Python Programming Basics for GIS Professionals

Description:

Programming tools are now a standard feature within GIS software packages and allow professionals to automate, speed up, and become more precise in their analytic work. This workshop is designed for GIS professionals and students who have little to no experience or exposure to computer programming. Core programming concepts related to GIS work will be presented using the Python programming language. The workshop will be focused on guiding attendees through hands-on modules designed to provide the essential skills to programmatically manipulate data as part of a GIS workflow. This workshop is designed to be preparation for the afternoon workshop on Getting Started with Python in ArcGIS but may be taken independently.

Specific Topics Include:

- Core programming concepts
- Working with CSV files within Python
- Data cleaning and processing tasks
- Explore GIS data using Python
- Preparing data sets for ingestion into GIS software

Computing and Software Needs:

- Your own laptop; computers will not be provided.
- Python IDE
  - PyScripter (http://sourceforge.net/projects/pyscripter/files/) (v. 2.6.0 32-bit) NOTE: The zip files contain portable versions of PyScripter. No installation is needed. Just unzip the archive and start using PyScripter.
  - JetBrains PyCharm Community Edition (https://www.jetbrains.com/pycharm/download/) NOTE: This IDE requires admin privileges to install.
  - IDLE is also shipped with ArcGIS for Desktop, so it can also be used.
- OSM Nominatim API module (https://pypi.python.org/pypi/nominatim/0.1) NOTE: Windows users will need to download the Windows installer .exe file.
- ArcGIS 10.1+ for Desktop (Standard or Advanced)

Instructors:

James Whitacre, GIS Specialist, University Library, University of Illinois at Urbana-Champaign

As the GIS Specialist in the Main Library at the University of Illinois at Urbana-Champaign, Mr. Whitacre's primary role is to provide GIS consultation and research assistance for faculty, staff, and students. Additionally, he teaches a myriad of GIS workshops for beginner to advanced users and helps manage the Library's GIS data and software assets. He is also a central resource for the GIS community on campus to promote the use of GIS in research. Mr. Whitacre holds a Master of Science in Geography and was previously the GIS Manager for the Carnegie Museum of Natural History.

Zoe Zaloudek, GIS Specialist, Illinois State Water Survey, Prairie Research Institute

Zoe earned her B.A. in Geography at the University of Illinois-Urbana in 2005. Shortly afterward, she began working at the Illinois State Water Survey (ISWS). As a GIS Specialist there, her work is currently split between FEMA’s Risk Mapping, Assessment and Planning program (Risk MAP), and the Midwestern Regional Climate

Center (MRCC). She started using Model Builder in 2007 to better catalog and streamline workflows. Upon starting with the MRCC in 2010, she realized the power of Python scripts and hasn't looked back!

I. Introductions

II. IDE Installation

- URLs to the softares are located above.
- For folks who can't download the files, we have it on a flash drive.
- Recommend installing the software if you can. Otherwise, you can use the portable version
- Already using a Python IDE? If you feel comfortable in your own envirionment, you can use it, but we can't provide a guarentee that we can troubleshoot or support issues.
- If all esle fails, you may need to use IDLE, which ships and is installed with ArcGIS for Desktop

Apple folks:

- Download and install developer tools: https://itunes.apple.com/us/genre/mac-developer-tools/id12002?mt=12 (https://itunes.apple.com/us/genre/mac-developer-tools/id12002?mt=12)
- Open up terminal and run `pip install nominatim`. You may need to use sudo.
- We may not be able to support this very well

Inside the IDE

- Open the IDE
- Let's take a quick tour!
  - Editor
  - Code explorer
  - Interpreter
  - Messages
  - Add line numbers
  - Change the theme

Type the code below and run it! This is your first code (maybe...for some

In [ ]:
```
print "hello world"
```

III. What is Python?

Time for PowerPoint!!!!

In [ ]:

Break

IV. Reading Data with Python

- When we read in a file we need to store the data in some way. There are many methods and structures to accomplish this.
- The best way will be something you are comfortable with and fits the data. Chances are you'll develop one way you're comfortable with that works with most of the data you work with.

Exercise: Opening and Reading Files

- There are two methods of opening a file: **`open()`** and **`close()`** OR **`with`**.
- We will go over both in the next 2 exercies. We will work with the **`with`** statement the most.
- In both cases, start with the file path as a string. In this case, we're doing a relative path because the file is a sibling to this notebook in the file system.

For more commands and options: https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files (https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files)

In [ ]:
```
filename_relative = 'Data/CSV/chicagoCSA.csv'
filename_full = 'C:/Users/jvwhit/Documents/IPythonNotebooks/ILGISA/Data/CSV/chicacoCSA.csv'

print filename_relative
print filename_full

# Mac full paths will look different!
#filename_full = '/Users/wickes1/Documents/wickes-ilgisa-workshop-2015/chicacoCSA.csv'
```

Now we need to make our file object.

We will print the result to see what it looks like

In [ ]:
```
f = open(filename_relative, 'rU')

print f
```

Notice that it indicating that it is an open file (the object)

Now we can read in the file. We can see the raw text values using a **`print`** statment

In [ ]:
```
data = f.read()

print data
```

There are many ways to read in files and well worth some investigation time. But there's a better way to read in a file based on lines.

`readlines()` is a very common method for reading in individual lines of a file.

Let's take a moment here to comment on the notation system happening.

We can see that **`f`** is becoming our open file object. This object is waiting to be read.

We can then pick whichever reading method we want to perform on it, and then do so via **`f.methodname(<parameters>)`**.

These methods are functions that we are calling on the objects.

In [ ]:
```
f = open(filename_relative, 'rU')

data = f.readlines()

print data
```

Is the output very readable? Let's make it more readable usinga **`for`** statement

In [ ]:
```
for e in data:
    print e
```

Check out those extra blank lines. This can be an artifact from coming in from Excel.

When you see this, use **`strip()`** to clean things up. This function cleans off any white space characters from both the left and the right of a line of text.

There are also **`rstrip()`** and **`lstrip()`** in case those would be useful.

In [ ]:
```
for e in data:
    print e.strip()
```

Exercise: Opening and Reading CSV Files with the CSV Module

What is a module?

- Extensions that can be imported and used in Python scripts to expand the buil-in capabilities
- Contain pre-written code to help out with specialized tasks and defines functions, classes and variables for those tasks
- Also allows you to logically organize your Python code
- Single file consisting of Python code and also include runnable code
- Modules are imported into a script using the **`import`** statement
- Many times you need to download and install new modules

What is the CSV module?

- It knows what a CSV is, how the text should behave or look, and how it should be sanitized, etc.
- It brings in helper functions for working with CSV files to help the programmer save time
- When opening a CSV file, CSV module is imported using the **`import csv`** statement
- You can do this yourself.....but you don't want to. Just use the CSV module!
- Luckily, the CSV mocule is included in the ArcGIS for Desktop Python installation

After we **`import csv`**, we can now call the csv module functions by appending them to **`csv`**.

The two core functions you'll use are:

- **`csv.reader(file object)`** to create a reading object
- **`csv.writer(file object`** to create a writing object.

When we write files we use these two functions:

- **`writerow(row)`** to write a single row
- **`writerows(list of lists)`** to write many rows

Below are some basic formulas for reading in data. Many of these formulas came from chapter 6 of the Python Cookbook (third edition) (http://chimera.labs.oreilly.com/books/1230000000393). Not all of this syntax will make sense, but just treat it as a formula for now. Things will make more sense as you learn more about Python.

You can read the chapter for free here: http://chimera.labs.oreilly.com/books/1230000000393/ch06.html (http://chimera.labs.oreilly.com/books/1230000000393/ch06.html)

In [ ]:
```
import csv

with open(filename_relative, 'rU') as file_in:
    file_in = csv.reader(file_in)
    headers = next(file_in)
    data = []
    for line in file_in:
        data.append(line)

print 'Headers:', headers
print 'Data:', data

    # What data type are you seeing with the "print 'Data:', data" statment?
```

In [ ]:
```
for l in data:
    print l
```

Now this looks like something we can start dealing with. Each row is coming in as a list, with each cell value as an element within that Nth corresponding position. Let's do some quick sanity checking.

In [ ]:
```
for row in data:
    print len(row)
```

Is each row the same length?

If it looks good, we can now loop through and get all the city names...

In [ ]:
```
for row in data:
    print row[1]
```

Again, there are many ways to get at the data from a CSV file. Take some time to experiment with a few other techniques. You'll find your comfort zone. What I've shown here is one of the simplest possible mechanisms. There are more advanced methods for more complex data.

Break

Creating Scripts with Python

Switching gears...

At this point we've looked at how to work with lists and read in a CSV into a list of lists. This means that we can get into a CSV file and access data columns within it. Let's take another file with free text city names. Let's work on writing a script that will loop through all the city names, lookup the coordinates for those places, and then output the results into a new file.

We can make a rough outline of our program...this is called '`pseudo code`':

```
* Read in the file
* Access the city names
* Run city names through our OpenStreet Map geocoding tool called `Nominatim`
* Save those geocoding latitude and logitude values to new fields/columns
* Output a new file
```

We already know how to do the first two tasks: open the file and get to the city names. What we need to work on next is a way to run those values through the Open Street Map nominatim tool.

Exercise: Playing with `nominatim`

Installation

- Download the instal file here: https://pypi.python.org/pypi/nominatim/0.1 (https://pypi.python.org/pypi/nominatim/0.1)
- Admin Privleges:
  - Download and install the .exe file
  - You may get some warning messages...Just click ok, and things should work.
- No Admin Privileges:
  - Download the 'nominatim-0.1' folder from https://uofi.box.com/ILGISA-Python (https://uofi.box.com/ILGISA-Python)
  - Copy the 'nominatim' and 'nominatim.egg-info' folders to 'C:\Python27\ArcGIS10.X\Lib\site-packages' replace the *x* with your version!!
- Note about ArcGIS 32 bit vs. 64 bit geoprocessing

Before we can plug this tool into a script we need to take some time getting to know how it works. The best place to start is to replicate what's in the documentation: https://github.com/damianbraun/nominatim (https://github.com/damianbraun/nominatim)

Test that Nominatim was installed correctly

In [ ]:
```
from nominatim import Nominatim, NominatimReverse #import the modules

nom = Nominatim() # instantiation of a Nominatim object.

result = nom.query('Springfield, IL') # act on this object, by doing a query

print result
```

That's a bunch of stuff! We need to dig in to see the data structure though. Here we're going to meet the other container type: the dictionary.

First, let's see what it is returning to us.

In [ ]:
```
print type(result)
```

So this is a list, and we can see that by the opening and closing square brackets. Let's look at an easy way to see what's inside of a list.

In [ ]:
```
# how long is it?

print len(result)
```

So there are 20 things in here...

Llet's look at this using how we know to loop through lists

In [ ]:
```
for each in result:
    print each
```

But what are these things?

In [ ]:
```
for each in result:
    print type(each)
```

Dictionaries

dicts are container-like objects but don't work at all like **list**s.

dicts are key/value pairs, almost like mini databases. For example, we know that cities have properties like latitude and longitude. Instead of storing these in specific positions we can name those values so we can look them up directly.

For example, we could say this:

In [ ]:
```
springfield = {'lat': '39.7989763', 'lon': '-89.6443688'}
print springfield
```

Let's look a this syntax.

A `dict` is stored withthin a pair of curly brackets {}. Keys and values are separated by a colon, with the key on the left and the value on the right. The **`springfield`** dictionary above has two key/value pairs.

We can access these values with a notation somewhat similar to looking things up in a list. The name of the key goes inside square brackets after the name of the variable. Note that I did have to include the quotes for the key names because they are strings. You can use any object as a key, but remember that you do need to look it up later.

Ever used JSON or need to? You'll interact with that data format much like working with a Python dictionary.

In [ ]:
```
print springfield['lat']
```

In [ ]:
```
print springfield['lon']
```

Now that we know a little bit better about how to read a dictionary, we can take a closer look at the first result coming in for Springfield.

Remember the index notation to look up items in a list.

In [ ]:
```
print result[0]

first_result = result[0]
```

Can we make this more readable?

In [ ]:
```
for each in first_result:
    print each
```

So that will print the key names, can we get the values?

Note that I don't have to put key in quotes because it is holding the full object value

In [ ]:
```
for key in first_result:
    print key, ": ", first_result[key]
```

This is helpful. We can see the key values on the left along with the example values next to that. Looks like we can get the lat and lon values from keys called `'lat'` and `'lon'`. Let's try that now.

In [ ]:
```
print first_result['lat'], first_result['lon']
```

The nominatim package will return a list of results for your text query, with the best hit being the first in the list. Once we have this first result, we can extract information from it. Let's put this all together.

In [ ]:
```
first = result[0]
name = first['display_name']
lat = first['lat']
lon = first['lon']

print name
print lat, lon
```

Exercise: Working with Functions

But what happens when we want to do this systematically? Say, we have a list of 800 city names and we want to gather the lat/lon results for them all? What will really help us is writing a function. These allow us to custom define a series of commands that we can call as needed. Functions can take input and return output.

In [ ]:
```
def get_lat_lon(nom_results):
    first = nom_results[0]  #get the first result
    lat = first['lat']
    lon = first['lon']
    return {'lat': lat, 'lon': lon}
```

So we can throw the query command into this function and get what we want

In [ ]:
```
get_lat_lon(nom.query("springfield, IL"))
```

The data are being returned as a dictionary, so we can access the values by name.

Let's put all this together into a basic script.

In [ ]:
```
from nominatim import Nominatim

def get_lat_lon(nom_results):
    first = nom_results[0]  #get the first result
    lat = first['lat']
    lon = first['lon']
    return {'lat': lat, 'lon': lon}

cities = ["Champaign, IL", "Springfield, IL", "Bloomington, IL"]

nom = Nominatim()

for city in cities:
    lookup = get_lat_lon(nom.query(city))
    print lookup['lat'], lookup['lon']
```

This is actually an entire Python script. It is still very simple because it is just looping over a small group of cities and plainly printing out the results, but hopefully you can see where we would add in code to read in a file and write out an output.

Note: The instantiation of nom is happening outside the function...which is kind of bad practice. However, reinstatiating it for every query significantly slows down the process. This is a case where we made a tweak to optimize the code for speed.

Exercise: Writing Out Files

We've got some output now that we're just printing to our console, but we'd like to write it out to a file so we can save our results and read them into something like ArcGIS.

We're going to look at two different ways to write out these results:

- First is the pretty raw method
- Second is the CSV module method.

We can use our script again as a base, but we will add in a way to store all the results

The 'Pretty Raw Method'

In [ ]:
```
from nominatim import Nominatim

def get_lat_lon(nom_results):
    first = nom_results[0]  #get the first result
    lat = first['lat']
    lon = first['lon']
    return {'lat': lat, 'lon': lon}

cities = ["Champaign, IL", "Springfield, IL", "Bloomington, IL"]

nom = Nominatim()

root_folder = 'C:/Users/jvwhit/Documents/IPythonNotebooks/ILGISA/Data/CSV/' # nee
d this for working in the IDE

output = open(root_folder + 'latlonresults_raw.txt', 'wt') # note the use of 'wt'
here, windows users might use 'w'

headers = "city\tlat\tlon\n" # manually putting in the tabs '\t' and newline '\n'

output.write(headers) # writing the headers to the open file

for city in cities:
    lookup = get_lat_lon(nom.query(city))
    line = city + "\t" + lookup['lat'] + "\t" + lookup['lon'] + "\n" # creating t
he line contents
    output.write(line) # writing the line to the file

output.close() # closing the file
```

This is a screenshot of the output file open in Word where we can see how the white space characters are formatted. We can see that **`\t`** turned into tabs and **`\n`** turned into a newline. Given that there are commas in the city name we need to use a delimiter other than a comma. You'll want to customize this according to how you need to read in your data into the next program.

```
city→lat → lon¶
Champaign, IL → 40.1164205→-88.2433829¶
Springfield, IL→39.7989763→-89.6443688¶
Bloomington, IL→40.4731073→-88.9941403¶
¶
```

This method of 1) open the file, 2) do the loop while writing, 3) closing the file can be valuable when you have very complex structures determining what is written or you are reading in stuff from other files.

The 'CSV Module Method'

For the other example we'll use another recipe from the Python Cookbook (http://chimera.labs.oreilly.com/books/1230000000393/ch06.html (http://chimera.labs.oreilly.com/books/1230000000393/ch06.html)) to make this happen.

A major disadvantage to the previous method is that it does not sanitize the data in any way. For example, we know that none of the city names have a **`\t`** character within them because we can see them all. However, if you are pulling in data from an external source and have thousands of records it isn't always possible to check what the values are.

Additionally, sometimes the destination for this output file will require a specific delimiter, and that might be a common character for the data that you are working with. Therefore, it would be nice to use a module that automatically handled these cases for us. The **`csv writer`** tools will automatically handle this.

There are many ways to use the CSV module to write a file. This is just one way. Again, I encourage you to explore and find a recipe that works well with your own data and methods. The method shown here will take lists and write those out such that each element in a list belongs to a different column.

In [ ]:
```
from nominatim import Nominatim
import csv

def get_lat_lon(nom_results):
    first = nom_results[0]  #get the first result
    lat = first['lat']
    lon = first['lon']
    return {'lat': lat, 'lon': lon}

cities = ["Champaign, IL", "Springfield, IL", "Bloomington, IL"]

nom = Nominatim()

headers = ["city", "lat", "lon"]

result_rows = []

for city in cities:
    lookup = get_lat_lon(nom.query(city))
    line = [city, lookup['lat'], lookup['lon']] # creating the line contents
    result_rows.append(line)

# at this point I've stored all my data locally and I just need to write it out.
# This is not ideal when dealing with LARGE amounts of data.
# Try the previous method if you are getting out of memory errors.

root_folder = 'C:/Users/jvwhit/Documents/IPythonNotebooks/ILGISA/Data/CSV/' # nee
d this for working in the IDE

with open(root_folder + 'latlonresults_csvmodule.csv', 'wt') as latlonout: # note
the use of 'wt' here, windows users might use 'w'
    latlonout = csv.writer(latlonout) # creates a writing object
    latlonout.writerow(headers) # writes out one line: the headers
    latlonout.writerows(result_rows) # writes out all the lists within this list
```

We can look at this file now in Word.

```
city,lat,lon¶
"Champaign, IL",40.1164205,-88.2433829¶
"Springfield, IL",39.7989763,-89.6443688¶
"Bloomington, IL",40.4731073,-88.9941403¶
```

Looks a little different. The quotes around the city name were automatically inserted by the CSV module to account for that value having the literal value of the delimiter character. Both of these file open up in Excel the same.

Now the only piece missing from this script is reading in city names from a file. Let's put all this together. We're going to start with an existing data file, read in the location information, and output a new version of this file with the addition of the the coordinates data.

Exercise: Create a Script to Read and Write CSV Files

The first step of any process here is to make an assessment of the file that we are working with. **`JeopardyContestants_Illinois.csv`** contains data on 409 Jeopardy! players from 1984-2014. We can view it in excel to get a quick sense of things...

	1	2	3	4	5	6	7	8
1	ID	name	year	loc	numgames	totalwinnings	state	studentflag
2	3513	Umiko Post	2002	Peoria, Illinois	2	0	illinois	
3	1170	Veronica Fazio	2005	Roselle, Illinois	1	-1200	illinois	
4	7186	Kristi Springer	2011	Naperville, Illinois	1	3600	illinois	
5	7455	Edgar Mihelic	2011	Chicago, Illinois	1	8398	illinois	
6	4728	Gretchen Wahl	2008	Chicago, Illinois	1	3399	illinois	
7	2301	Bob Bearse	1988	Chicago, Illinois	1	0	illinois	
8	5847	Jennifer Pasche	2002	McHenry, Illinois	1	0	illinois	
9	5844	James Pethokoukis	2002	Chicago, Illinois	2	25600	illinois	
10	4623	Brandon Jones	2008	Chicago, Illinois	1	22000	illinois	
11	176	Kathleen Hennessey	1998	Western Springs, Illinois	1	2800	illinois	
12	7218	Lindsey Thiesfeld	2011	Clarendon Hills, Illinois	2	30802	illinois	TRUE
13	7215	Andrew Van Duyn	2011	Wheaton, Illinois	1	6000	illinois	TRUE
14	7665	Taylor Cope	2012	Chicago, Illinois	2	56000	illinois	
15	1545	Susan Lee	1997	Glencoe, Illinois	1	100	illinois	
16	8712	Rich Hansen	2013	Chicago, Illinois	1	9155	illinois	
17	6770	Jessica Trudeau	2010	Barrington, Illinois	2	68802	illinois	
18	2511	Matt Thomas	2006	Chicago, Illinois	1	4399	illinois	
19	3530	Jason Sass	2002	Springfield, Illinois	1	0	illinois	
20	4293	Gina Bronsberg	2000	Chicago, Illinois	2	14800	illinois	TRUE
21	4745	Ellen Eden	2008	Dorsey, Illinois	1	11500	illinois	
22	4607	Carl Gilbertsen	2000	Chicago, Illinois	4	57604	illinois	TRUE
23	4601	Dave Sabath	2000	Western Springs, Illinois	2	14400	illinois	TRUE
24	8133	Ashok Poozhikunnel	2013	Wheaton, Illinois	6	129600	illinois	

So let's use what we've learned to open it up and explore things.

In [ ]:
```
import csv

root_folder = 'C:/Users/jvwhit/Documents/IPythonNotebooks/ILGISA/Data/CSV/' # nee
d this for working in the IDE

source = root_folder + 'JeopardyContestants_Illinois.csv'

playerdata = [] # creates and empty list to hold the players' data

with open(source, 'rt') as file_in:
    file_in = csv.reader(file_in) # converts 'file_in' to a 'csv.reader' object
    headers = next(file_in) # reads the first line
    for row in file_in: # the file 'curser' now starts at the second row
        playerdata.append(row)

print 'Headers:', headers # so these are our column names
print 'Rows:', len(playerdata) # how many data rows we have
```

How can we get all the city names? Pretty easily.

In [ ]:
```
cityindex = headers.index('loc')

for player in playerdata[:10]: # handy trick to just get the first 10 values
    print player[cityindex]
```

But how many cities are there? For this we need to:

1. Collect all the city names
2. Determine how many unique values there are

This is actually a pretty straight forward task. Python has an object type called **set**, which is sort of like a list, but has unique properties and tools based on set theory. You can read more about this data structure in the documentation (https://docs.python.org/2/tutorial/datastructures.html#sets (https://docs.python.org/2/tutorial/datastructures.html#sets)). **set**s make list manipulation very very simple with some basic application of the set theory. Remember to look for the green boxes in the documentation for examples.

For those of you who know SQL, there are already some **set** theory tools implemented. Remember the Venn diagrams all over the documentation (refresher: http://seanmehan.globat.com/blog/2011/12/20/set-theory-and-sql-concepts/)? (http://seanmehan.globat.com/blog/2011/12/20/set-theory-and-sql-concepts/)?) The **set** data structure gives you the power to do inner/outer join-like activities.

In [ ]:
```
cityindex = headers.index('loc')

allcities = [] # creates and empty list to hold the all city names

for player in playerdata:
    allcities.append(player[cityindex])

print len(allcities) # just checking that we have the same number as we did for p
layers

print allcities[:10] # now checking that we have the same list of cities as we di
d last time
```

Now we can use the set tools to easily get a list of the unique values

When we have **allcities** populated with every city seen within the data, we can do several things.

If we cast it as a **set** object it will become a set of just the unique values (since a set cannot contain duplicates).

In [ ]:
```
unique_cities = set(allcities)

print len(unique_cities)
print unique_cities
```

So there are 134 unique cities in this data file. Neat! Maybe we also want to count them. Why not, right?

Python has a great tool (as discussed in the Python Cookbook: http://chimera.labs.oreilly.com/books/1230000000393/ch01.html#_determining_the_most_frequently_occurring_item (http://chimera.labs.oreilly.com/books/1230000000393/ch01.html#_determining_the_most_frequently_occurring_iter called **`Counter()`**

When you pass **`Counter`** a list, it will return a **`Counter`** object that contains the unique values and tallies for each. **`Counter`** objects can easily be recast back to a dictionary object for further manipulation. Let's look at it in action.

In [ ]:
```
from collections import Counter # you need to import it, and note the capitalization

city_count = Counter(allcities)

print city_count

# cool! now let's make it into a dictionary

city_count = dict(city_count)

# and as a byproduct we know that the keys are all unique

unique_cities = city_count.keys()

# .keys() actually does work on a Counter object, but you might need it to be a dict later

print 'Number of unique cities:',len(unique_cities)
print unique_cities
```

Finalizing the script

Now that we have all the cities, we need to make a choice about how to do the lookup. We don't want to hit the OSM service too many times, but the difference here might not be too bad considering the small volume (~400) cities total that we need to hit. This is a decision that you'll need to investigate and make depending on the size, volume, and time resources that you have with whatever service you are using.

For now, we'll just explore how to look up every value and add it to the data set.

To start, we need to plan out what we want to accomplish.

1. Read in the data.
2. Find the city.
3. Look the coordinates up with nominatim.
4. Add those coordinates to relevant row.
5. Write out the file with the new data.

Our goal is to preserve the current data and add the new data. We can start with the script that we already have in place for reading in the data. We'll then add pieces that we've developed in previous sections.

In [ ]:
```
import csv
from nominatim import Nominatim
import copy # we'll talk about this one in a bit!

# bring in the function we wrote for looking up the coordinates
def get_lat_lon(nom_results):
    if len(nom_results) == 0:
        lat = ''
        lon = ''
    else:
        first = nom_results[0]  #get the first result
        lat = first['lat']
        lon = first['lon']
    return {'lat': lat, 'lon': lon}

# instantiate the Nominatim object
nom = Nominatim()

root_folder = 'C:/Users/jvwhit/Documents/IPythonNotebooks/ILGISA/Data/CSV/' # n
eed this for working in the IDE

source = root_folder + 'JeopardyContestants_Illinois.csv'

# empty container to hold the player data
playerdata = []

# read in the file and populate the playerdata list
with open(source, 'r') as file_in:
    file_in = csv.reader(file_in)
    headers = next(file_in) # reads the first line
    for row in file_in: # the file 'curser' now starts at the second row
        playerdata.append(row)

print source, "is read."
# So we have the data!

cityindex = headers.index('loc')

# making another container to hold the new data

playerdata_withcoords = []

print "Getting coordinates."

for player in playerdata:
    p_with_coords = player[:] # empty container for the new line of data
    # the [:] notation is to make a deep copy
    city = player[cityindex]
    nom_results = nom.query(city)
    coords = get_lat_lon(nom_results)
    p_with_coords.append(coords['lat'])
    p_with_coords.append(coords['lon'])
    playerdata_withcoords.append(p_with_coords) # add our new row to the main s
tore

# should now look like
# ['3513', 'Umiko Post', '2002', 'Peoria, Illinois', '2', '0', 'illinois',
```

```
#          '', u'40.6938609', u'-89.5891008']

# now we need to make the new headers
print "Got all coordinates."
new_headers = headers[:] # again we need to do a deep copy here

# looking back to our code we added lat then lon

new_headers.append('lat')
new_headers.append('lon')

# so we have our headers and we have our data, time to write it out!
# note the 'wb' parameter. For windows, this is needed otherwise an extra line
break will be added

with open(root_folder + 'JeopardyContestants_LatLon.csv', 'wb') as file_out:
    file_out = csv.writer(file_out)
    file_out.writerow(new_headers)
    file_out.writerows(playerdata_withcoords)

print "CSV file is complete."
```

Looking at the final output of this script in Excel, we can see that our original data were preserved with just the lat and lon added at the end.

	1	2	3	4	5	6	7	8	9	10
1	ID	name	year	loc	numgames	totalwinning	state	studentflag	lat	lon
2	3513	Umiko Post	2002	Peoria, Illinoi	2	0	illinois		40.6938609	-89.589101
3	1170	Veronica Faz	2005	Roselle, Illinc	1	-1200	illinois		41.9847505	-88.079793
4	7186	Kristi Springe	2011	Naperville, Il	1	3600	illinois		41.7729107	-88.147867
5	7455	Edgar Miheli	2011	Chicago, Illin	1	8398	illinois		41.8755546	-87.624421
6	4728	Gretchen Wa	2008	Chicago, Illin	1	3399	illinois		41.8755546	-87.624421
7	2301	Bob Bearse	1988	Chicago, Illin	1	0	illinois		41.8755546	-87.624421
8	5847	Jennifer Pasc	2002	McHenry, Illi	1	0	illinois		42.3294391	-88.460571
9	5844	James Petho	2002	Chicago, Illin	2	25600	illinois		41.8755546	-87.624421
10	4623	Brandon Jon	2008	Chicago, Illin	1	22000	illinois		41.8755546	-87.624421
11	176	Kathleen Her	1998	Western Spri	1	2800	illinois		41.8097533	-87.900616
12	7218	Lindsey Thies	2011	Clarendon Hi	2	30802	illinois	TRUE	41.7975307	-87.954784
13	7215	Andrew Van	2011	Wheaton, Illi	1	6000	illinois	TRUE	41.8646959	-88.110171
14	7665	Taylor Cope	2012	Chicago, Illin	2	56000	illinois		41.8755546	-87.624421
15	1545	Susan Lee	1997	Glencoe, Illin	1	100	illinois		42.1350268	-87.758119
16	8712	Rich Hansen	2013	Chicago, Illin	1	9155	illinois		41.8755546	-87.624421
17	6770	Jessica Trude	2010	Barrington, Il	2	68802	illinois		42.1539141	-88.136189
18	2511	Matt Thomas	2006	Chicago, Illin	1	4399	illinois		41.8755546	-87.624421