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

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 asign 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', 'IncidntNum': '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. Coincidently, 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/Apple/clang-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', 'IncidntNum', 'DayOfWeek', 'Descript', 'PdDistrict', 'Y', 'Location', 'Time', 'Dat
e', '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 ot 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 →