

# Setting up your Mac OS X

All the needed dependencies for setting up your machine for these tutorials.

---

## — Overview of requirements —

---

The installation will depend on your operating system, but overall, you will need:

- Python 2.x - there are plans to update/include Python 3.x
- git
- A C compiler
- pip
- virtualenv
- virtualenvwrapper
- text editor of your choice

---

## — Installation —

---

### Python

Macs come with Python pre-installed. To check, open up the Terminal application, then type `python`:

```
$ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Appletclang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python.org<sup>[1]</sup> has a good Python on the Mac<sup>[2]</sup> page if the above does not work for you.

### git

You will need to install git[3] on your machine through their download page[4]. You can then follow the Save your Progress[5] page to set it up.

## C compiler

To test if you have either GCC or clang, type `$ gcc` or `$ clang` into your terminal. If you get an error that says “command not found” then follow the install instructions:

You will need the XCode[6] application. Once you have XCode on your machine, you will need to navigate to Preferences → Downloads, then select **Command Line Tools** to download & install (this may take a while, get some coffee, go take a shower).

This gives you the GCC[7] or the GNU Compiler Collection. To test installation, within the Terminal application, type `gcc` and you should get the following:

```
$ gcc
i686-apple-darwin11-llvm-gcc-4.2: no input files
```

## pip

pip[8], stands for “python install python”, is a tool for installing and managing Python packages. Within your Terminal application, use the following commands (ignore the leading `$` as that is your terminal prompt) for downloading & installing. It may prompt you for your computer login password.

```
$ sudo curl -O http://python-distribute.org/distribute_setup.py | python
$ sudo curl -O https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py | python
$ pip
Usage: pip COMMAND [OPTIONS]
You must give a command (use "pip help" to see a list of commands)
```

## virtualenv & virtualenvwrapper

virtualenv[9] creates isolated environments for each of your Python projects. It helps to solve version & dependency problems with multiple Python installations and/or multiple versions of different Python packages. We’ll use `pip` to install it:

```
$ sudo pip install virtualenv
```

virtualenvwrapper[10] is a great (but not required) tool for using virtualenv by simplifying the commands that virtualenv needs. We’ll use `pip` again to install it:

```
$ sudo pip install virtualenvwrapper
$ export WORKON_HOME=~/.Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
```

---

## — Text Editor —

---

If you already have a text editor that you like to use, great!

If not, I would suggest grabbing Sublime Text 2<sup>[11]</sup>. It's free, and very user-friendly, especially for the beginner.

Now continue on to “Test your setup”.

# Setting up your Linux machine

All the needed dependencies for setting up your machine for these tutorials.

---

## — Overview of requirements —

---

The installation will depend on your operating system, but overall, you will need:

- Python 2.x - there are plans to update/include Python 3.x
- git
- A C compiler
- pip
- virtualenv
- virtualenvwrapper

---

## — Installation —

---

### Python

Linux machines come with Python pre-installed. To check, open up the Terminal application, then type `python`:

```
$ python
Python 2.7.3 (default, Aug  9 2012, 17:23:57)
[GCC 4.7.1 20120720 (Red Hat 4.7.1-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

### git

You will need to install git<sup>[12]</sup> either from commands below or through their download page<sup>[13]</sup>. You can then follow the Save your Progress<sup>[14]</sup> page to set it up.

- Fedora: `sudo yum git`
- Ubuntu: `sudo apt-get install git`

## C Compiler

A C compiler, either GCC or clang, is needed because the `numpy` library we are using has some C extensions, which will need to be compiled.

To test if you have either GCC or clang, type `$ gcc` or `$ clang` into your terminal. If you get an error that says “command not found” then follow the install instructions:

- Fedora:
  - `sudo yum groupinstall "Developer Tools"`
  - `sudo yum install python-devel`
- Ubuntu:
  - you may need to run `sudo apt-get update` first.
  - `sudo apt-get install build-essential python-dev`

This gives you the GCC<sup>[15]</sup> or the GNU Compiler Collection. To test installation, within the Terminal application, type `gcc` and you should get something like the following:

```
$ gcc
gcc: fatal error: no input files
compilation terminated.
```

## pip

`pip`<sup>[16]</sup>, stands for “python install python”, is a tool for installing and managing Python packages. Within your Terminal application, use the following commands (ignore the leading `$` as that is your terminal prompt) for downloading & installing. It may prompt you for your computer login password.

```
$ sudo curl -O http://python-distribute.org/distribute_setup.py | python
$ sudo curl -O https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py | python
$ pip
Usage: pip COMMAND [OPTIONS]
You must give a command (use "pip help" to see a list of commands)
```

## virtualenv & virtualenvwrapper

`virtualenv`<sup>[17]</sup> creates isolated environments for each of your Python projects. It helps to solve version & dependency problems with multiple Python installations and/or multiple versions of different Python packages. We’ll use `pip` to install it:

```
$ sudo pip install virtualenv
```

virtualenvwrapper<sup>[18]</sup> is a great (but not required) tool for using virtualenv by simplifying the commands that virtualenv needs. We'll use `pip` again to install it:

```
$ sudo pip install virtualenvwrapper
$ export WORKON_HOME=~/.Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
```

---

## — Text Editor —

---

If you already have a text editor that you like to use, great!

If not, I would suggest grabbing Sublime Text 2<sup>[19]</sup>. It's free, and very user-friendly, especially for the beginner.

Now continue on to “Test your setup”.

Now continue on to “Test your setup”.

# Setting up your Windows machine

All the needed dependencies for setting up your machine for these tutorials.

---

## — Overview of requirements —

---

The installation will depend on your operating system, but overall, you will need:

- Python 2.x - there are plans to update/include Python 3.x
- git
- A C compiler
- pip
- virtualenv
- virtualenvwrapper

---

## — Installation —

---

### Python

1. Go here: `http://python.org/ftp/python/2.7.1/python-2.7.1.msi` [20] and click “run” if given the option. Otherwise, save it to your Desktop, then minimize windows to see your desktop, and double click on it to start the installer. Follow the installer instructions to completion.
2. Open a command prompt (we will be doing this multiple times, so make a note of how to do this!):
  - On Windows Vista or Windows 7: click on the Start menu (the Windows logo in the lower left of the screen), type cmd into the Search field directly above the Start menu button, and click on “cmd” in the search results above the Search field.
  - On Windows XP: click on the Start menu (the Windows logo in the lower left of the screen), click on “Run...”, type cmd into the text box, and hit enter.
3. At this `C:\` prompt that appears, test your Python install by typing `\Python27\python.exe` and hitting `enter`. You should see something like

Python 2.7.3 (r271:86832,...) on win32

Type "help", "copyright", "credits" or "license" for more information

>>>

1. You just started Python! The `>>>` indicates that you are at a new type of prompt - a Python prompt. The command prompt lets you navigate your computer and run programs, and the Python prompt lets you write and run Python code interactively.
2. To exit the Python prompt, type `exit()` and press Enter. This will take you back to the Windows command prompt (the `C:\` you saw earlier).
3. Put Python on the PATH - You might have noticed that you typed a "full path" to the Python application above when launching Python (`python.exe` is the application, but we typed `\Python27\python.exe`). In this step, you will configure your computer so that you can run Python without typing the "Python27" directory name.
  - Get to System Properties
    1. Open up "My Computer" by clicking on the Start menu or the Windows logo in the lower-left hand corner, and navigate to "My Computer" (for Windows XP) or "Computer" (For Vista and Windows 7).
    2. Right-click on the empty space in the window, and choose "Properties".
      - If you're using XP: window labeled "System Properties" will pop up. Click the "Advanced" tab. A window with the title "System Properties" will appear.
      - If you're **not** using XP: A window labeled "View basic information about your computer" will appear. In this window, click "Advanced system settings". A window with the title "System Properties" will appear.
  - Edit the Path
    1. Within System Properties, make sure you are in the tab labeled "Advanced".
    2. Click the button labeled "Environment Variables". A window labeled "Environment Variables" will appear.
    3. In this window, the screen is split between "User variables" and "System variables". Within "System variables", scroll down and find the one labeled "Path". Click the "Edit..." button. A window with the "Variable name" and the "Variable value" should appear. The "Variable value" will already have some text in it; click in the box to unhighlight it (we don't want to accidentally delete that text).
  - In the "Variable value" box, scroll to the end. Add the following text, and hit OK. Make sure to include the semicolon at the start! `;c:\python27;c:\python27\scripts;c:\python27\tools\scripts`
  - Hit "OK" to close out the system properties window.
  - Test your change:
    1. Open up a new command prompt: you do this the same way you did above when installing python. This needs to be a new command prompt because the changes you just made didn't take affect in prompts that were already open.
    2. Type `python` into the command prompt to start Python
    3. Notice that you now get a Python interpreter, indicated by the change to a `>>>` prompt.



4. Exit the Python prompt by typing `exit()` and hitting enter. Now you're back at the Windows command prompt (`C:\`).

Success! You have installed Python!

## git

Download git through git's website: <http://git-scm.com/download/win> [21].

## C Compiler

Download the MinGW GCC compiler here:

<http://sourceforge.net/projects/mingw/files/latest/download?source=files> [22] and follow installation instructions here: <http://www.mingw.org/wiki/InstallationHOWTOforMinGW> [23]

## virtualenv + pip

1. From here, <http://pypi.python.org/pypi/virtualenv> [24], under Installation, see "You can just grab the single file virtualenv.py ..". Right-click and save-as and place in the Scripts folder that we earlier added to system PATH.
2. From the command line, `cd` into the directory that you want to save your Python work. **IMPORTANT TIP:** Do not create or use a directory that has spaces in the name. For instance, 'Python Projects' is not good, but 'PythonProjects' is fine.
3. Within your Python project directory, type: `virtualenv.py dataviz`. A virtual environment called "dataviz" has been created, and now lives in your Python projects directory. This also installs pip: pip is a tool that is used to install Python packages
4. You will now need to activate the virtual environment by typing in your command line: `dataviz\Scripts\activate.bat`. You should see something like this: `(dataviz) C:\PythonProjects>`
5. To stop working in the virtual environment, type the following in your command line: `dataviz\Scripts\deactivate.bat`. You will need to run the activate script again (stated in #4) to start up the virtual environment.

---

## — Text Editor —

---

If you already have a text editor that you like to use, great!

If not, I would suggest grabbing Sublime Text 2 [25]. It's free, and very user-friendly, especially for the beginner.

Now continue on to "Test your setup".

# Test your setup

Now let's test our installation and get familiar with creating & using virtual environments:

- Mac OS X/Linux:

```
$ mkvirtualenv TestEnv
Installing
distribute.....
.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/postactivate
virtualenvwrapper.user_scripts creating /Users/lynnroot/Envs/TestEnv/bin/get_env_details
```

- Windows:

```
# Within your ProjectFolder
C:\dataviz\Scripts> virtualenv.py TestEnv
Installing
distribute.....
.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/lynnroot/Envs/TestEnv/bin/postactivate
virtualenvwrapper.user_scripts creating /Users/lynnroot/Envs/TestEnv/bin/get_env_details
```

Now that you made a virtual environment called `TestEnv`, you should see `(TestEnv)` before your prompt:

```
(TestEnv) $
```

Let's play around with commands for virtualenv:

- Mac OS X/Linux:

```
# deactivate the TestEnv
(TestEnv) $ deactivate
$
# reactivate the TestEnv
$ workon TestEnv
(TestEnv) $
# install the Django package in your TestEnv environment
(TestEnv) $ pip install django
Downloading/unpacking django
  Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
  Running setup.py egg_info for package django
Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.6/django-admin.py from 644 to 755
    changing mode of /Users/lynnroot/Envs/TestEnv/bin/django-admin.py to 755
Successfully installed django
(TestEnv) $
```

- Windows:

```
# deactivate the TestEv
(TestEnv) dataviz\Scripts> deactivate.bat
C:\dataviz\Scripts>
C:\dataviz\Scripts> activate.bat
(TestEnv) C:\dataviz\Scripts>
```

```
# install the Django package in your TestEnv environment
(TestEnv) C:\ pip install django
Downloading/unpacking django
  Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
  Running setup.py egg_info for package django
Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.6/django-admin.py from 644 to 755
    changing mode of /Users/lynnroot/Envs/TestEnv/bin/django-admin.py to 755
Successfully installed django
(TestEnv) C:\dataviz\Scripts>
```

- All operating systems (for Windows, know that instead of the `$` prompt, you will see `C:\` + folder name:

```
# test the installation of Django
(TestEnv) $ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> exit()
# deactivate the TestEnv virtual environment
(TestEnv) $ deactivate
$
```

```
# try to import Django again
$ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named django
>>> exit()
$
```

```
# reactivate the TestEnv virtual environment
$ workon TestEnv
(TestEnv) $
# try again to import Django
(TestEnv) $ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Appletclang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> exit()
(TestEnv) $
```

```
# see what libraries are installed in the TestEnv virtual environment:
(TestEnv) $ pip freeze
django==1.5
(TestEnv) $
```

- Here's a run-down of useful commands for pip, virtualenv & virtualenvwrapper:
  - For Linux + Mac OS:
    - `mkvirtualenv [ENV_NAME]` - creates and activates a fresh virtual environment
    - `workon [ENV_NAME]` - activates an already-created virtual environment
    - `deactivate` - deactivates the virtual environment that is currently active
    - within an activated virtualenv, `pip install [PACKAGE_NAME]` installs a package into the virtualenv
    - within an activated virtualenv, `pip freeze` lists the packages that is installed & accessible within the virtualenv
  - For Windows:
    - `virtualenv.py [ENV_NAME]` - creates and activates a fresh virtual environment within `ProjectFolder`.
    - `ProjectFolder\Scripts\activate.bat` - activates an already-created virtual environment
    - `ProjectFolder\Scripts\deactivate.bat` - deactivates the virtual environment that is currently active
    - within an activated virtualenv, `pip install [PACKAGE_NAME]` installs a package into the virtualenv
    - within an activated virtualenv, `pip freeze` lists the packages that is installed & accessible within the virtualenv

You're good to go with your setup!

# Part 0: Setup

Initial setup for our Data Visualization tutorial.

Within your terminal/command line (note: if using Windows, your prompt will look like `C:\` rather than `$`)

- To get to your “Home” directory:

```
$ cd
```

- To create a new `Projects` folder and move to that directory. You can name it whatever you want, just remember what you named it, and where it is:

```
$ mkdir Projects && cd Projects
```

- Clone the New Coder project into the directory you’re currently in, which is `Projects` (unless you named it something else):

```
$ git clone https://github.com/econchick/new-coder.git
```

- Change into the Data Viz project:

```
$ cd new-coder/dataviz
```

- To make a virtual environment specific to your Data Viz project, run the following command. You should see `(DataVizProject)` before your prompt.

```
# for Linux/Mac OS X
$ mkvirtualenv DataVizProj
(DataVizProj)$
# for Windows
C:\Projects\new-coder\dataviz\virtualenv.py DataVizProj
(DataVizProj) C:\
```

- Now we will install the package requirements for this project. Your virtual environment will store the required packages in a self-contained area to not mess up with other Python projects.

```
(DataVizProject) $ pip install -r requirements.txt
```

- **NOTE** Sometimes, NumPy is finicky. If the previous step returns errors, try:

```
(DataVizProj)$ pip install numpy
(DataVizProj)$ pip install matplotlib
```

- Test the installation real quick by starting up the Python interpreter:

```
(DataVizProj)$ python
>>> import numpy
>>> import matplotlib
```

- If you have no errors (you would just see the `>>>` prompt), then you're good to go. You can close out of the Python interpreter by pressing `CTRL+D`. If you do have errors, I'd try downloading numpy and matplotlib manually from <http://scipy.org/Download> and <http://matplotlib.org/downloads.html>, respectively.

**Continue on to Part 1: Parsing our Data →**

# Part 1: Parsing Data

Walk through of how to parse CSV data with Python using sample crime data from San Francisco.

---

## — Module Setup —

---

Open up `parse.py`, found: [new-coder/dataviz/tutorial\\_source/parse.py](https://github.com/new-coder/dataviz/tutorial_source/parse.py)

The beginning of the module, `new-coder/blob/master/dataviz/tutorial_source/parse.py` lines 1-12, is an introduction as well as any copyright and/or license information.

In order to read a CSV/Excel file, we have to import the `csv` module from Python's standard library.

```
import csv
```

`MY_FILE` is defining a global - notice how it's all caps, a convention for variables we won't be changing. Included in this repo is a sample file to which this variable is assigned.

```
MY_FILE = "../data/sample_sfpd_incident_all.csv"
```

---

## — The Parse Function —

---

In defining the function, we know that we want to give it the CSV file, as well as the delimiter in which the CSV file uses to delimit each element/column.

```
def parse(raw_file, delimiter):
```

We also know that we want to return a JSON-like object. A JSON file/object is just a collection of dictionaries, much like Python's dictionary.



```
def parse(raw_file, delimiter):  
  
    return parsed_data
```

Let's be good coders and write a documentation-string (doc-string) for future folks that may read our code. Notice the triple-quotes:

```
def parse(raw_file, delimiter):  
    """Parses a raw CSV file to a JSON-line object."""  
  
    return parsed_data
```

---

## For the curious

If you are interested in understanding how docstrings work, Python's PEP (Python Enhancement Proposals) documents spell out how one should craft his/her docstrings: PEP8 and PEP257. This also gives you a peek at what is considered "Pythonic".

The difference between `"""docstrings"""` and `# comments` have to do with who the reader will be. Within the a Python shell, if you call `help` on a particular function or class, it will return the `"""docstring"""` that the developer has written.

There are also documentation programs that look specifically for `"""docstrings"""` to help the developer automatically produce documentation separated out of the code. Within docstrings, it's helpful to say imperatively what the function/method or class is supposed to do. Examples of how the documented code should work can also be written in the docstrings (and, subsequently, tested). `# comments`, on the otherhand, are for those reading through the code — the comments are to simply say what a specific piece/line of code is meant to do. Inline `# comments` are always appreciated by those reading through your code. Many developers also litter `# TODO` or `# FIXME` statements for combing through later.

---

What we have now is a pretty good skeleton - we know what parameters the function will take (`raw_file` and `delimiter`), what it is supposed to do (our `"""doc-string"""`), and what it will return, `parsed_data`. Notice how the parameters and the return value is descriptive in itself.

Let's sketch out, with comments, how we want this function to take a raw file and give us the format that we want. First, let's open the file, and the read the file, then build the `parsed_data` element.

```
def parse(raw_file, delimiter):
    """Parses a raw CSV file to a JSON-line object"""

    # Open CSV file

    # Read CSV file

    # Close CSV file

    # Build a data structure to return parsed_data

    return parsed_data
```

Thankfully, there are a lot of built-in methods that Python has that we can use to do all the steps that we've outlined with our comments. The first one we'll use is `open` and pass `raw_file` to it, which we got from defining our own parameters in the `parse` function:

```
opened_file = open(raw_file)
```

```
...
```

So we've told Python to open the file, now we have to read the file. We have to use the CSV module that we imported earlier:

```
csv_data = csv.reader(opened_file, delimiter=delimiter)
```

Here, `csv.reader` is a function of the CSV module. We gave it two parameters: `opened_file`, and `delimiter`. It's easy to get confused when parameters and variables share names. In `delimiter=delimiter`, the first 'delimiter' is referring to the name of the parameter that `csv.reader` needs; the second 'delimiter' refers to the argument that our `parse` function takes in.

Just to quickly put these two lines in our `parse` function:

```
def parse(raw_file, delimiter):  
    """Parses a raw CSV file to a JSON-line object"""  
  
    # Open CSV file  
    opened_file = open(raw_file)  
  
    # Read the CSV data  
    csv_data = csv.reader(opened_file, delimiter=delimiter)  
  
    # Build a data structure to return parsed_data  
  
    # Close the CSV file  
  
    return parsed_data
```

---

## For the curious

The `csv_data` object, in Python terms, is now an iterator. In very simple terms, this means we can get each element in `csv_data` one at a time.

---

Alright – the building of the data structure might seem tricky. The best way to start off is to set up an empty Python list to our `parsed_data` variable so we can add every row of data that we will parse through.

```
parsed_data = []
```

Good – we have a good data structure to add to. Now let's first address our column headers that came with the CSV file. They will be the first row, and we'll assign them to the variable `fields`:

```
fields = csv_data.next()
```

---

## For the curious

We were able to call the `.next` method on `csv_data` because it is a generator. We just call `.next` once, since headers are in the 1st and only row of our CSV file.

---

Let's loop over each row now that we have the headers properly taken care of. With each loop, we will add a dictionary that maps a field (those column headers) to the value in the CSV cell.

```
for row in csv_data:
    parsed_data.append(dict(zip(fields, row)))
```

Here, we iterated over each row in the `csv_data` item. With each loop, we appended a dictionary (`dict()`) to our list, `parsed_data`. We use Python's built-in `zip()` function to zip together header → value to make our dictionary of every row.

Now let's put the function together:

```
def parse(raw_file, delimiter):
    """Parses a raw CSV file to a JSON-like object"""

    # Open CSV file
    opened_file = open(raw_file)

    # Read the CSV data
    csv_data = csv.reader(opened_file, delimiter=delimiter)

    # Setup an empty list
    parsed_data = []

    # Skip over the first line of the file for the headers
    fields = csv_data.next()

    # Iterate over each row of the csv file, zip together field -> value
    for row in csv_data:
        parsed_data.append(dict(zip(fields, row)))

    # Close the CSV file
    opened_file.close()

    return parsed_data
```

---

## — Using the new Parse function —

---

Let's define a `main()` function to act as the starting point for our script, and use our new `parse()` function:

```
def main():  
    # Call our parse function and give it the needed parameters  
    new_data = parse(MY_FILE, ",")  
  
    # Let's see what the data looks like!  
    print new_data
```

We called our function `parse()` and gave it the `MY_FILE` global variable that we defined at the beginning, as well as the delimiter `", "`.

We assign the function to the variable `new_data` since the `parse()` function will return a `parsed_data` object. Last, we print `new_data` to see our list of dictionaries!

One final bit – when running a Python file from the command line, Python will execute all of the code found on it. Since the following bit is `True`,

```
if __name__ == "__main__":  
    main()
```

it will call the `main()` function. By doing the `name == __main__` check, you can have that code only execute when you want to run the module as a program (via the command line) and not have it execute when someone just wants to import the `parse()` function itself into another Python file. This is referred to as “boilerplate code” – code doesn’t really do anything and yet is necessary.

---

## Putting it to action

---

So you’ve written the parse function and your `parse.py` file looks like mine in `new-coder/blob/master/dataviz/tutorial_source/parse.py`. Now what? Let’s run it and parse some d\*mn files!

Be sure to have your virtualenv activated that you created earlier in setup. Your terminal prompt should look something like this:

```
(DataVizProj) $
```

Within the `new-coder/dataviz/` directory, let’s make a directory for the python files you are writing with the bash command `mkdir [Directory_Name]`:

```
(DataVizProj) $ mkdir MySourceFiles
(DataVizProj) $ ls # list available files and directories where we are
README.me requirements.txt data full_source MySourceFiles tutorial_source
(DataVizProj) $ pwd # current location in our directory structure
Users/lynnroot/MyProjects/new-coder/dataviz/
(DataVizProj) $ cd MySourceFiles # change directories into our new directory
```

Go ahead and save your copy of `parse.py` into `MySourceFiles` (through “Save As” within your text editor). You should see the file in the directory if you return to your terminal and type `ls`.

To run the python code, you have to tell the terminal to execute the `parse.py` file with python:

```
(DataVizProj) $ python parse.py
```

If you got a traceback, or an error message, compare your `parse.py` file with `new-coder/dataviz/tutorial_source/parse.py`. Perhaps a typo, or you don’t have your virtualenv setup properly.

The output from the `(DataVizProj) $ python parse.py` should look like a bunch of dictionaries in one list. For reference, the last bit of output you should see in your terminal should look like (doesn’t have to be exact data, but the structure of {“key”: “value”} should look familiar):

```
'ARRESTED, BOOKED'},{'Category': 'OTHER OFFENSES', 'IncidentNum': '030204238',
'DayOfWeek': 'Tuesday', 'Descript': 'OBSCENE PHONE CALLS(S)', 'PdDistrict':
'PARK', 'Y': '37.7773636900243', 'Location': '800 Block of CENTRAL AV', 'Time':
'18:59', 'Date': '02/18/2003', 'X': '-122.445006858202', 'Resolution': 'NONE'}]
```

You see this output because in the `def main()` function, and you explicitly say `print new_data` which feeds to the output of the terminal. You could, for instance, not print the `new_data` variable, and just pass the `new_data` variable to another function. Coincidentally, that’s what Part II and Part III are about!

---

## — Explore further —

---

Play around with `parse.py` within the Python interpreter itself. Make sure you’re in your `MySourceFiles` directory, then start the Python interpreter from there:

```
(DataVizProj) $ python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Appletclang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To exit out of the Python shell, press `CTRL-D`.

Next, import your `parse.py` file into the interpreter. Notice there is no need to include the `.py` portion when importing:

```
>>> import parse
>>>
```

If all things go well with `import parse` you should just see the `>>>` prompt. If there's an error, perhaps you are not in the correct directory from two steps ago.

Play with the following commands. Notice to access any object defined in `parse.py` (object meaning a variable, function, etc), you must preface it with `parse`:

```
>>> parse.MY_FILE
'../data/sample_sfpd_incident_all.csv'
>>> type(parse.MY_FILE)
<type: 'str'>
>>> copy_my_file = parse.MY_FILE
>>> copy_my_file
'../data/sample_sfpd_incident_all.csv'
>>> type(copy_my_file)
<type: 'str'>
```

So we made what seems like a copy. Not so! check it out:

```
>>> id(copy_my_file)
4404350288
>>> id(parse.MY_FILE)
4404350288
>>>
```

Those numbers from calling the `id` function reflect where the variable is saved in the computer's memory. Since they are the *same* number, Python has set up a reference from `copy_my_file` to the same location that

`parse.MY_FILE` was saved. No need to allocate new space in memory for what is essentially the same variable with a different name.

Let's play with the parser function a bit:

```
>>> new_data = parse.parse(copy_my_file, ",")
>>> type(new_data)
<type: 'list'>
>>> type(new_data[0])
<type: 'dict'>
>>> type(new_data[0]["DayOfWeek"])
<type: 'str'>
>>> new_data[0].keys()
['Category', 'IncidentNum', 'DayOfWeek', 'Descript', 'PdDistrict', 'Y', 'Location', 'Time', 'Date', 'X', 'Resolution']
>>> new_data[0].values()
['FRAUD', '030203898', 'Tuesday', 'FORGERY, CREDIT CARD', 'NORTHERN', '37.8014488257836', '2800 Block of VAN NESS AV', '16:30', '02/18/2003', '-122.424612993055', 'NONE']
>>> for dict_item in new_data:
...     print dict_item["Descript"]
...
DRIVERS LICENSE, SUSPENDED OR REVOKED
LOST PROPERTY
POSS OF LOADED FIREARM
<--snip-->
BATTERY
OBSCENE PHONE CALLS(S)
>>>
```

Here we checked out the type of data that gets returned back to use from the parse function, as well as ways to simply check out what is the contents of the parsed data.

You can continue to play around; try `>>> help(parse.parse)` to see our docstring, see what happens if you feed the parse function a different file, delimiter, or just a different variable. Challenge yourself to see if you can create a new file to save the parsed data, rather than just a variable. The example in the python docs may help.

**Continue on to Part 2: Graphing →**



# Part 2: Graph

Walk through of how to visualize our parsed data with Python's matplotlib.

---

## — Module Setup —

---

- Similar as before, when you open up `graph.py` from `new-coder/dataviz/tutorial_source/graph.py`, you'll see the language environment setup, as well as an introduction to the module itself.
- The next few lines are the libraries we import. Notice how the import statements are in alphabetical order. The general rule of ordering imports, in alphabetical order:
  1. Standard Library modules
  2. External/third party packages/modules
  3. Internal/self-written modules
- When importing, we can also give the object we're importing whatever name we want because we're lazy programmers. When we `import matplotlib.pyplot as plt` we're essentially renaming the `pyplot` object (which FYI is `<type: 'module'>`) of `matplotlib` as `plt`. You don't have to name it `plt`, but it's a handy trick when you want to access different objects that the `pyplot` module has, as you'll see later.

---

## — Review of Parse Function —

---

- Once again, you see the `MY_FILE` as a global variable that points to the sample data file that's included in the repository.
- In a quick review of Part I: Parse - tutorial comments removed - we see that the `parse()` function still takes in two parameters: `raw_file` and `delimiter`. The process of the `parse()` function is as follows:
  1. Open the raw file.
  2. Read the CSV file with the appropriate delimiter, then close the file.
  3. Initialize an empty list which will be returned by the function.
  4. Grab the first row of the CSV file, the headers/column names, and assign them to the `fields` variable, which will be a list.
  5. Iterate over each remaining row in the CSV file, mapping column headers → row values, and add to our list we initialized in step 3.
  6. Return the `parsed_data` variable.

- We include the parse function here so we build on the process of parse → plot. We need to parse the data into the list of dictionaries so that we can easily tell matplotlib what and how to plot. We could, however, import it from `parse.py`. As a **challenge** to you, try editing away the parse function in `graph.py` and import it from your `parse.py`.

---

## — Visualize Functions —

---

Let's first take a look at a chunk of data that we just parsed to get a better idea of what sort of data we're working with:

```
{
  'Category'      : 'ASSAULT',
  'IncidentNum'   : '030204181',
  'DayOfWeek'     : 'Tuesday',
  'Descript'      : 'BATTERY',
  'PdDistrict'    : 'CENTRAL',
  'Y'             : '37.7981847618287',
  'Location'      : '300 Block of COLUMBUS AV',
  'Time'          : '18:15',
  'Date'          : '02/18/2003',
  'X'             : '-122.407069627873',
  'Resolution'    : 'ARREST, BOOKED'
},
```

By looking at a snippet of data, we can understand how we can play/visualize it. The kind of data we are working with is where one entry equals an incident that the San Francisco Police recorded. The following two functions are just two ways of playing with the data, but note that these functions are specific to *our* data.

**Disclaimer:** As with understanding statistics, correlation does *not* mean causation. This is a small sample size, not current, and it's from the point of view of officers reporting incidents. Take everything with a grain of salt!

---

## Visualize Days Function

---

As we read from the docstring, this will give us a visualization of data by the day of the week. For instance, are SF police officers more likely to file incidents on Monday versus a Tuesday? Or, tongue-in-cheek, should you stay in your house Friday night versus Sunday morning?

You'll also notice that the `def visualize_days()` function does not take any parameters. An option to explore would be to pass this function already-parsed data. If you feel up to it after understanding this

function, explore redefining the function like so: `def visualize_days(parsed_data):`.

Let's walk through this function like we did the parse function. Below is the walk through of comments for the code that we will want to write:

```
def visualize_days():  
    """Visualize data by day of week"""  
  
    # grab our parsed data that we parsed earlier  
  
    # make a new variable, 'counter', from iterating through each  
    # line of data in the parsed data, and count how many incidents  
    # happen on each day of the week  
  
    # separate the x-axis data (the days of the week) from the  
    # 'counter' variable from the y-axis data (the number of  
    # incidents for each day)  
  
    # with that y-axis data, assign it to a matplotlib plot instance  
  
    # create the amount of ticks needed for our x-axis, and assign  
    # the labels  
  
    # show the plot!
```

Working through the first in-line comment should force you to recall our parse function. How do we get a parsed data object that is returned from our parse function to a variable? Well thankfully we still have the parse function in our `graph.py` file so we can easily access it's parsing-abilities! Like so:

```
def visualize_days():  
    """Visualize data by day of week"""  
  
    # grab our parsed data that we parsed earlier  
    data_file = parse(MY_FILE, ",")
```

Notice how we assign `data_file` to our parse function, and the parameters we feed through our parse functions are `MY_FILE` and a comma-delimiter. Because we know the parse function returns `parsed_data`, we can expect that `data_file` will be that exact return value.

This next one is a little tricky, and not very intuitive at all. Remember earlier, we imported `Counter` from the module `collections`. This is demonstrative of Python's powerful standard library.

Here, `Counter` behaves very similarly to Python's dictionary structure (because under the hood, the `Counter`

class inherits from dictionary). What we will do with Counter is iterate through each line item in our `data_file` variable (since it's just a list of dictionaries), grabbing each key labelled "DayOfWeek".

What the Counter does is everytime it sees the "DayOfWeek" key set to a value of "Monday", it will give it a tally; same with "DayOfWeek" key set to "Tuesday", etc. This works great for very well structured data.

```
def visualize_days():  
    """Visualize data by day of week"""  
  
    # grab our parsed data that we parsed earlier  
    data_file = parse(MY_FILE, ",")  
  
    # make a new variable, 'counter', from iterating through  
    # each line of data in the parsed data, and count how many  
    # incidents happen on each day of the week  
    counter = Counter(item["DayOfWeek"] for item in data_file)
```

Notice, within Counter(...) we have an interesting loop construct: `item["DayOfWeek"] for item in data_file`. This is called a list comprehension. You can read it as, "iterate every dictionary value of every dictionary key set to 'DayOfWeek' for every line item in `data_file`." A list comprehension just a for-loop put in a more elegant, "Pythonic" way.

*Challenge yourself:* write out a for-loop for our `counter` variable.

The counter object is a dictionary with the keys as days of the week, and values as the count of incidents per day. In order for our visualization to make sense, we need to make sure the order that we plot the data makes sense. For instance, it would make no sense to plot our data in alphabetical order rather than order of the days of the week. We can force our order by separating keys and values to lists:

```
# separate the x-axis data (the days of the week) from the  
# 'counter' variable from the y-axis data (the number of  
# incidents for each day)  
data_list = [  
    counter["Monday"],  
    counter["Tuesday"],  
    counter["Wednesday"],  
    counter["Thursday"],  
    counter["Friday"],  
    counter["Saturday"],  
    counter["Sunday"]  
]  
day_tuple = tuple(["Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun"])
```

Here, `data_list` takes each key of `counter` to grab the value associated with each day. Because we manually write out each `counter` key, we force the order that we want. **Note:** a dictionary does *not* preserve order, but a list does; this is why we're electing to manually key into each value of a dictionary to make a list of each value.

The `day_tuple` is just a tuple of strings that we will use for our x-axis labels. **A quick note:** we had to make our `day_tuple` variable a tuple because `plt.xticks()` only accepts tuples for labeling the x-axis. This is because tuples are an immutable type of data structure in Python's library, meaning you can't change it (not without making a copy of the variable onto a new variable), as well as it preserves order.

We now tell `matplotlib` to use our `data_list` as data points to plot. The `pyplot` module, what we've renamed as `plt`, has a function called `plot()` which takes a list of data points to plot on the y-axis:

```
# with that y-axis data, assign it to a matplotlib plot instance
plt.plot(data_list)
```

If you are curious about the `plot()` function, open a `python` prompt in your terminal, then `import matplotlib.pyplot as plt` followed by `help(plt)` and/or `dir(plt)`. Again, to exit out of the Python shell, press `CTRL-D`.

Just creating the variable `day_tuple` for our x-axis isn't enough — we also have to assign it to our `plt` by using the method `xticks()`:

```
# Assign labels to the plot
plt.xticks(range(len(day_tuple)), day_tuple)
```

We give `plt.xticks()` two parameters, one being a list and the other being our tuple, `labels`.

The first parameter is `range(len(day_tuple))`. Here, we call `len()` on our `day_tuple` variable — `len()` returns an integer, a count of the number of items in our tuple `day_tuple`. Since we have seven items in our `day_tuple` (**pop quiz:** why do we have seven items?), the `len()` will return 7. Now we have `range()` on our length of the `day_tuple`. If you feed `range()` one parameter `x`, it will produce a list of integers from 0 to `x` (not including `x`). So, deconstructed, we fed `plt.xticks()` the following:

```
parameter 1 = [0, 1, 2, 3, 4, 5, 6]
parameter 2 = ("Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun")
```

The first parameter is so `matplotlib` knows how many ticks it needs to place.

We're nearly there! So far, we've assigned our `plt` instance data with just the y-axis variables through the `plot()` method, as well as the count and string labels for the x-axis with `xticks()`. Now all we need is to render the visualization! Here we use `plt`'s `show()` method:

```
# Render the plot!
plt.show()
```

Notice we didn't finish with `return` – you can put a `return` call at the end of the function, but we aren't returning anything, per se, and because we aren't, we don't need to have the `return` call in there.

The function all together:

```
def visualize_days():
    """Visualize data by day of week"""
    data_file = parse(MY_FILE, ",")
    # Returns a dict where it sums the total values for each key.
    # In this case, the keys are the DaysOfWeek, and the values are
    # a count of incidents.
    counter = Counter(item["DayOfWeek"] for item in data_file)

    # Separate out the counter to order it correctly when plotting.
    data_list = [counter["Monday"],
                 counter["Tuesday"],
                 counter["Wednesday"],
                 counter["Thursday"],
                 counter["Friday"],
                 counter["Saturday"],
                 counter["Sunday"]
                ]

    day_tuple = tuple(["Mon", "Tues", "Wed", "Thurs", "Fri", "Sat", "Sun"])

    # Assign the data to a plot
    plt.plot(data_list)

    # Assign labels to the plot
    plt.xticks(range(len(day_tuple)), day_tuple)

    # Render the plot!
    plt.show()
```

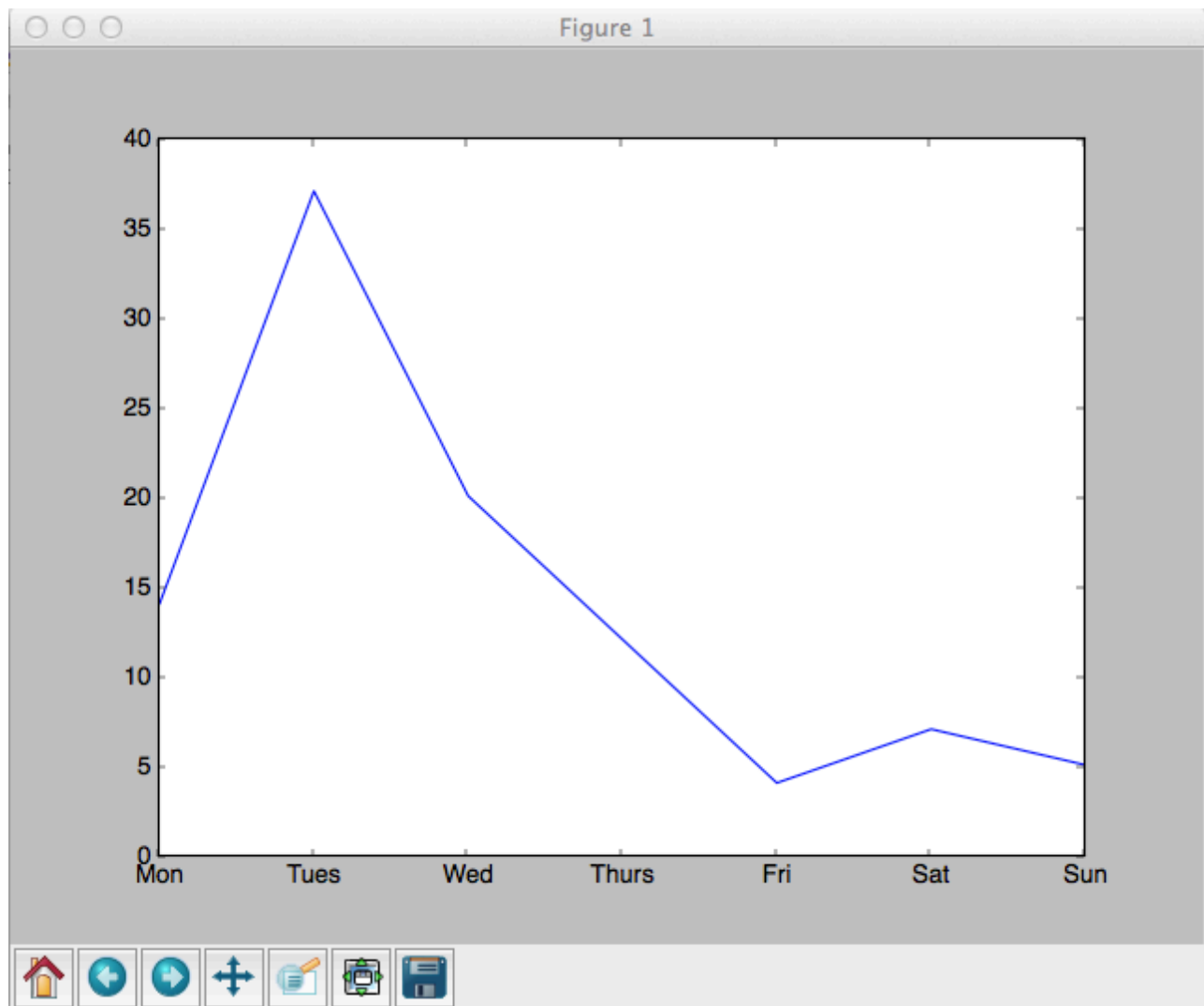
To actually see the visualization (and to test your code), add the following boilerplate code again:

```
def main():  
    visualize_days()  
  
if __name__ == "__main__":  
    main()
```

Next, save this file as `graph.py` into the `MySourceFiles` directory that we created earlier, and make sure you are in that directory in your terminal by using `cd` and `pwd` to navigate as we did before. Also — make sure your virtualenv is active. Now, in your terminal, run:

```
(DataVizProj) $ python graph.py
```

You should see a nice rendering of our graph:



When you're done marveling at your work, close the graph window and you should be back at your terminal.

You can also start up a Python shell, and play around a little bit:

```

>>> from graph import visualize_days
>>> visualize_days() # should see the graph pop up again
>>> MY_FILE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'MY_FILE' is not defined
>>> from graph import MY_FILE
>>> MY_FILE
'../data/sample_sfpd_incident_all.csv'
>>> parse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'parse' is not defined
>>> from graph import parse
>>> parse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: parse() takes exactly 2 arguments (0 given)
>>> parse(MY_FILE, ",") # should see a big list of dicts

```

Remember that `CTRL+D` exits out of the Python shell and brings you back to where you were in the terminal.

---

## Visualize Type Function

---

The next function that we will walk through, `visualize_type()`, is constructed very similarly, but takes advantage of how you can manipulate the size and image of the graph. I will not rehash familiar/repetitive lines of code since a lot is similar to `visualize_days()`.

Starting with our comment outline and function scaffolding:



```

def visualize_type():
    """Visualize data by category in a bar graph"""

    # grab our parsed data

    # make a new variable, 'counter', from iterating through each line
    # of data in the parsed data, and count how many incidents happen
    # by category

    # Set the labels which are based on the keys of our counter.
    # Since order doesn't matter, we can just use counter.keys()

    # Set exactly where the labels hit the x-axis

    # Width of each bar that will be plotted

    # Assign data to a bar plot (similar to plt.plot()!)

    # Assign labels and tick location to x-axis

    # Give some more room so the x-axis labels aren't cut off in the
    # graph

    # Make the overall graph/figure is larger

    # Render the graph!

```

The first three lines of code should look familiar. Here, we're counting over "Category" rather than "DayOfWeek" data. And since order doesn't matter to us here, we can just use `counter.keys()` and `counter.values()` to get the items we need for plotting:

```

# grab our parsed data
data_file = parse(MY_FILE, ",")

# Same as before, this returns a dict where it sums the total
# incidents per Category.
counter = Counter(item["Category"] for item in data_file)

# Set the labels which are based on the keys of our counter.
# Since order doesn't matter, we can just use counter.keys()
labels = tuple(counter.keys())

```

Next we finally use a bit of numpy magic (we had imported the numpy library as `na`):

```
# Set where the labels hit the x-axis  
xlocations = na.array(range(len(labels))) + 0.5
```

We have a new variable, `xlocations`, which will be used to help place the `plt.xticks()`. We're using the `numpy.ndarray` (aka `na`) module to access the `array` function. This turns the list that `range(len(labels))` would make into an array that you can manipulate a bit differently. Here, we're adding `0.5`. If you were to `print xlocations`, you would see `[0.5, 1.5, 2.5, ..., 16.5, 17.5]` where `0.5` was added to each int of the list. You'll see why we need the `0.5` a bit later.

Now we assign our x- & y-ticks (should be familiar to `visualize_days()`):

```
# Assign labels and tick location to x-axis  
plt.xticks(xlocations + width / 2, labels, rotation=90)
```

For the `plt.xticks()`, the first parameter should look similar to before, but here we're feeding three parameters: `xlocations + width / 2`, `labels`, and `rotation=90`. The first parameter will place the center of the bar in the middle of the xtick. `labels` we know already. `rotation=90` is, as you might have guessed, rotates each label 90 degrees. This allows our x-axis to be more readable. You can try out another values.

Notice how we can pass `xticks()` more parameters than we did before. If you read the documentation of that function, you can pass it `*args` and `**kwargs`, or arguments and keyword arguments. It mentions that you can pass matplotlib-defined text properties for the labels – so that would explain the `**kwargs` element there. If nothing is passed in for `rotation` then it's set to a default defined in their text properties documentation.

Next, we just add a little bit of spacing to the bottom of the graph so the labels (since some of them are long, like `Forgery/Counterfeiting`). We use the `.subplots_adjust()` function. In matplotlib, you have the ability to render multiple graphs on one window/function, called subplots. With one graph, subplots can be used to adjust the spacing around the graph itself.

```
# Give some more room so the labels aren't cut off in the graph  
plt.subplots_adjust(bottom=0.4)
```

I'll be honest, `0.4` was a guess-and-check. When the graph shows up, the button on the bottom, one in from the right (right next to the Save button) will show you the Subplot Configuration Tool to play with spacing.

Nearly there – before we render the graph, the actual size of the window can be played with too. The `rcParams` dictionary, explained in their docs, allows us to dynamically play with matplotlibs global settings. In particular, the `'figure.figsize'` key is expecting two values: `height + width`:

```
# Make the overall graph/figure larger  
plt.rcParams['figure.figsize'] = 12, 8
```

Again — here I just played with the numbers until I got something I liked. I encourage you to put in different numbers to change the size of your graph.

Finally, our favorite — rendering the graph!

```
# Render the graph!  
plt.show()
```

A reiteration: notice we didn't finish with `return` — you can put a `return` call at the end of the function, but we aren't returning anything, per se, and because we aren't, we don't need to have the `return` call in there.

The function all together:

```

def visualize_type():
    """Visualize data by category in a bar graph"""

    data_file = parse(MY_FILE, ",")

    # Same as before, this returns a dict where it sums the total
    # incidents per Category.
    counter = Counter(item["Category"] for item in data_file)

    # Set the labels which are based on the keys of our counter.
    labels = tuple(counter.keys())

    # Set where the labels hit the x-axis
    xlocations = na.array(range(len(labels))) + 0.5

    # Width of each bar
    width = 0.5

    # Assign data to a bar plot
    plt.bar(xlocations, counter.values(), width=width)

    # Assign labels and tick location to x-axis
    plt.xticks(xlocations + width / 2, labels, rotation=90)

    # Give some more room so the labels aren't cut off in the graph
    plt.subplots_adjust(bottom=0.4)

    # Make the overall graph/figure larger
    plt.rcParams['figure.figsize'] = 12, 8

    # Render the graph!
    plt.show()

```

To actually see the visualization (and to test your code), add the following boilerplate code:

```

def main():
    # visualize_days() # commenting out the visualize_days() function
    visualize_type()

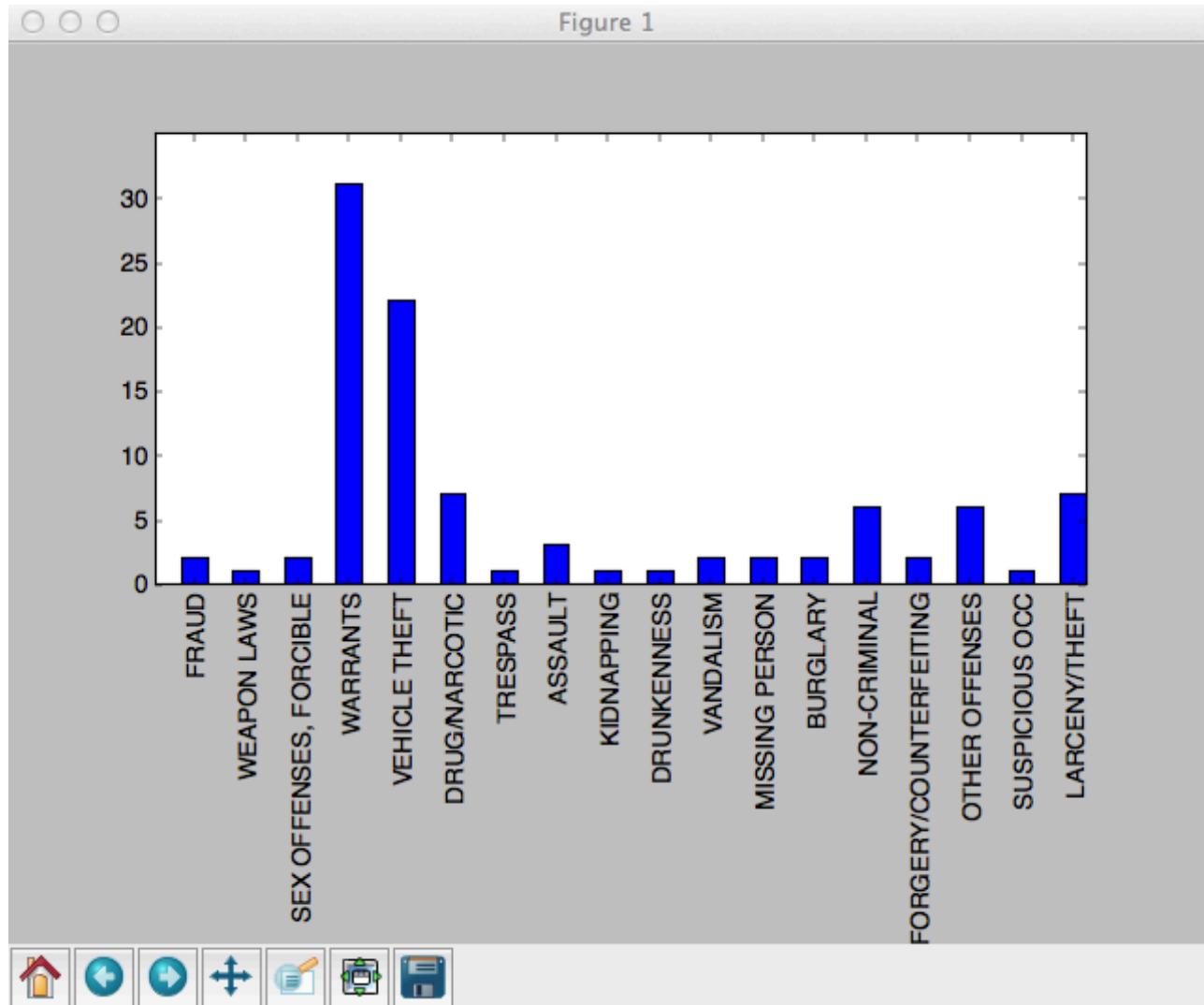
if __name__ == "__main__":
    main()

```

Next, save this file as `graph.py` into the `MySourceFiles` directory that we created earlier, and make sure you are in that directory in your Terminal by using `cd` and `pwd` to navigate as we did before. Also – make sure your `virtualenv` is active. Now, in your terminal, run:

```
(DataVizProj) $ python graph.py
```

and you should see:



When you're done marveling at your work, close the graph window and you should be back at your terminal.

You can also start up a Python shell, and play around a little bit like we did with our `visualize_days()` code. Remember that `CTRL+D` exits out of the Python shell and brings you back to where you were in the terminal.

**Continue on to Part 3: Mapping →**