All About Files

Lesson 3 (Haddock and Dunn Chapter 10, modified)

In this lesson we will go over handling different file formats, how to convert between different formats, and parse through data. Comments in blue will indicate code written in the text editor and comments in brown will indicate commands used in the terminal (bold indicates commands you need to type in to the terminal).

Before you open a data file, it is always a good idea to make a plan. Determine your goal and make an outline of what you want to do and the end result you wish to achieve. This can help to prevent small errors that you may experience in the future. Look at the data in the file you are working with and make sure it is in the format that you expect it to be in. You can read in an entire file in Python or you can choose to read it in line by line. First, you need to know if your data are in single lines or are separated across multiple lines (which is very common with sequencing data). Next, you need to know if the desired outputs are dependent on single lines or multiple lines, for example, if you are calculating a running average, each line is dependent on the next.

Reading in a text file

We will use an example file that contains data for a single item on each line. Copy this into a text editor and remember to turn on 'show invisibles' to make sure there are no extra spaces or tabs. Spaces are shown by a dot and spaces are shown by a triangle. Save the file as `filepractice.txt`.

```
Dive Date Lat  Lon  Depth     Notes
Tiburon 596    19-Jul-03 36 36.12 N     122 22.48 W    1190
     holotype
JSL II 1411    16-Sep-86 39 56.4 N 70 14.3 W 518  paratype
JSL II 930     18-Aug-84 40 05.03 N     69 03.01 W     686
     Youngbluth (1989)
Ventana 1575   11-Mar-99 36 42.24 N     122 02.52 W    767
Ventana 1777   16-Jun-00 36 42.60 N     122 02.70 W    934
Ventana 2243   9-Sep-02  36 42.48 N     122 03.84 W    1001
Tiburon 515    24-Nov-02 36 42.00 N     122 01.98 W    1156
Tiburon 531    13-Mar-03 24 19.02 N     109 12.18 W    1144
Tiburon 547    31-Mar-03 24 14.04 N     109 40.02 W    1126
JSL II 3457    26-Sep-03 40 17.77 N     68 06.68 W     862
     Francesc Pages (pers.comm)
```

The simple script below will show you how to open the file, read and count each line, and close the file. Save the file as `latlon.py`.

```
#! /usr/bin/env python    ← Specifies how to run the script

InFileName = "filepractice.txt"

InFile = open(InFileName, 'r')

LineNumber = 0

for Line in InFile:
     Line = Line.strip('\n')
     print (LineNumber, ":", Line)

     LineNumber = LineNumber + 1

InFile.close()
```

To run this script, you need to make it executable using `chmod` (see lesson 2). In the terminal, `cd` to the directory that contains the file you wish to open (`"filepractice.txt"`). Call the script file, `latlon.py`, within that directory and each line should be printed out with a line number.

Note: The header line (the first line) will begin with 0.

After the shebang, we first name the file we want to read in, in this case `InFileName`. The variable `InFileName` now contains the path to the file we wish to open. The next line opens the file and saves it in the variable `InFile`. The `'r'` indicates that this file is for reading. This is the default mode but you can search for `open` on the Python docs website (https://docs.python.org/3/) to learn about other options for opening files. Next, we need to tell Python that we want the variable `LineNumber` to be an integer and begin at `0`.

The `for` loop then removes the end of line character and prints each line with its respective line number. The method `.strip` removes the first and last specified characters from a string. Here, we removed the end of line character denoted by `'\n'`. If you don't specify a character to be removed, all the spaces and tabs will be removed from the front and end of each line. Windows users will need to include `'\r'`as well as `'\n'`. It is recommended to use both `'\r'` and `'\n'` if you are reading in files from different sources. This will make sure the end of line characters are removed.

The command that adds `1` to each line number (`LineNumber = LineNumber + 1`) is beneath the print statement so the first line that contains the headers of each column is counted as line `0` and not `1`. The last line then closes the file. The `.close()` method can be used on the `InFile` variable because it was opened using the function `open()`.

After running this script you will see that each line is printed out on a separate line. This is because the print statement adds an end of line character when each line is returned.

When analysing data in this file, we will not want to use the header row. Since this row is line `0` we can add an `if` statement to specify that we only want lines that are `1` or greater.

```
#! /usr/bin/env python

InFileName = "filepractice.txt"

InFile = open(InFileName, 'r')

LineNumber = 0

for Line in InFile:
     if LineNumber > 0:      ← Don't forget the colon!
          Line = Line.strip('\n')
          print (LineNumber, ":", Line)
     LineNumber = LineNumber + 1

InFile.close()
```

Note: Watch the changes in indentation. Be sure to keep the line '`LineNumber = LineNumber + 1`' one indentation in from the `for` loop so it is not executed within the `if` statement.

Parsing data

Using the `.split()` method, you can split a string at a specified location and create a list of strings. Like `.strip()`, `.split()` will split using white spaces by default. In this case, we will split our data by tabs. Edit your script as follows to put each line into a list:

```
#! /usr/bin/env python

InFileName = "filepractice.txt"

InFile = open(InFileName, 'r')
```

```
LineNumber = 0

for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')
          print (LineNumber, ':', ElementList) ← Change the print statement
     LineNumber = LineNumber + 1

InFile.close()
```

The variable `ElementList` refers to each line. Now each line is its own list:

```
1 : ['Tiburon 596', '19-Jul-03', '36 36.12 N', '122 22.48 W',
     '1190', 'holotype']
2 : ['JSL II 1411', '16-Sep-86', '39 56.4 N', '70 14.3 W',
'518',
     'paratype']
… Lines omitted
9 : ['Tiburon 547', '31-Mar-03', '24 14.04 N', '109 40.02 W',
     '1126', '']
10 : ['JSL II 3457', '26-Sep-03', '40 17.77 N', '68 06.68 W',
     '862', 'Francesc Pages (pers.comm)']
```

Due to the format of this data, if we had not specified tabs as the break parameter and had instead left the `.split()` method blank, some of the pieces of data would be further split within the lists (for example, latitude and longitude would be split because they contain spaces). Also, see the empty single quotes at the end of list `9`. The last column in this file is a notes column, which is common with data sets. Be sure to be aware of this and make sure your program is dealing with these empty spaces correctly.

Selecting elements from lists

Remember that objects in lists start with 0, not 1. We can print out specific values using the square brackets (see lesson 2). We can add a print statement within the `for` loop that returns specific values.

```
#! /usr/bin/env python

InFileName = "filepractice.txt"

InFile = open(InFileName, 'r')

LineNumber = 0
```

```
for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')
          print (LineNumber, ':', ElementList)
          print ("Depth: %s\tLat: %s\tLon: %s" % \
(ElementList[4], ElementList[2], ElementList[3]))

     LineNumber = LineNumber + 1

InFile.close()
```

Here, we included another print statement. The statement uses the `%` symbol which we went over in lesson 1. Also, see the added a backslash that breaks the line in two. You can use this to escape long lines of code. When using the backslash you do not need to worry about the indentation. Python reads this as a single line of code so it will interpret it at the indentation at which it began. The `s` after the `%` specifies that the objects to be placed there are strings.

Creating files

So far, all of the outputs we have generated have been printed directly on the terminal window. Often, you will want to redirect the outputs of a script to a new file. Copy the contents of `latlon.py` and paste them into an empty text file. Save this new script as `latlon2.py`. We will be making edits to this script but you may want to save the original.

First, change the name of the file we want to send out, not in. Make sure there is no other file (or at least no other file that you care about) with this name because it will be overwritten. Also, we will need to change the mode for opening the file. The previous script used the reading mode, `'r'`, however, this time we want to use `'w'`, specifying that we are writing data to the file named above. You can add data to a file if you use `'a'`, which stands for append.

Example of code needed to write to a file instead of just opening a file:

```
#! /usr/bin/env python

OutFileName = "filepracticeoutput.kml"
OutFile = open(OutFileName, 'w')
```

These commands now allow us to write to the new (or old, depending on what are you doing) file using `.write()`. This method is written into the variable `OutFile` because it was used with the `open()` function. The following script collects the depth, latitude, and longitude from each line and saves it to a different file (`filepracticeoutput.kml`). Watch for the small changes from the original script.

```
#! /usr/bin/env python

InFileName = "filepractice.txt"

InFile = open(InFileName, 'r')

LineNumber = 0

OutFileName = "filepracticeoutput.kml"

OutFile = open(OutFileName, 'w')


for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')
          OutputString = "Depth: %s\tLat: %s\tLon: %s" % \
(ElementList[4],ElementList[2], ElementList[3])
          print (OutputString)
          OutFile.write(OutputString + "\n")
     LineNumber = LineNumber + 1


InFile.close()
OutFile.close()
```

When using the `.write()` method, notice that there is an added `"\n"`. This is so each piece of data is on a separate line and not in one long line. Print statements add end of line characters, however, the `.write()` method does not. Also, you need to close both files at the end of the script.

Parsing using regular expressions

In this example, we want to convert our file into a KML (Keyhole Markup Language) file so it can be used with Google Earth. First, we need to change the latitude and longitude formats to decimals so it can be properly read by Google Earth. We can use regular expressions to capture the data we want and convert it to the format needed.

We need to import the `re` module into Python, which is a module that will allow Python to understand regular expressions. Into the script, add under the shebang line:

```
import re
```

We will use the `.search()` method to specify what we want to search and which string to search in. First, return the latitude or longitude item from the string to see the structure so we know what we need to modify.

```
"36 36.12 N"
```

There are 3 different sections. The first, one or more digit; second, one or more digits or a decimal point; and third, a letter. Also, there is a space between the three elements. So in regular expression format it would look as follows:

```
\d+ [\d\.]+ \w
```

Using the `.search()` method the command would look like this:

```
SearchStr = '(\d+) ([\d\.]+) (\w)' ← Capture the search terms
Result = re.search(SearchStr, ElementList[2])
```

You can put the search term directly into the `.search()` statement instead of creating a new variable if you prefer. In the example above, the results of the search term are stored in groups within the variable we named `Result`. You can retrieve these groups individually or all at once. Using the method `.group()` you can specify which group you would like to see. In this case we have groups 1, 2, and 3. To see all groups you can use the method `.groups()` or you can use `.group(0)`. Next we will separate each group. The first two groups will be converted into a floating point type and the third group will be specified as uppercase.

```
Degrees = float(Result.group(1))
Minutes = float(Result.group(2))
Compass = Result.group(3).upper()
```

Now, we need to include an `if` statement for south or west measurements and convert them to negative numbers.

```
DecimalDegree = Degrees + Minutes/60

if Compass == 'S' or Compass =='W':
     DecimalDegree = -DecimalDegree
```

Altogether this section of the script should look as follows:

```
SearchStr = '(\d+) ([\d\.]+) (\w)'
Result = re.search(SearchStr, ElementList[2])

Degrees = float(Result.group(1))
Minutes = float(Result.group(2))
Compass = Result.group(3).upper()

DecimalDegree = Degrees + Minutes/60

if Compass == 'S' or Compass =='W':
     DecimalDegree = -DecimalDegree
```

Similar to `.search()`, you can use `.sub()` to rearrange the order or structure of the regular expression. For example:

```
>>> import re
>>> OriginalString = "Wilde,Oscar" ← Last name, first name
>>> Find = r"(\w+),(\w+)"   ← Search term captures two sections of one or more word
character
>>> Result = re.search(Find, OriginalString)
>>> print (Result.groups())
('Wilde', 'Oscar')
>>> Replace = r"\2\t\1"   ← Reordering search terms
>>> NewString = re.sub(Find, Replace, OriginalString)
>>> print (NewString)
Oscar     Wilde   ← First name, last name
```

Creating functions in Python

Using loops is a common and useful way to reduce the amount of code used in a script. Another way to reduce the amount of code needed and to make a script cleaner and easier to read is to create your own function that can be used over and over again. In this example, we want to convert both latitude and longitude into a different format. We will use `def` to create a function called `latlon()` so we can easily convert both latitude and longitude with a single function. To create the function, place the parameters that you want to be sent to the function inside the parentheses, followed by a colon. Since we have already written the code needed to convert the latitude and longitude format, few changes and additions are needed.

```
import re

def latlon(DegString):
     SearchStr = '(\d+) ([\d\.]+) (\w)'
     Result = re.search(SearchStr, DegString) ← Change string name here
     Degrees = float(Result.group(1))
     Minutes = float(Result.group(2))
     Compass = Result.group(3).upper()

     DecimalDegree = Degrees + Minutes/60

     if Compass == 'S' or Compass =='W':
          DecimalDegree = -DecimalDegree

     return DecimalDegree
```

Last, we need to include the new function in the `for` loop and name the variables we want from `ElementList`.

```
for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')

          Dive = ElementList[0]
          Date = ElementList[1]
          Depth = ElementList[4]
          Comment = ElementList[5]

          LatDegrees = latlon(ElementList[2]) ← Saving into new variable
          LonDegrees = latlon(ElementList[3]) ← Saving into new variable
          print ("Lat: %f, Lon: %f" % (LatDegrees, LonDegrees))
```

In this kind of scenario, it is a good idea to double check that the latitude and longitude are under the correct labels. It would be easy to swap their labels since they have the same format. All of these pieces can now be put together to make a fun script that reads in a dataset and creates a new file with specific converted data items. The entire script should look as follows. Look for the small changes and additions within the script and try to determine what they are doing.

```
#! /usr/bin/env python

import re

def latlon(DegString):
     SearchStr = '(\d+) ([\d\.]+) (\w)'
     Result = re.search(SearchStr, DegString)

     Degrees = float(Result.group(1))
     Minutes = float(Result.group(2))
     Compass = Result.group(3).upper()

     DecimalDegree = Degrees + Minutes/60

     if Compass == 'S' or Compass =='W':
          DecimalDegree = -DecimalDegree

     return DecimalDegree


InFileName = "filepractice.txt"
OutFileName = 'New_' + InFileName   ← Change file name

InFile = open(InFileName, 'r')
HeaderLine = 'dive\tdepth\tlatitude\tlongitude\tdate\tcomment'
print (HeaderLine)

OutFile = open(OutFileName, 'w')
OutFile.write(HeaderLine + '\n')

LineNumber = 0
for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')

          Dive = ElementList[0]
          Date = ElementList[1]
          Depth = ElementList[4]
          Comment = ElementList[5]

          LatDegrees = latlon(ElementList[2])
          LonDegrees = latlon(ElementList[3])
          print ("Lat: %f, Lon: %f" % (LatDegrees, LonDegrees))

          OutputString = "%s\t%4s\t%10.5f\t%10.5f\t%9s\t%s" % \
          (Dive, Depth, LatDegrees, LonDegrees, Date, Comment)

          print (OutputString)
          OutFile.write(OutputString + "\n")

     LineNumber += 1   ← Shorthand way of increasing

InFile.close()
OutFile.close()
```

Reformatting data

Now we will reformat the file so it can be interpreted by Google Earth as a KML file. Markup languages use special tags to denote items in a document. Remember the source code we obtained from the internet for the amino acid table in lesson 2? That was HTML, which is the markup language for websites. Each line was encompassed by some characters within the symbols < >. In markup languages, each element is specified using these tags. For example:

```
<recipe>
     <title>Cookie</title>
     <ingredients>flour, milk, eggs, chocolate</ingredients>
     <bakedegree>350C</bakedegree>
     <baketime>1.5hr</baketime>
</recipe>
```

Each item must begin and end with the tag; the last tag including a forward slash /. Like using brackets and parentheses, all elements must be closed and you can nest elements within others. Although we will not use all the data from the current file, it is recommended to convert all of this data to the new file type. In the future you may want to use the converted file with the data you previously did not use. In this case, it would cause more difficulty had you removed that data from the converted file. There are times when you won't want to do this, however, so try to plan out what you may do with the data in the future and make the best decision you can.

Converting to KML

In KML files, there are three main sections: the header, the data, and the footer. Similar to the shebang line that we have used at the beginning of all our Python scripts, the header specifies the formatting of the data, what it is to be converted to, and the opening tag for the file. The first line (the line that specifies the formatting) does not need to be closed but the other two lines will be closed at the end of the document in the footer.

Header:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
```

Next is the data. Like described above, each element must be surrounded by tags.

Example data:

```
<Data>
     <name>Tiburon 596</name>
     <description>Tiburon 596 19-Jul-03 36 36.12 N      122 \
22.48W    1190 holotype</description>
```

```
     <point>
          <altitudeMode>absolute</altitudeMode>
          <coords>-122.374667, 36.602000, -1190</coords>
     </point>
</Data>
```

The footer simply closes the two open tags from the header.

Footer:

```
</Document>
</kml>
```

Now we can change the `latlon2.py` script to create the header, data, and footer strings needed. Make sure to keep the header and footer outside of the `for` loop since there is only one occurrence of them but the data string will be placed within the loop. To generate the data string, which is the most complex, we can use triple single quotes (`'''`) around the entire block of code. In this way, the string can be spread across multiple lines.

```
DataString = '''
<Data>
     <name>Data - %s</name>
     <description>%s</description>
     <point>
          <altitudeMode>absolute</altitudeMode>
          <coords>%f, %f, -%s</coords>
     </point>
</Data> ''' % (Dive, Line, LonDegrees, LatDegrees, Depth)
```

Note: A negative symbol is added to the depth element because Google Earth requires depth measurements to be negative.

The entire script should look as follows:

```
#! /usr/bin/env python

import re

def latlon(DegString):
     SearchStr = '(\d+) ([\d\.]+) (\w)'
     Result = re.search(SearchStr, DegString)

     Degrees = float(Result.group(1))
     Minutes = float(Result.group(2))
     Compass = Result.group(3).upper()

     DecimalDegree = Degrees + Minutes/60

     if Compass == 'S' or Compass =='W':
          DecimalDegree = -DecimalDegree

     return DecimalDegree


InFileName = "filepractice.txt"
OutFileName = "PracticeKMLfile.kml"   ← Change name of new file

InFile = open(InFileName, 'r')

HeaderString = '''<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>'''

OutFile = open(OutFileName, 'w')
OutFile.write(HeaderString)         ← Removed '\n'

LineNumber = 0
for Line in InFile:
     if LineNumber > 0:
          Line = Line.strip('\n')
          ElementList = Line.split('\t')

          Dive = ElementList[0]
          Date = ElementList[1]
          Depth = ElementList[4]
          Comment = ElementList[5]

          LatDegrees = latlon(ElementList[2])
          LonDegrees = latlon(ElementList[3])

          PlacemarkString = ''' ← Keep indent in line with if statement
<Placemark>                       ← Indentation for the rest of the command does not
matter
 <name>Data - %s</name>
 <description>%s</description>
 <Point>
  <altitudeMode>absolute</altitudeMode>
  <coordinates>%f, %f, -%s</coordinates>
 </Point>
</Placemark>''' % (Dive, Line, LonDegrees, LatDegrees, Depth)
```

```
          OutFile.write(PlacemarkString)

     LineNumber += 1

InFile.close()

print ("Saved", LineNumber, "records from", InFileName, "as",
     OutFileName)
OutFile.write('\n</Document>\n</kml>\n')
OutFile.close()
```

On Google Earth, under the My Places tab you can upload your KML file and the data points will appear!