Part 2: Graph

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

— Module Setup —

- Simililar 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, imported 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 chuck of data that we just parsed to get a better idea of what sort of data we're working with:

```
{
'Category'   : 'ASSAULT',
'IncidntNum' : '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 policy 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 varible `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:

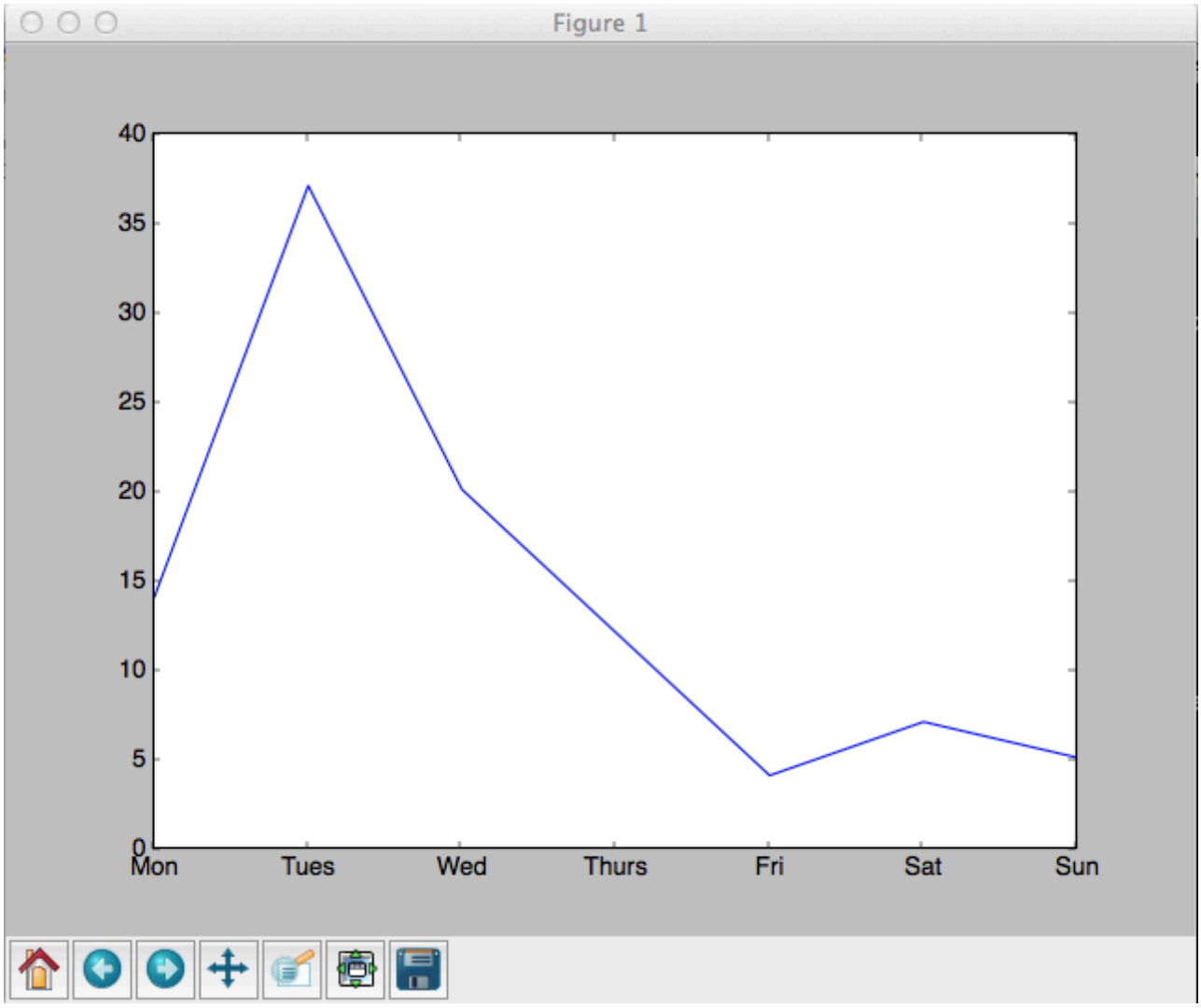
```
def main():
    visualize_days()

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 used 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 used 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.numarray` (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:

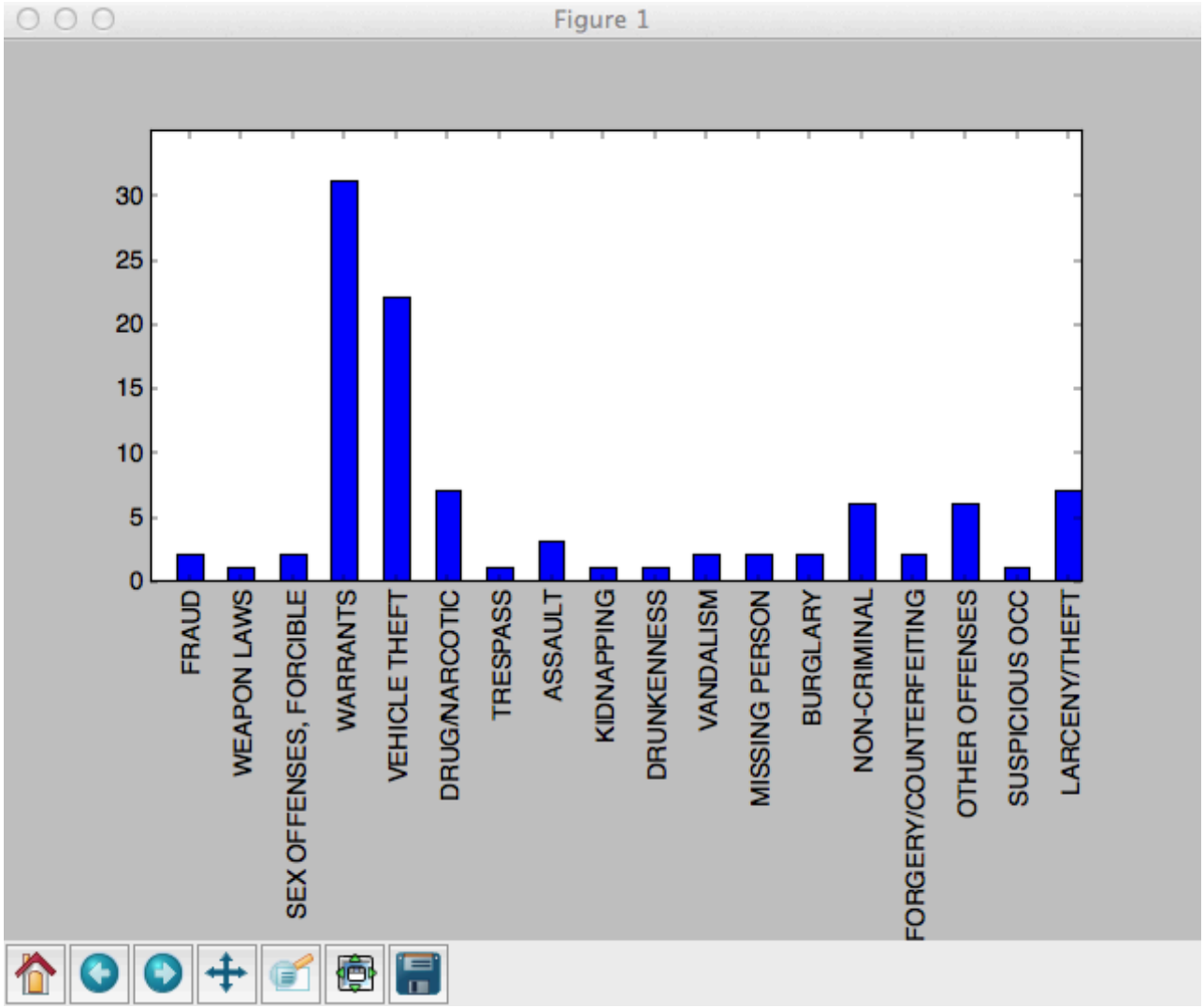
```
def main():
    # visualize_days() # commenting out the visualize_days() function
    visualize_type()

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 →