Part 3: Map

Walk through of how to plot our parsed data on Google Maps.

— Module Setup —

Looking at `map.py` from `new-coder/dataviz/tutorial_source/map.py`, you'll see in lines after the preamble that we're importing `xml.dom.minidom` which is Python's minimal implementation of DOM interface, as well as our own module, `parse` as `p`.

Other ways you could have done the import statements:

```
from xml.dom import minidom
import xml.dom.minidom as m
import xml.dom
import xml

import parse
from parse import parse, MY_FILE
import parse as iLoveParsingSoMuch
```

Of course, we're lazy programmers, so we're not going to `import parse as iLoveParsingSoMuch` because each time we want to refer to our `parse()` function in the `parse` module, we'd have to type out `iLoveParsingSoMuch.parse(iLoveParsingSoMuch.MY_FILE, ",")` — you can probably see why I elected `p`.

We also don't import the whole `xml` library, or `xml.dom` library for that matter. We want to run lean code, so only import the specific module that you need, or even objects (classes, functions, variables, etc) defined from within that module.

For the curious

- A package is a collection of modules (or packages). A module is one python file, so a package is a collection of python files within the same directory.
- A distributed collection of packages is can be referred to as a library.
- Python has a standard library already built-in (built-in meaning that you don't have to download extra packages, it's default within the language and just have to import what you need), but that standard library

contains many packages and modules.

  - Python will follow your import statements like a file structure. For instance, we have `new-coder/dataviz/tutorial_source/`. So within Python's standard library, `minidom` is defined within `dom`, and that within `xml`.
  - A bit of a **warning**: if you try to run `map.py` outside of `new-coder/dataviz/tutorial_source` without adjusting the `import parse`, you may see an `ImportError`. When making a package yourself for distribution, there are ways to void this issue, and you can read more in the Python docs.

— Helper Functions —

We've defined two helper functions for our `create_gmap()` function: `create_document()` and `create_placemark`. I won't spend too much time on the detail of these two functions, but what I want you to understand is the concept of breaking out your code to have functions do one thing and one thing only. We don't want `create_gmap()` to get too muddled up, the main reasons being that it lends to code being far more readable, as well as testable and debugging.

The `create_document(title, description='')` function essentially will create/initialize a KML document. It first makes an XML document, then defines it as KML, grabs common KML attributes that are defined at `www.opengis.net` (which catalogs web resources for anyone to refer to). Lastly, it creates meta data that we want for our map: Title and Description.

```python
def create_document(title, description=''):
    """Create the overall KML document."""

    # Initialization of an XML doc
    doc = xml.dom.minidom.Document()

    # Define as a KML-type XML doc
    kml = doc.createElement('kml')

    # Pull in common attributes and set it for our doc
    kml.setAttribute('xmlns', 'http://www.opengis.net/kml/2.2')
    doc.appendChild(kml)

    # Create common elements that Google will read/plot
    document = doc.createElement('Document')
    kml.appendChild(document)
    docName = doc.createElement('title')
    document.appendChild(docName)
    docName_text = doc.createTextNode(title)
    docName.appendChild(docName_text)
    docDesc = doc.createElement('description')
    document.appendChild(docDesc)
    docDesc_text = doc.createTextNode(description)
    docDesc.appendChild(docDesc_text)
    return doc
```

The `createElement()` and `appendChild()` is specific to DOM functions that the `xml.dom.minidom` gives us access to. We first create an element (either Document, title, or description), then assign that element a value if needed (title, and description, if given). Finally, we return the initialized document.

The `create_placemark(address)` creates an initial XML document so we can build one placemark (equal to one piece of our data). The function actually creates the placemark data by doing the same process from earlier, `createElement` to create a type of DOM element, and assign it a value if needed (e.g. name, coordinates, description). This just returns one placemark in the correct KML format.

```python
def create_placemark(address):
    """Generate the KML Placemark for a given address.
    This is the function that takes the info from the
    file we parse at the end of this script"""

    # Create an initial XML document
    doc = xml.dom.minidom.Document()

    # Create elements for Placemark and add to our new doc
    pm = doc.createElement("Placemark")
    doc.appendChild(pm)
    name = doc.createElement("name")
    pm.appendChild(name)
    name_text = doc.createTextNode('%(name)s' % address)
    name.appendChild(name_text)
    desc = doc.createElement("description")
    pm.appendChild(desc)
    desc_text = doc.createTextNode('Date: %(date)s, %(description)s' % address)
    desc.appendChild(desc_text)
    pt = doc.createElement("Point")
    pm.appendChild(pt)
    coords = doc.createElement("coordinates")
    pt.appendChild(coords)
    coords_text = doc.createTextNode('%(longitude)s,%(latitude)s' % address)
    coords.appendChild(coords_text)
    return doc
```

I want to point out the following syntax: `Date: %(date)s, %(description)s' % address`. The parameter, `address` is passed to the `create_placemark()` function. We can access elements in that parameter (you'll see later that it's a dictionary) with Python's 'string-fu' — it has a built-in method with the `%` operator (aka Modulo) for string formatting, following the convention `format % values`. You can access values in a dictionary by calling the dictionary key in parenthesis:

```python
>>> print '%(language)s has %(number)03d quote types.' % {"language": "Python", "number": 2}
Python has 002 quote types.
```

You see that `(language)` is specified to be a string with the `s`, and `(number)` is a decimal specified by the `d`. The `03` in front of the `d` refers to number of digits (3) and with zeros padding the number. More information can be read in the Python docs.

— Create G-Map! —

Now on to the good stuff. The function `create_gmap(data_file)` uses the two helper functions to build a KML document with our data.

Again with our initial comment setup:

```
def create_gmap(data_file):
    # Create a new KML doc with our previously-defined
    # create_document() function

    # Get the specific DOM element that we created with create_document()
    # Returns a list, so call the first one

    # Iterate over our data to create KML document
    for line in data_file:
        # Parses the data into a dictionary

        # Avoid null values for lat/long

        # Calls create_placemark() to parse line of data into KML-format

        # Adds the placemark we just created to the KML doc

    # Now that all data is parsed in KML-format, write to a file so we
    # can upload it to maps.google.com
```

The first that we need to do is just to create a new KML document for us to work with. We'll use our helper function, `create_document` and pass in a title and description as parameters to create a new variable, `kml_doc`:

```
    # Create a new KML doc with our previously-defined
    # create_document() function
    kml_doc = create_document("Crime map", "Plots of Recent SF Crime")
```

Next, we just want to get that specific DOM element, `"Document"` to build each placemark to. So we need to create the document, then grab the right element, coincidently named Docoment, so we can add placemarks to it.

```python
# Get the specific DOM element that we created with create_document()
# Returns a list, so call the first one
document = kml_doc.documentElement.getElementsByTagName("Document")[0]
```

Next, we iterate through the parsed data (`data_file`) that we fed the `create_gmap(data_file)` and make sure we build our dictionary of data, `placemark_info` so that `create_placemark` can build a placemark out of it.

```python
# Iterate over our data to create KML document
for line in data_file:
    # Parses the data into a dictionary
    placemark_info = {'longitude': line['X'],
                      'latitude': line['Y'],
                      'name': line['Category'],
                      'description': line['Descript'],
                      'date': line['Date']
                      }

    # Avoid null values for lat/long
    if placemark_info['longitude'] == "0":
        continue

    # Calls create_placemark() to parse line of data into KML-format
    placemark = create_placemark(placemark_info)

    # Adds the placemark we just created to the KML doc
    document.appendChild(placemark.documentElement)
```

So for each line in our `data_file`, we take certain values of that line, `X`, `Y`, `Category`, etc, and assign it to a key. If, for whatever instance, longitude is `0`, we'll skip over it. The assumption is if the longitude is `0`, then we can't plot it (or it will be plotted as `0,0` and screw with our map). This is a simple form of skipping over errors in the data.

We then create the variable `placemark` by calling the `create_placemark()` function, and feeding it our dictionary, `placemark_info`. `create_placemark()` will return an object that can easily be added to our KML document, `document`:

```python
    # Calls create_placemark() to parse line of data into KML-format
    placemark = create_placemark(placemark_info)

    # Adds the placemark we just created to the KML doc
    document.appendChild(placemark.documentElement)
```

So looping over each line item is done, we've built our KML document, now how to we *get* that document so we can upload it to Google Maps? We can do that with Python's file I/O — by opening a file (if it doesn't exist, it will be created for us), and writing to that file.

```
# Now that all data is parsed in KML-format, write to a file so we
# can upload it to maps.google.com
with open('file_sf.kml', 'w') as f:
    f.write(kml_doc.toprettyxml(indent="  ", encoding='UTF-8'))
```

This is a new loop construct: `with` — it allows us to not have to worry about closing a file; it will be done automatically for us.

So `with open('file_sf.kml', 'w') as f` assigns the opened file as f; it also will either open the file `file_sf.kml` or create it (**note**: it will be in your current directory unless you specify otherwise, like `/Users/lynnroot/NotMyDevFolder/file_sf.kml` with absolute file paths), and give it `write` capabilities (versus read-only).

Then we write the `kml_doc` to the file. We use the `toprettyxml()` method so that we can specify encoding and indentation, making it more readable for us.

Let's see the `create_gmap()` function all together:

```python
def create_gmap(data_file):
    """
    Creates Google Maps KML Doc.
    Returns a KML file to be uploaded at maps.google.com.
    Navigate to 'My places' -> 'Create Map' -> 'Import' to
    upload the file and see the data.
    """

    # Create a new KML doc with our previously-defined
    # create_document() function
    kml_doc = create_document("Crime map", "Plots of Recent SF Crime")

    # Get the specific DOM element that we created with create_document()
    # Returns a list, so call the first one
    document = kml_doc.documentElement.getElementsByTagName("Document")[0]

    # Iterate over our data to create KML document
    for line in data_file:
        # Parses the data into a dictionary
        placemark_info = {'longitude': line['X'],
                          'latitude': line['Y'],
                          'name': line['Category'],
                          'description': line['Descript'],
                          'date': line['Date']
                          }

        # Avoid null values for lat/long
        if placemark_info['longitude'] == "0":
            continue

        # Calls create_placemark() to parse line of data into KML-format
        placemark = create_placemark(placemark_info)

        # Adds the placemark we just created to the KML doc
        document.appendChild(placemark.documentElement)

    # Now that all data is parsed in KML-format, write to a file so we
    # can upload it to maps.google.com
    with open('file_sf.kml', 'w') as f:
        f.write(kml_doc.toprettyxml(indent="  ", encoding='UTF-8'))
```

That's it! Now we just have some boiler code for that `main()` function:

```
def main():
    data = p.parse(p.my_file, ",")

    return create_gmap(data)

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

Here we just first parse our data, then return the KML document using that parsed data.

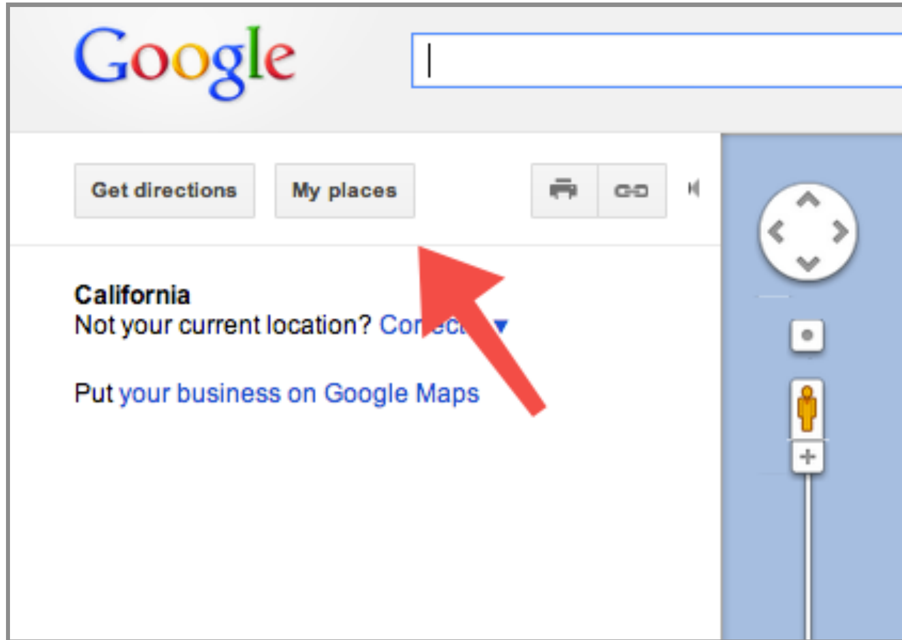
Next, save this file as `map.py` into the `MySourceFiles` directory that we created earlier, and make sure you are in that directory in your terminal by using `cd` and `pwd` to navigate as we did before. Also – make sure your virtualenv is active. Now, in your terminal, run:

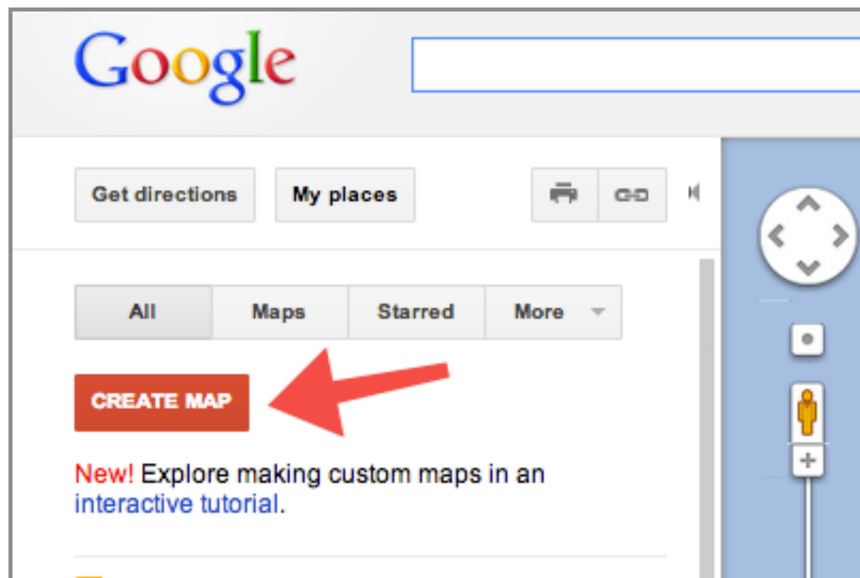
```
(DataVizProj) $ python map.py
(DataVizProj) $ ls
```

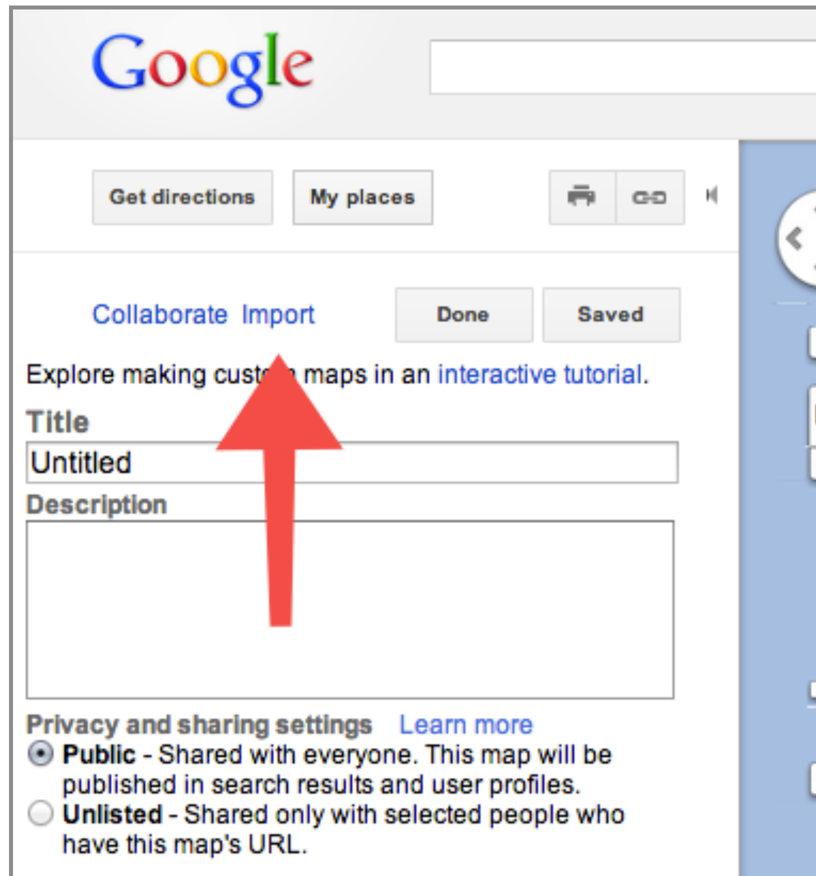
You should see `file_sf.kml` file now! You can open it up in your text editor; a snipit should look like this:

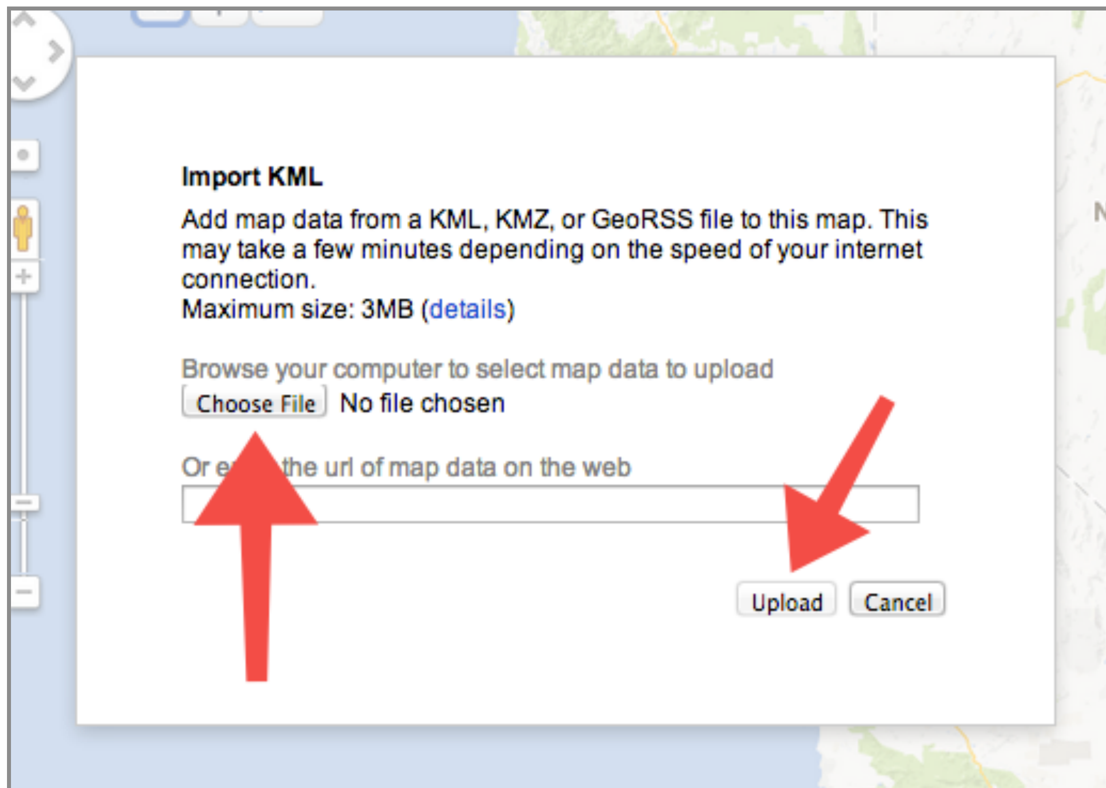
```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <name>
      Crime map
    </name>
    <description>
      Plots of Recent SF Crime
    </description>
    <Placemark>
      <name>
        FRAUD
      </name>
      <description>
        Date: 02/18/2003, FORGERY, CREDIT CARD
      </description>
      <Point>
        <coordinates>
          -122.424612993055,37.8014488257836
        </coordinates>
      </Point>
    </Placemark>
```

To see it up on Google maps, navigate to maps.google.com, then click the button "My Places", then "Create Map", then "Import", and select your `file_sf.kml` and upload:









Go ahead and upload it and marvel in your new Google Map!