` directory and let's tell `git` to include all the changes within this directory:

command-line

```
$ git add --all .
```

> `--all` means that `git` will also recognize if you've deleted files (by default, it only recognizes new/modified files). Also remember (from chapter 3) that `.` means the current directory.

Before we upload all the files, let's check what `git` will be uploading (all the files that `git` will upload should now appear in green):

command-line

```
$ git status
```

We're almost there, now it's time to tell it to save this change in its history. We're going to give it a "commit message" where we describe what we've changed. You can type anything you'd like at this stage, but it's helpful to type something descriptive so that you can remember what you've done in the future.

command-line

```
$ git commit -m "Changed the HTML for the site."
```

> Make sure you use double quotes around the commit message.

Once we've done that, we upload (push) our changes up to GitHub:

command-line

```
$ git push
```

Push your new code up to Heroku, and reload your web app

command-line

```
$ git push heroku master
[...]
```

Your update should be live! Go ahead and refresh your website in the browser. Changes should be visible. :) Remember, you can use the `heroku open` command as a shortcut to open your website.

Dynamic data in templates

We have different pieces in place: the data is in `db.json`, we have `index` in `app.py` and the template added. But how will we actually make our posts appear in our HTML template? Because that is what we want to do – take some content (models saved in the database) and display it nicely in our template, right?

This is exactly what *views* are supposed to do: connect models and templates. In our `index` *view* we will need to take the models we want to display and pass them to the template. In a *view* we decide what (model) will be displayed in a template.

OK, so how will we achieve this?

We need to open our `app.py`. So far `index` *view* looks like this:

app.py

```
@route('/')
def index():
  '''
  db = TinyDB("db.json")
  posts = db.table("posts")
  '''
  return template('index.html')
```

But, there was a problem. None of the data from our database was actually showing up in the webpage. That was partially due to the fact that we were not sending the data to the template. Change `app.py` in the following manner to fix this error. Only modify the index route right now.

app.py

```
@route('/')
def index():
  db = TinyDB("db.json")
  posts = db.table("posts")
  return template('index.html', posts=list(posts.all()))
```

That's it! Time to go back to our template and display this QuerySet!

Note: Still nothing will have changed on your webpage, we still need to utilize template variables.

Bottle templates

Time to display some data! Bottle gives us some helpful built-in **template tags** for that.

What are template tags?

You see, in HTML, you can't really write Python code, because browsers don't understand it. They know only HTML. We know that HTML is rather static, while Python is much more dynamic.

Bottle template tags allow us to transfer Python-like things into HTML, so you can build dynamic websites faster and easier. Cool!

Display index template

In the previous chapter we gave our template a list of posts in the `posts` variable. Now we will display it in HTML.

To print a variable in Bottle templates, we use double curly brackets with the variable's name inside, like this:

```
{{ posts }}
```

Bottle understands posts is a list of objects. Remember from **Introduction to Python** how we can display lists? Yes, with for loops! In a Bottle template you do them like this:

views/index.html

```
<html>
    <head>
        <title>Bottle Boys blog</title>
    </head>
    <body>
        <div>
            <h1><a href="">Bottle Boys Blog</a></h1>
        </div>
        % for post in posts:
          <div>
            <p>published: {{post["date"]}}</p>
            <h2>{{post["title"]}}</h2>
            <h3>{{post["author"]}}</h3>
            <p>{{post["body"]}}</p>
          </div>
        % end
    </body>
</html>
```

Try this in your template.

Have you noticed that we used a slightly different notation this time (`{{ post["title"] }}` or `{{ post["body"] }})` ? We are accessing data in each of the fields defined in our blog model.

Fire up the server and see if it worked. If it did, it should look something like this:

One more thing

It'd be good to see if your website will still be working on the public Internet, right? Let's try deploying to Heroku again. Here's a recap of the steps…

- First, push your code to Github

command-line

```
$ git status
[...]
$ git add --all .
$ git status
[...]
$ git commit -m "Modified templates to display posts from database."
[...]
$ git push
```

- Then, run:

command-line

```
$ git push heroku master
[...]
```

- Finally, hop on over to your web app. Your update should be live! If you don't have blog posts on your Heroku site, that's OK. The databases on your local computer and Heroku don't sync with the rest of your files. In fact, there is no data on the Heroku version as of yet.

CSS – make it pretty!

Our blog still looks pretty ugly, right? Time to make it nice! We will use CSS for that.

What is CSS?

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of a website written in a markup language (like HTML). Treat it as make-up for our web page. ;)

But we don't want to start from scratch again, right? Once more, we'll use something that programmers released on the Internet for free. Reinventing the wheel is no fun, you know.

Let's use Bootstrap!

Bootstrap is one of the most popular HTML and CSS frameworks for developing beautiful websites: https://getbootstrap.com/

It was written by programmers who worked for Twitter. Now it's developed by volunteers from all over the world!

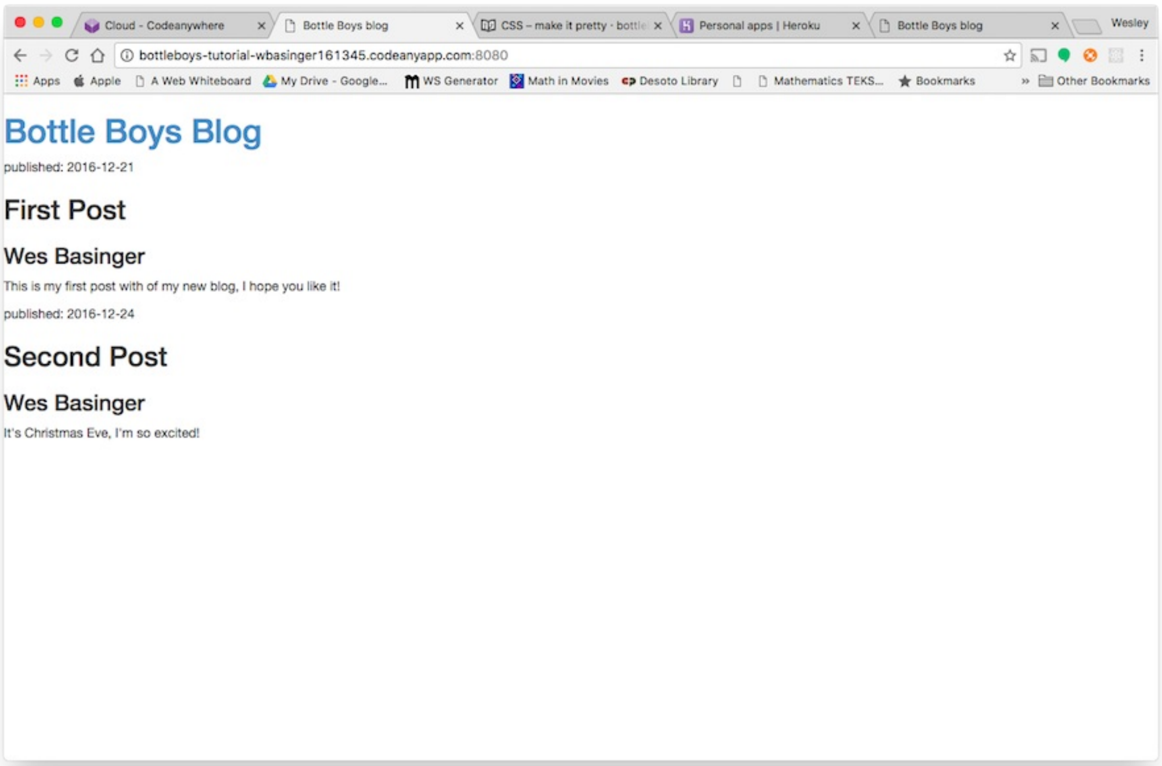
Install Bootstrap

To install Bootstrap, you need to add this to your `<head>` in your `.html` file:

views/index.html

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.
css">
```

This doesn't add any files to your project. It just points to files that exist on the Internet. Just go ahead, open your website and refresh the page. Here it is!



Bottle Boys Blog

published: 2016-12-21

First Post

Wes Basinger

This is my first post with of my new blog, I hope you like it!

published: 2016-12-24

Second Post

Wes Basinger

It's Christmas Eve, I'm so excited!

Looking nicer already! But things are a little too tight to the left, let's fix that with a CSS style sheet of our own.

Static files in Bottle

Finally we will take a closer look at these things we've been calling **static files**. Static files are all your CSS and images. Their content doesn't depend on the request context and will be the same for every user.

Where to put static files for Bottle

Basically, we need to set up a special folder to serve static files from.

We do that by creating a folder called `static` inside the blog app:

```
bottleboys
└───/views
└───app.py
└───Procfile
└───db.json
└───/myvenv
└───requirements.txt
└───/static
```

We need to make another route in `app.py` to serve files from the `static` directory.

app.py

```
from sys import argv
from bottle import route, run, template, static_file # This line is new!!!
from tinydb import TinyDB, Query
import os # This line is new!!!

@route('/')
def index():
  db = TinyDB("db.json")
  posts = db.table("posts")
  return template('index.html', posts=list(posts.all()))

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  return "This is blog number " + str(post_number)

################################
#  This route is new           #
################################
@route('/static/<filename>')
def server_static(filename):
    cwd = os.getcwd()
    return static_file(filename, root=cwd + '/static')


run(host="0.0.0.0", port=argv[1], debug=True)
```

Your first CSS file!

Let's create a CSS file now, to add your own style to your web page. Create a new directory called `css` inside your `static` directory. Then create a new file called `blog.css` inside this `css` directory. Ready?

```
bottleboys
└──/views
└──app.py
└──Procfile
└──db.json
└──/myvenv
└──requirements.txt
└──/static
      └── blog.css
```

Time to write some CSS! Open up the `static/blog.css` file in your code editor.

We won't be going too deep into customizing and learning about CSS here. It's pretty easy and you can learn it on your own after this workshop. There is a recommendation for a free course to learn more at the end of this page.

But let's do at least a little. Maybe we could change the padding on the left side? To understand spacing, computers use pixel units.

In your `static/blog.css` file you should add the following code:

static/blog.css

```css
body {
    padding-left: 15px;
}
```

`body` is a CSS Selector. This means we're applying our styles to the `body` element. So this rule will apply to anything within the body tags and will override any Bootstrap stylings from above.

In a CSS file we determine styles for elements in the HTML file. The first way we identify elements is with the element name. You might remember these as tags from the HTML section. Things like `a` , `h1` , and `body` are all examples of element names. We also identify elements by the attribute `class` or the attribute `id` . Class and id are names you give the element by yourself. Classes define groups of elements, and ids point to specific elements. For example, you could identify the following tag by using the tag name `a` , the class `external_link` , or the id `link_to_wiki_page` :

```html
<a href="https://en.wikipedia.org/wiki/Bottle_(web_framework)" class="external_link" id="link_t
o_wiki_page">
```

You can read more about CSS Selectors at w3schools.

Between the `<head>` and `</head>` tags, after the links to the Bootstrap CSS files, add this line to `index.html` :

views/index.html

```html
<link rel="stylesheet" href="/static/blog.css">
```

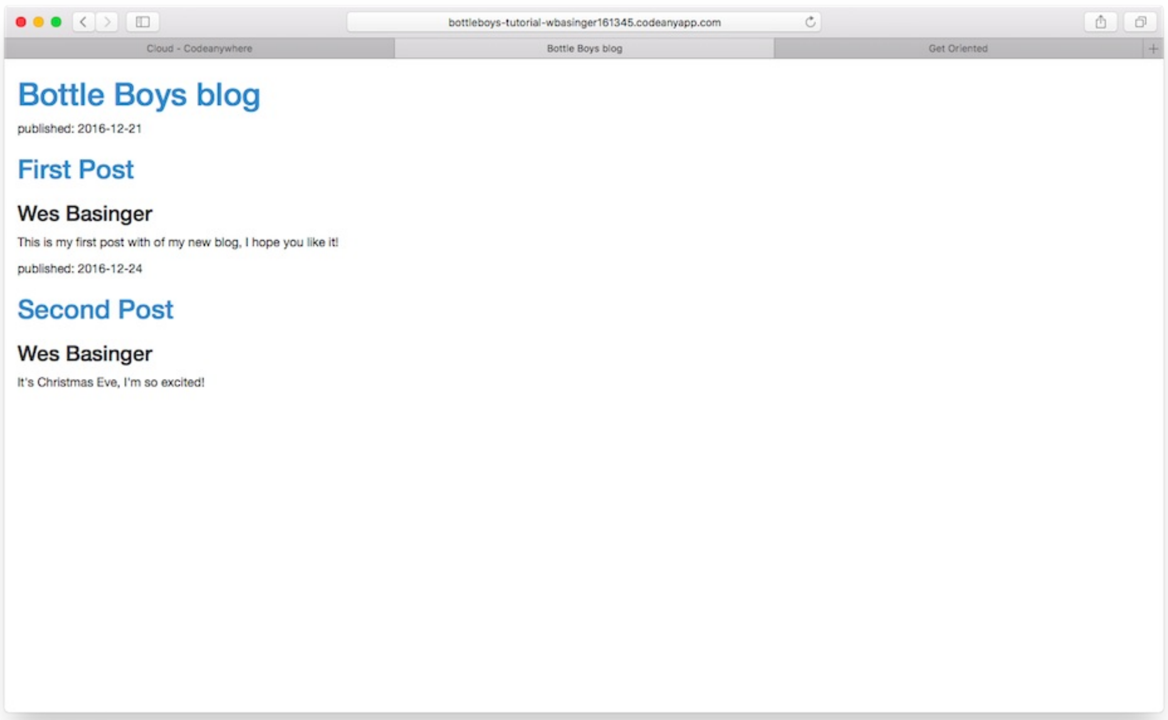
The browser reads the files in the order they're given, so we need to make sure this is in the right place. Otherwise the code in our file may override code in Bootstrap files. We just told our template where our CSS file is located.

Your file should now look like this:

views/index.html

```
<html>
    <head>
        <title>Bottle Boys blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="/static/blog.css">
    </head>
    <body>
        <div>
            <h1><a href="/">Bottle Boys blog</a></h1>
        </div>

        % for post in posts:
          <div>
            <p>published: {{post["date"]}}</p>
            <h2>{{post["title"]}}</h2>
            <h3>{{post["author"]}}</h3>
            <p>{{post["body"]}}</p>
          </div>
        % end
    </body>
</html>
```

OK, save the file and refresh the site!



Bottle Boys blog

published: 2016-12-21

First Post

Wes Basinger

This is my first post with of my new blog, I hope you like it!

published: 2016-12-24

Second Post

Wes Basinger

It's Christmas Eve, I'm so excited!

Nice work! Do you notice the increased spacing on the left hand side? Maybe we can customize the font in our header? Paste this into your `<head>` in `views/index.html` file:

views/index.html

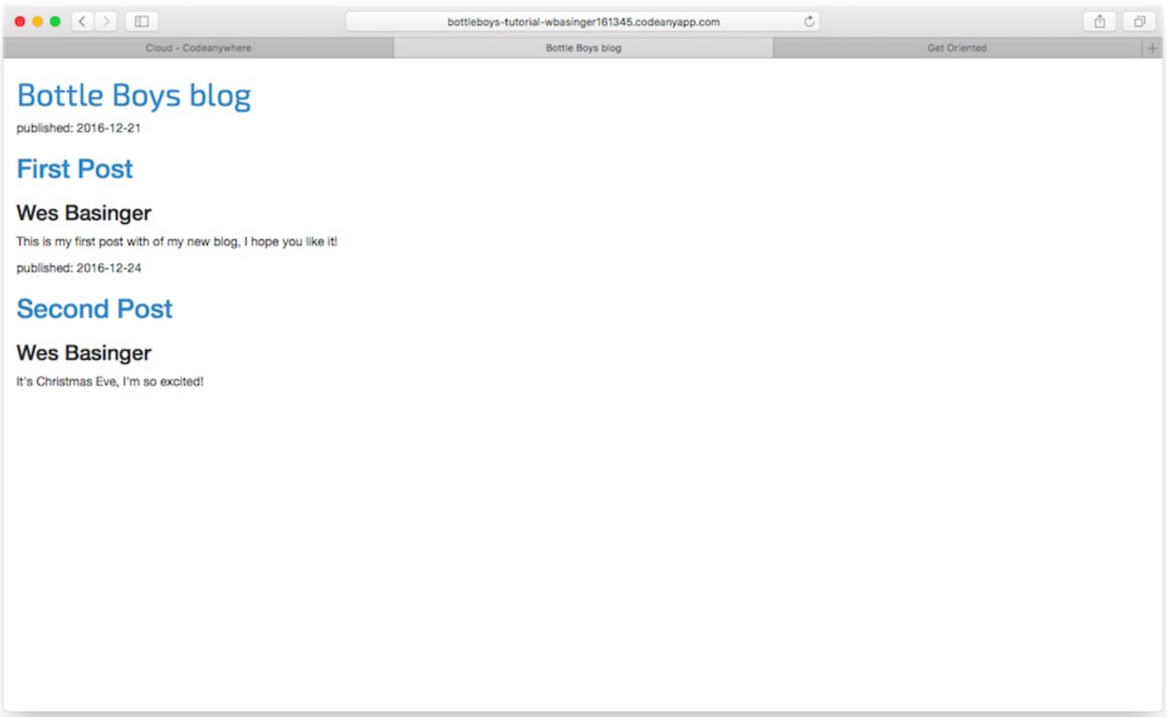
```
<link href="https://fonts.googleapis.com/css?family=Exo+2" rel="stylesheet">
```

As before, check the order and place before the link to `static/blog.css`. This line will import a font called *Exo 2* from Google Fonts (https://www.google.com/fonts).

Find the `h1 a` declaration block (the code between braces `{` and `}`) in the CSS file `views/blog.css`. Now add the line `font-family: 'Exo 2';` between the braces (before or after the body declaration), and refresh the page:

views/blog.css

```
h1 a {
    font-family: 'Exo 2';
}
```



Bottle Boys blog

published: 2016-12-21

First Post

Wes Basinger

This is my first post with of my new blog, I hope you like it!

published: 2016-12-24

Second Post

Wes Basinger

It's Christmas Eve, I'm so excited!

Hopefully you can see the slight change in the font family heading.

As mentioned above, CSS has a concept of classes. These allow you to name a part of the HTML code and apply styles only to this part, without affecting other parts. This can be super helpful! Maybe you have two divs that are doing something different (like your header and your post). A class can help you make them look different.

Go ahead and name some parts of the HTML code. Add a class called `page-header` to your `div` that contains your header, like this:

views/index.html

```
<div class="page-header">
    <h1><a href="/">Bottle Boys Blog</a></h1>
</div>
```

And now add a class `post` to your `div` containing a blog post.

views/index.html

```
<div class="post">
  <p>published: {{post["date"]}}</p>
  <h2><a href="">{{post["title"]}}</a></h2>
  <h3>{{post["author"]}}</h3>
  <p>{{post["body"]}}</p>
</div>
```

Then surround the HTML code which displays the posts with declarations of classes. Replace this:

views/index.html

```
% for post in posts:
  <div class="post">
    <p>published: {{post["date"]}}</p>
    <h2><a href="">{{post["title"]}}</a></h2>
    <h3>{{post["author"]}}</h3>
    <p>{{post["body"]}}</p>
  </div>
% end
```

in the `view/index.html` with this:

views/index.html

```
<div class="content container">
  <div class="row">
      <div class="col-md-8">
        % for post in posts:
          <div class="post">
            <p>published: {{post["date"]}}</p>
            <h2><a href="">{{post["title"]}}</a></h2>
            <h3>{{post["author"]}}</h3>
            <p>{{post["body"]}}</p>
          </div>
        % end
      </div>
  </div>
</div>
```

If you got lost, here's how the entire `index.html` should look.

views/index.html

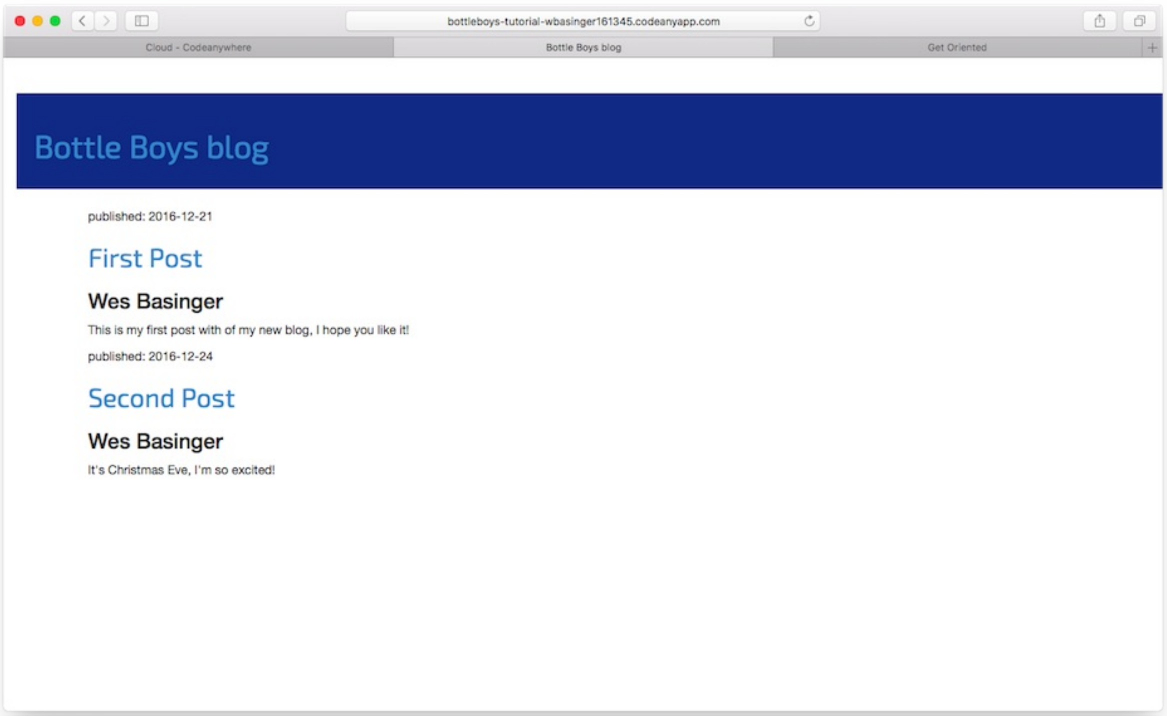
```
<html>
    <head>
        <title>Bottle Boys blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.mi
n.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-th
eme.min.css">
        <link href="https://fonts.googleapis.com/css?family=Exo+2" rel="stylesheet">
        <link rel="stylesheet" href="/static/blog.css">
    </head>
    <body>
        <div class="page-header">
            <h1><a href="/">Bottle Boys blog</a></h1>
        </div>
      <div class="content container">
        <div class="row">
            <div class="col-md-8">
              % for post in posts:
                <div class="post">
                  <p>published: {{post["date"]}}</p>
                  <h2><a href="">{{post["title"]}}</a></h2>
                  <h3>{{post["author"]}}</h3>
                  <p>{{post["body"]}}</p>
                </div>
              % end
            </div>
        </div>
      </div>
    </body>
</html>
```

Save that file and let's add some more custom styling with `blog.css`. When finished, it should contain the following style rules.

static/blog.css

```
body {
    padding-left: 15px;
}
h1 a {
    font-family: 'Exo 2';
}
.page-header {
  background-color: #033482;
  color: white;
  padding: 20px;
}

.post h2 {
  font-family: 'Exo 2';
}
.content-container {
  background-color: #949fb2;
}
```

Save all files, restart the server and reload the page.



Bottle Boys blog

published: 2016-12-21

First Post

Wes Basinger

This is my first post with of my new blog, I hope you like it!

published: 2016-12-24

Second Post

Wes Basinger

It's Christmas Eve, I'm so excited!

Woohoo! Looks awesome, right? Look at the code we just pasted to find the places where we added classes in the HTML and used them in the CSS. Where would you make the change if you wanted the date to be turquoise?

Don't be afraid to tinker with this CSS a little bit and try to change some things. Playing with the CSS can help you understand what the different things are doing. If you break something, don't worry – you can always undo it!

We really recommend taking this free online Codeacademy HTML & CSS course. It can help you learn all about making your websites prettier with CSS.

Ready for the next chapter?! :)

Template including

Another nice thing Bottle has for you is **template including**. What does this mean? It means that you can use the same parts of your HTML for different pages of your website.

Templates help when you want to use the same information or layout in more than one place. You don't have to repeat yourself in every file. And if you want to change something, you don't have to do it in every template, just one!

Create a header template

A header and footer template will let you include a header and a footer on every page of your website.

Let's create a `header.html` and a `footer.html` file in `views` :

```
views
└───index.html
└───header.html
└───footer.html
```

Then open it up and copy everything from `index.html` to `header.html` file, like this:

header.html

```
<html>
    <head>
        <title>Bottle Boys blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.mi
n.css">
         <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-t
heme.min.css">
         <link href="https://fonts.googleapis.com/css?family=Exo+2" rel="stylesheet">
         <link rel="stylesheet" href="/static/blog.css">
    </head>
    <body>
        <div class="page-header">
            <h1><a href="/">Bottle Boys Blog</a></h1>
        </div>
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                % for post in posts:
                  <div class="post">
                    <p>published: {{post["date"]}}</p>
                    <h2><a href="">{{post["title"]}}</a></h2>
                    <h3>{{post["author"]}}</h3>
                    <p>{{post["body"]}}</p>
                  </div>
                % end
              </div>
          </div>
      </div>
    </body>
</html>
```

Do the same thing with `footer.html`. You should now have three identical files. Wait, what? What's going on? You're about to see.

Then in `header.html`, delete everything from `<div class="row">` down. It should end up looking like this:

views/header.html

```
<html>
    <head>
        <title>Bottle Boys blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.mi
n.css">
         <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-t
heme.min.css">
         <link href="https://fonts.googleapis.com/css?family=Exo+2" rel="stylesheet">
         <link rel="stylesheet" href="/static/blog.css">
    </head>
    <body>
        <div class="page-header">
            <h1><a href="/">Bottle Boys Blog</a></h1>
        </div>
```

Next, we'll doctor up the index page. Notice how much less code it takes to define the way the body of the page will look.

views/index.html

```
% include("header.html")
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                % for post in posts:
                  <div class="post">
                    <p>published: {{post["date"]}}</p>
                    <h2><a href="">{{post["title"]}}</a></h2>
                    <h3>{{post["author"]}}</h3>
                    <p>{{post["body"]}}</p>
                  </div>
                % end
              </div>
          </div>
      </div>
% include('footer.html')
```

Last, the lowly footer. There's hardly any code in this file.

views/footer.html

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

That's it! The include functions will piece all the parts of our website together for us. Now, when we start making new pages, we only have to change the middle part of the page, the body, or the div with class `content container`. Check if your website is still working properly. :)

Extend your application

We've already completed all the different steps necessary for the creation of our website: we know how to write data, routes, views and templates. We also know how to make our website pretty.

Time to practice!

The first thing we need in our blog is, obviously, a page to display one post, right?

Create a template link to a post's detail

We will start with adding a link inside `views/index.html` file. So far it should look like this:

views/index.html

```
% include("header.html")
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                % for post in posts:
                  <div class="post">
                    <p>published: {{post["date"]}}</p>
                    <h2><a href="">{{post["title"]}}</a></h2>
                    <h3>{{post["author"]}}</h3>
                    <p>{{post["body"]}}</p>
                  </div>
                % end
              </div>
          </div>
      </div>
% include('footer.html')
```

We want to have a link from a post's title in the post list to the post's detail page. Let's change `<h2><a href=""> {{post["title"]}}</a></h2>` so that it links to the post's detail page:

views/index.html

```
<h2><a href="{{"/blog/" + post["postId"]}}">{{post["title"]}}</a></h2>
```

If you fire up your server and check out the page now, you'll have noticed that each of the links now becomes clickable. And... when you click on the link, you are taken to a very plain and boring page that we created a long time ago, earlier in the tutorial. Good news, our route still works, now let's go back to `app.py` and dress it up a bit. Let's just focus on the blog route, everything else can stay the same. Currently it looks like this:

app.py

```
@route('/blog/<post_number>')
def blog(post_number):
  return "This is blog number " + str(post_number)
```

Let's change it up just a bit.

app.py

```
@route('/blog/<post_number>')
def blog(post_number):
  db = TinyDB('db.json')
  posts = db.table('posts')
  Post = Query()
  blog = posts.get(Post.postId == post_number)
  return template('blog.html', blog=blog)
```

Create a new template to handle the blog view

Make a new file called `blog.html` in `views` .

views/blog.html

```
% include("header.html")
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                <h1>{{blog["title"]}}</h1>
                <h2>{{blog["date"]}}</h2>
                <br>
                <p>{{blog["body"]}}</p>
              </div>
          </div>
      </div>
% include('footer.html')
```

Nice! What a good clean look at each post. Let's go back to the index route and only give the reader a preview of the blog body, so that they'll be forced to click the link to read on. *Insert evil laugh.*

views/index.html

```
% include("header.html")
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                % for post in posts:
                  <div class="post">
                    <p>published: {{post["date"]}}</p>
                    <h2><a href="{{"/blog/" + post["postId"]}}">{{post["title"]}}</a></h2>
                    <h3>{{post["author"]}}</h3>
                    <p>{{post["body"][0:20] + ' ...'}}</p>
                  </div>
                % end
              </div>
          </div>
      </div>
% include('footer.html')
```

Go ahead, check out your site and see if it's still working. It should be!

One more thing: deploy time!

It'd be good to see if your website will still be working on Heroku, right? Let's try deploying again.

command-line

```
$ git status
$ git add --all .
$ git status
$ git commit -m "Added view and template for detailed blog post as well as CSS for the site."
$ git push
```

Then, in a console:

command-line

```
$ git push heroku master
[...]
```

And that should be it! Congrats :)

Bottle Forms

The final thing we want to do on our website is create a nice way to add and edit blog posts. So far, we can only view existing posts. If we wanted new entries, we would have to edit the JSON file, and that's pretty cumbersome. With forms we will have absolute power over our interface – we can do almost anything we can imagine!

We'll have to build this all from scratch, and it's going to take several new routes and learning some new HTTP methods. Let's start with the easiest part, let's build a template for the HTML form. Start a new file called `new.html` in `views` .

views/new.html

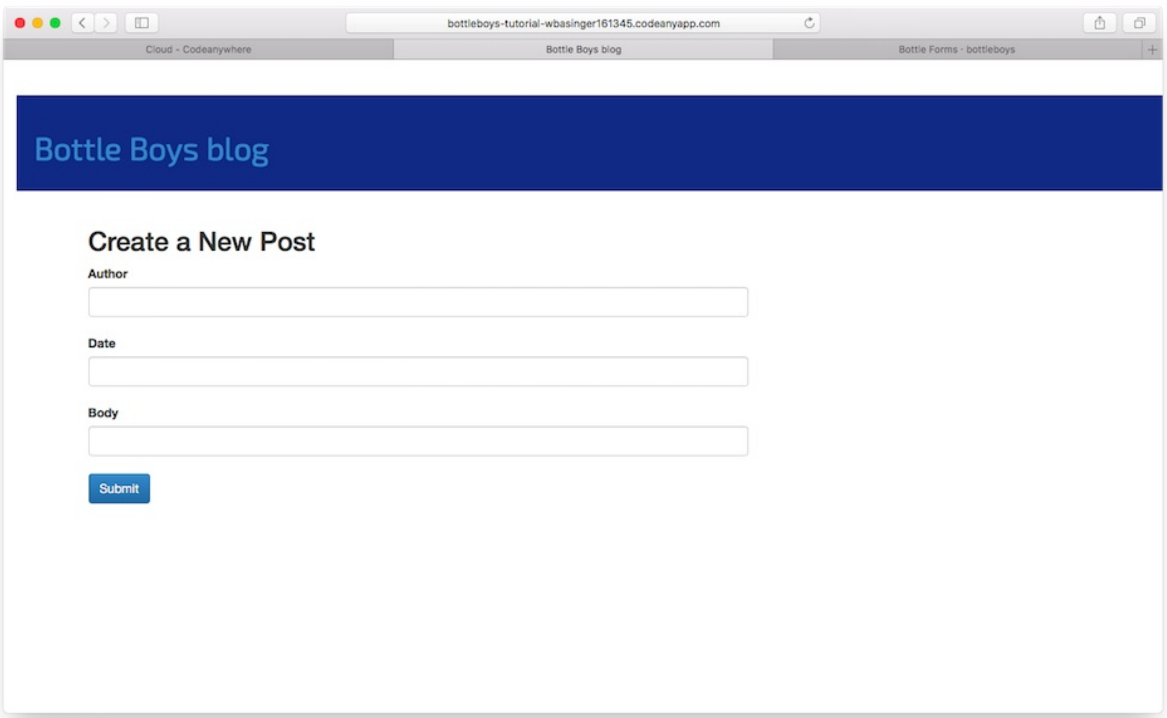
```
% include("header.html")
        <div class="content container">
          <div class="row">
              <div class="col-md-8">
                <h2>Create a New Post</h2>
                <div class="form-group">
                <form action='/new' method="POST">
                  <label for="author">Author</label>
                  <input type="text" required="yes" name="author" class="form-control"></input><
br>
                  <label for="title">Title</label>
                  <input type="text" required="yes" name="title" class="form-control"></input><
br>
                  <label for="date">Date</label>
                  <input type="date" required="yes" name="date" class="form-control"></input><br
>
                  <label for="body">Body</label>
                  <input type="textarea" required="yes" name="body" class="form-control"></input
><br>
                  <button type="submit" class="btn btn-primary">Submit</button>
                </form>
                </div>
              </div>
          </div>
      </div>
% include('footer.html')
```

We won't be able to see our nice new form until we make a route for it. So let's go to `app.py` and fix that. Note the new methods that we are importing from `bottle` at the top of the file.

app.py

```
from sys import argv
from bottle import route, run, template, static_file, get, post # This line is new!!!
from tinydb import TinyDB, Query
import os

@route('/')
def index():
  db = TinyDB("db.json")
  posts = db.table("posts")
  return template('index.html', posts=list(posts.all()))

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  db = TinyDB('db.json')
  posts = db.table('posts')
  Post = Query()
  blog = posts.get(Post.postId == post_number)
  return template('blog.html', blog=blog)

@get('/new')
def create_new():
  return template('new.html')

@route('/static/<filename>')
def server_static(filename):
    cwd = os.getcwd()
    return static_file(filename, root=cwd + "/static")

run(host="0.0.0.0", port=argv[1], debug=True)
```

If you run the server and visit http://0.0.0.0:8080/new, you can see the new shiny form you built.



But if you try to fill it out and submit, you'll get a big fat error. That's because the action is a POST route and we haven't defined that in `app.py` yet. That's our next step. Open up `app.py` and let's modify it again. This time we'll add a post route.

app.py

```
from sys import argv
from bottle import route, run, template, static_file, get, post, request, redirect # This line
is new!!!
from tinydb import TinyDB, Query
import os

@route('/')
def index():
  db = TinyDB("db.json")
  posts = db.table("posts")
  return template('index.html', posts=list(posts.all()))

@route('/about')
def about():
  return "This is the about me route."

@route('/blog/<post_number>')
def blog(post_number):
  db = TinyDB('db.json')
  posts = db.table('posts')
  Post = Query()
  blog = posts.get(Post.postId == post_number)
  return template('blog.html', blog=blog)

@get('/new')
def create_new():
  return template('new.html')

@post('/new')
def post_new():
  author = request.forms.get('author')
  date = request.forms.get('date')
  body = request.forms.get('body')
  title = request.forms.get('title')
  db = TinyDB('db.json')
  posts = db.table('posts')
  new_post_id = posts.insert(
      {
        'author': author,
        'date': date,
        'body': body,
        'postId': "",
        'title' : title})
  posts.update({'postId': str(new_post_id)}, eids=[new_post_id])
  redirect('/')


@route('/static/<filename>')
def server_static(filename):
    cwd = os.getcwd()
    return static_file(filename, root=cwd + "/static")

run(host="0.0.0.0", port=argv[1], debug=True)
```

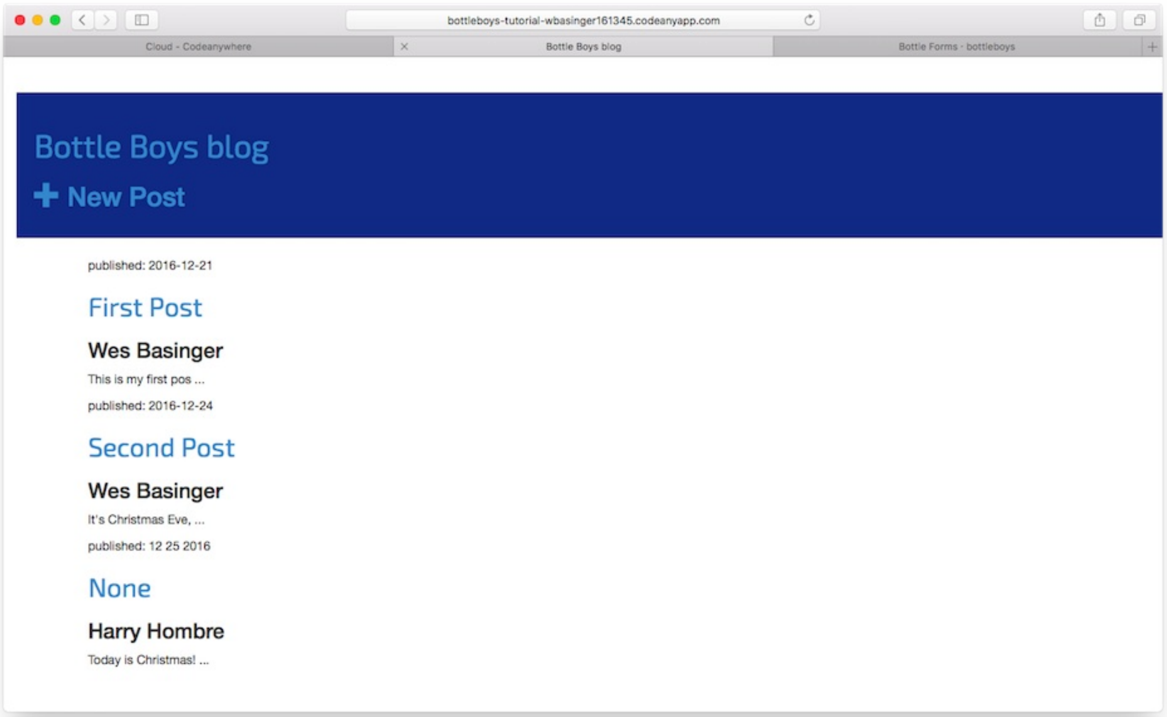
Now restart the server, go back and visit http://0.0.0.0:8080/new. You should be able to write new blog posts with ease!

But instead of having to type in the `/new` part on the end of the address, let's modify the header file to include a new post link. Open up `header.html`.

views/header.html

```
<html>
    <head>
        <title>Bottle Boys Blog</title>
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.mi
n.css">
         <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-t
heme.min.css">
         <link href="https://fonts.googleapis.com/css?family=Exo+2" rel="stylesheet">
         <link rel="stylesheet" href="/static/blog.css">
    </head>
    <body>
        <div class="page-header">
            <h1><a href="/">Bottle Boys Blog</a></h1>
            <a href="/new"><h2><span class="glyphicon glyphicon-plus"></span>New Post</h2></a>
        </div>
```

This is what I got when I finished...


bottleboys-tutorial-wbasinger161345.codeanyapp.com
Cloud - Codeanywhere
Bottle Boys blog
Bottle Forms - bottleboys

Bottle Boys blog

+ New Post

published: 2016-12-21

First Post

Wes Basinger

This is my first pos ...

published: 2016-12-24

Second Post

Wes Basinger

It's Christmas Eve, ...

published: 12 25 2016

None

Harry Hombre

Today is Christmas! ...

One more thing: deploy time!

Let's see if all this works on Heroku. Time for another deploy!

- First, commit your new code, and push it up to Github:

command-line

```
$ git status
$ git add --all .
$ git status
$ git commit -m "Added views to create/edit blog post inside the site."
$ git push
```

- Then:

command-line

```
$ git push heroku master
[...]
```

And that should be it! Congrats :)

What's next?

Congratulate yourself! **You're totally awesome**. We're proud! <3

What to do now?

Take a break and relax. You have just done something really huge.

After that, make sure to follow me on Facebook or Twitter to stay up to date.

Can you recommend any further resources?

Yes!

Try the resources listed below. They're all very recommended!

- New Coder tutorials
- Code Academy Python course
- Code Academy HTML & CSS course
- Learn Python The Hard Way book
- Hello Web App: Learn How to Build a Web App