4.3 Reading and parsing text using the Python csv module

Print

One of the best ways to increase your effectiveness as a GIS programmer is to learn how to manipulate text-based information. In Lesson 3, we talked about how to read data in ArcGIS's native formats, such as feature classes. But often GIS data is collected and shared in more "raw" formats such as a spreadsheet in CSV (comma-separated value) format, a list of coordinates in a text file, or an XML response received through a Web service.

When faced with these files, you should first understand if your GIS software already comes with a tool or script that can read or convert the data to a format it can use. If no tool or script exists, you'll need to do some programmatic work to read the file and separate out the pieces of text that you really need. This is called parsing the text.

For example, a Web service may return you many lines of XML describing all the readings at a weather station, when all you're really interested in are the coordinates of the weather station and the annual average temperature. Parsing the response involves writing some code to read through the lines and tags in the XML and isolating only those three values.

There are several different approaches to parsing. Usually the wisest is to see if some Python module exists that will examine the text for you and turn it into an object that you can then work with. In this lesson, you will work with the Python "csv" module that can read comma-delimited values and turn them into a Python list. Other helpful libraries such as this include lxml and xml.dom for parsing XML, and BeautifulSoup for parsing HTML.

If a module or library doesn't exist that fits your parsing needs, then you'll have to extract the information from the text yourself using Python's string manipulation methods. One of the most helpful ones is string.split(), which turns a big string into a list of littler strings based on some delimiting character, such as a space or comma. When you write your own parser, however, it's hard to anticipate all the exceptional cases you might run across. For example, sometimes a comma-separate value file might have substrings that naturally contain commas, such as dates or addresses. In these cases, splitting the string using a simple comma as the delimiter is not sufficient and you need to add extra logic.

Another pitfall when parsing is the use of "magic numbers" to slice off a particular number of characters in a string, to refer to a specific column number in a spreadsheet, and so on. If the structure of the data changes, or if the script is applied to data with a slightly different structure, the code could be rendered inoperable and would require some precision surgery to fix. People who read your code and see a number other than 0 (to begin a series) or 1 (to increment a counter) will often be left wondering how the number was derived and what it refers to. In programming, numbers other than 0 or 1 are magic numbers that should typically be avoided, or at least accompanied by a comment explaining what the number refers to.

There are an infinite number of parsing scenarios that you can encounter. This lesson will attempt to teach you the general approach by walking through just one module and example. In your final project for this course, you may choose to explore parsing other types of files.

The Python csv module

A common text-based data interchange format is the comma-separated value (CSV) file. This is often used when transferring spreadsheets or other tabular data. Each line in the file represents a row of the dataset, and the columns in

the data are separated by commas. The file often begins with a header line containing all the field names.

Spreadsheet programs like Microsoft Excel can understand the CSV structure and display all the values in a row-column grid. A CSV file may look a little messier when you open it in a text editor, but it can be helpful to always continue thinking of it as a grid structure. If you had a Python list of rows and a Python list of column values for each row, you could use looping logic to pull out any value you needed. This is exactly what the Python csv module gives you.

It's easiest to learn about the csv module by looking at a real example. The scenario below shows how the csv module can be used to parse information out of a GPS track file.

Introducing the GPS track parsing example

This example reads a text file collected from a GPS unit. The lines in the file represent readings taken from the GPS unit as the user traveled along a path. In this section of the lesson, you'll learn one way to parse out the coordinates from each reading. The next section of the lesson uses a variation of this example to show how you could write the user's track to a polyline feature class.

The file for this example is called gps_track.txt and it looks something like the text string shown below. (Please note, line breaks have been added to the file shown below to ensure that the text fits within the page margins. Click on this link to the gps track.txt file to see what the text file actually looks like.)

```
type,ident,lat,long,y_proj,x_proj,new_seg,display,color,altitude,dept
h,temp,time,model,filename,ltime
TRACK,ACTIVE LOG,40.78966141,-
77.85948515,4627251.76270444,1779451.21349775,True,False,
    255,358.228393554688,0,0,2008/06/11-14:08:30,eTrex Venture,
,2008/06/11 09:08:30
TRACK,ACTIVE LOG,40.78963995,-
77.85954952,4627248.40489401,1779446.18060893,False,False,
    255,358.228393554688,0,0,2008/06/11-14:09:43,eTrex Venture,
,2008/06/11 09:09:43
TRACK,ACTIVE LOG,40.78961849,-
77.85957098,4627245.69008772,1779444.78476531,False,False,
    255,357.747802734375,0,0,2008/06/11-14:09:44,eTrex Venture,
,2008/06/11 09:09:44
TRACK,ACTIVE LOG,40.78953266,-
77.85965681,4627234.83213242,1779439.20202706,False,False,
    255,353.421875,0,0,2008/06/11-14:10:18,eTrex Venture, ,2008/06/11
09:10:18
TRACK,ACTIVE LOG,40.78957558,-
77.85972118,4627238.65402635,1779432.89982442,False,False,
    255,356.786376953125,0,0,2008/06/11-14:11:57,eTrex Venture,
,2008/06/11 09:11:57
TRACK,ACTIVE LOG,40.78968287,-
77.85976410,4627249.97592111,1779427.14663093,False,False,
    255,354.383178710938,0,0,2008/06/11-14:12:18,eTrex Venture,
,2008/06/11 09:12:18
TRACK,ACTIVE LOG,40.78979015,-
77.85961390,4627264.19055204,1779437.76243578,False,False,
    255,351.499145507813,0,0,2008/06/11-14:12:50,eTrex Venture,
,2008/06/11 09:12:50
etc. ...
```

Notice that the file starts with a header line, explaining the meaning of the values contained in the readings from the GPS unit. Each subsequent line contains one reading. The goal for this example is to create a Python list containing the X,Y coordinates from each reading. Specifically, the script should be able to read the above file and print a text string like the one shown below.

```
[['-77.85948515', '40.78966141'], ['-77.85954952', '40.78963995'],
['-77.85957098', '40.78961849'], etc.]
```

Approach for parsing the GPS track

Before you start parsing a file, it's helpful to outline what you're going to do and

break up the task into manageable chunks. Here's some pseudocode for the approach we'll take in this example:

1. Open the file.
2. Read the header line.
3. Loop through the header line to find the index positions of the "lat" and "long" values.
4. Read the rest of the lines one by one.
5. Find the values in the list that correspond to the lat and long coordinates and write them to a new list.

Importing the module

When you work with the csv module, you need to explicitly import it at the top of your script, just like you do with arcpy.

```
import csv
```

You don't have to install anything special to get the csv module; it just comes with the base Python installation.

Opening the file and creating the CSV reader

The first thing the script needs to do is open the file. Python contains a built-in open() method for doing this. The parameters for this method are the path to the file and the mode in which you want to open the file (read, write, etc.). In this example, "r" stands for read-only mode. If you wanted to write items to the file, you would use "w" as the mode.

```
gpsTrack = open("C:\\data\\Geog485\\gps_track.txt", "r")
```

Notice that your file does not need to have the extension .csv in order to be read by the CSV module. It can be suffixed .txt as long as the text in the file conforms to the CSV pattern where commas separate the columns and a carriage returns separate the rows. Once the file is open, you create a CSV reader object, in this manner:

```
csvReader = csv.reader(gpsTrack)
```

This object is kind of like a cursor. You can use the next() method to go to the next line, but you can also use it with a for loop to iterate through all the lines of the file.

Reading the header line

The header line of a CSV file is different from the other lines. It gets you the information about all the field names. Therefore, you will examine this line a little differently than the other lines. First, you advance the CSV reader to the header line by using the next() method, like this:

```
header = csvReader.next()
```

This gives you back a Python list of each item in the header. Remember that the header was a pretty long string beginning with: "type,ident,lat,long...". The CSV reader breaks the header up into a list of parts that can be referenced by an index number. The default delimiter, or separating character, for these parts is the comma. Therefore, header[0] would have the value "type", header[1] would have the value "ident", and so on.

We are most interested in pulling latitude and longitude values out of this file, therefore, we're going to have to take note of the position of the "lat" and "long" columns in this file. Using the logic above, you would use header[2] to get "lat" and header[3] to get "long". However, what if you got some other file where these field names were all in a different order? You could not be sure that the column with index 2 represented "lat" and so on.

A safer way to parse is to use the list.index() method and ask the list to give you the index position corresponding to a particular field name, like this:

```
latIndex = header.index("lat")
```

```
lonIndex = header.index("long")
```

In our case, latIndex would have a value of 2 and lonIndex would have a value of 3, but our code is now flexible enough to handle those columns in other positions.

Processing the rest of the lines in the file

The rest of the file can be read using a loop. In this case, you treat the csvReader as an iterable list of the remaining lines in the file. Each run of the loop takes a row and breaks it into a Python list of values. If we get the value with index 2 (represented by the variable latIndex), then we have the latitude. If we get the value with index 3 (represented by the variable lonIndex), then we get the longitude. Once we get these values, we can add them to a list we made, called coordList:

```
# Make an empty list
coordList = []

# Loop through the lines in the file and get each coordinate
for row in csvReader:
    lat = row[latIndex]
    lon = row[lonIndex]
    coordList.append([lat,lon])

# Print the coordinate list
print coordList
```

Note a few important things about the above code:

- coordList actually contains a bunch of small lists within a big list. Each small list is a coordinate pair representing the x (longitude) and y (latitude) location of one GPS reading.
- The list.append() method is used to add items to coordList. Notice again that you can append a list itself (representing the coordinate pair) using this method.

Full code for the example

Here's the full code for the example. Feel free to download the text file and try it out on your computer.

```
# This script reads a GPS track in CSV format and
#  prints a list of coordinate pairs
import csv

# Set up input and output variables for the script
gpsTrack = open("C:\\data\\Geog485\\gps_track.txt", "r")

# Set up CSV reader and process the header
csvReader = csv.reader(gpsTrack)
header = csvReader.next()
latIndex = header.index("lat")
lonIndex = header.index("long")

# Make an empty list
coordList = []

# Loop through the lines in the file and get each coordinate
for row in csvReader:
    lat = row[latIndex]
    lon = row[lonIndex]
    coordList.append([lat,lon])

# Print the coordinate list
print coordList
```

Applications of this script

You might be asking at this point, "What good does this list of coordinates do for me?" Admittedly, the data is still very "raw." It cannot be read directly in this state by a GIS. However, having the coordinates in a Python list makes them easy to get into other formats that can be visualized. For example, these coordinates could be written to points in a feature class, or vertices in a polyline or polygon feature class. The list of points could also be sent to a Web service for reverse geocoding, or finding the address associated with each point. The points could also be plotted on top of a Web map using programming tools like the Google Maps API. Or, if you were feeling really ambitious, you might use Python to write a new file in KML format, which could be viewed in 3D in Google

Earth.

Summary

Parsing any piece of text requires you to be familiar with file opening and reading methods, the structure of the text you're going to parse, the available parsing modules that fit your text structure, and string manipulation methods. In the preceding example, we parsed a simple text file, extracting coordinates collected by a handheld GPS unit. We used the csv module to break up each GPS reading and find the latitude and longitude values. In the next section of the lesson, you'll learn how you could do more with this information by writing the coordinates to a polyline dataset.

As you use Python in your GIS work, you could encounter a variety of parsing tasks. As you approach these, don't be afraid to seek help from Internet examples, code reference topics such as the ones linked to in this lesson, and your textbook.

Readings

It's worth your time to read Zandbergen 7.6, which talks about parsing text files. Any examples you can pick up with text parsing will help you when you encounter a new file that you need to read. You'll have this experience in the practice exercises and projects this week.

‹ 4.2 Python Dictionaries up 4.4 Writing geometries ›

Author(s) and/or Instructor(s): Sterling Quinn, John A. Dutton e-Education Institute, College of Earth and Mineral Sciences, The Pennsylvania State University;
Jim Detwiler, John A. Dutton e-Education Institute, College of Earth and Mineral Sciences, The Pennsylvania State University;
Frank Hardisty, John A. Dutton e-Education Institute, College of Earth and Mineral Sciences, The Pennsylvania State University;
James O'Brien, John A. Dutton e-Education Institute, College of Earth and Mineral Sciences, The Pennsylvania State University