Python File IO

November 4, 2015

1 Overview

A language that only accepts user input is basically useless. No one is going to hand enter shapefile data each time they run a program. To be useful a language needs to do more than read and write console data; it must be able to deal with files.

This topic covers files and "file-like" objects that can be accessed with similar methods (e.g., scraping web pages, pipes from other commands).

Python is heavily oriented toward text files, even those opened as "binary". File read/write methods are not part of the set of base functions in Python. Modules are needed to read or write files containing binary numeric data such as floats. Those modules won't be covered in this course because you are more likely to use GDAL, OGR, or ArcPY to open a GIS file than you are to write your own program using a binary module.

In addition to the libraries mentioned above, there are other modules for reading and writing GIS data, such as `Pyshp` for shapefiles. The GIS modules for Python provide GIS capability that go far beyond simply reading and writing data. There are also Python modules for dealing with non-GIS files, such as Excel spreadsheets and MS Word files. Before you start writing your own functions to read files make sure you look at the available modules in Python.

2 Basic Text File Processing

Like most languages, Python adopts a book metaphor:

```
Open
Read and/or write
Close (does not mean delete!)
```

The default for file access is sequential access. Python begins reading a file at the first byte and reads every byte in sequence. But we can move the cursor (known as the read/write "head") to arbitrary positions within a file. This is known as "random access", which is provided in many other programming languages. In this case "random" simply means we can go wherever we want within a file. It does not mean random in a statistical sense, because that would be pretty much useless.

`open` function: `open` is a built-in Python function that must be called before reading or writing data. It returns a file object with methods for reading, writing, etc.

Open function Syntax:

```
In [ ]: f = open(fileName,mode, buffering)
```

| `fileName` | required string containing the name of existing or new file (required) | | `mode` | optional string indicating how the file will be processed (defaults to 'rt') | | `buffering` | optional integer requesting special buffering (advanced use) |

2.0.1 Examples:

```
In [ ]: f = open('ClassRoster.txt', 'rt')
```

Open ClassRoster.txt as a text file for reading only. The file must exist. Trying to write the file throws an error because the `w` (write) option is not selected.

```
In [ ]: f = open('ClassRoster.txt', 'rb')
```

Open `ClassRoster.txt` as a binary file for reading only. The file must exist. (We'll cover binary files later)

f = open('MadisonDEM.asc', 'wt')

Open MadisonDEM.asc as a text file for writing. If the file does not exist, it is created. If it exists, it will be overwritten without prompting, so be very careful with this option. You may want to use a check before you try it.

```
In [ ]: f = open(r'c:\tmp\Tempfile.txt', 'r+t')
```

Open `c:\tmp\Tempfile.txt` as a text file for reading and writing. The file must exist.

```
In [ ]: f = open('Timetable.csv', 'w+t')
```

Open `Timtetable.csv` as a text file for reading and writing. The file need not exist.

```
In [ ]: f = open("Timetable.csv", 'at')
```

Open `Timetable.csv` as a text file for appending. The file need not exist. All data will be written at end of file.

Assuming success, `f` will be a file object with the following properties (remember you can use `dir(f)` to see all the methods available for a file object).

f.name #name as supplied to open() f.mode #mode as supplied or default ("r") f.closed #False until f.close() is called

2.1 Example 10.1:

```
In [2]: #  This should point to a file in the txtFiles directory of your data folder.
        #  Use the explicit path, or copy the file into the same directory as your Python script

        tFile = open("ClassRoster.txt",'r+t')

        print '  file name:',tFile.name
        print '  file mode:',tFile.mode
        print 'file closed:',tFile.closed
        print type(tFile)
```

```
file name: ClassRoster.txt
  file mode: r+t
file closed: False
<type 'file'>
```

Other Python functions also return `file` objects. `file` objects all have the same methods and behavior because `file` is a defined class in Python.

2.2 Reading a file:

The main method, or at least, one of the more commonly used methods is using a for `loop` to read every line of a file:

```
for line in file:
statement1
statement2
...
StatementN
```

Note : `file` in this case is the variable name assigned to a file object (not a file name!!), so you need to `open` the file first.

2.3 Example 10.2

Open a file and print every line.

```
In [3]: classFile = open("ClassRoster.txt")
        for student in classFile:
           print student

Abel, John: Geography

Lewinski, Ralph: Communication Arts

Lassen, Bella: Environmental Studies

Poppich, Bruce: Geography

Sternobli, Cassandra: Undeclared

Tavlin, Robin: Geography
```

Why is the output double-spaced with empty lines? Ans: The file has 6 lines, each ending in a newline (n). The string variable student in the program contains the newline. The entire string gets printed (including the newline), and the Python print statement adds a newline after that.

3 Exercise 10.1:

Modify the program so no empty lines appear in the output. [Hint: Remember the strip() method, see the notes on Python strings.]

Go to the Exercise 10.1. PDF on Learn@UW

3.1 Closing a file

Any open file will be closed when program terminates (normally or not). So why close a file? Reasons to close a file:

1. Frees resources: Python knows we are done with the file object, and so it can be deleted. (The file object is delete, not the external data file.)
2. Make it clear to people using the code (and other programs that may want to use the code) that program is finished with the file
3. If the file is used for writing the buffer is flushed, and that means your data are safe even if the program crashes later on.

This last point is important. Python doesn't write every byte to file when told to do so. For efficiency's sake is stores a group of bytes in an internal buffer and writes the batch all at once. We have no control over when that happens, but closing a file guarantees the buffer is emptied.

How to close a file? It couldn't be simpler:

```
In [ ]: f.close()
```

After `f.close()` attempts to read or write f are errors until you `open` a new file. It is good practice to close as file when you are done with it.

3.2 Example 10.3

We have a file (`LandTemp.txt` in your data folder's `txtFiles` folder) that contains the average winter temperature for continental locations on a 1° x 1° grid in longitude and latitude. You can look at the file directly, but a sample is included here:

```
In [ ]: -180.00    71.00   -11.90
        -180.00    68.00   -12.70
        -180.00    67.00   -11.60
        -180.00    66.00    -6.50
        -180.00    65.00    -5.70
        -180.00   -85.00   -36.20
        -180.00   -86.00   -42.00
        -180.00   -87.00   -46.60
        -180.00   -88.00   -48.90
        -180.00   -89.00   -50.00
        -179.00    71.00   -12.10
        -179.00    68.00   -10.50
```

All continental values for -180oW are shown above. As you know, the dateline is mostly ocean, so no surprise there. The task is to prompt the user for a location and either print the temperature or report that the place isn't on land.

Pseudo-code:

1. Read temperatures T for all locations and store in a dictionary using (lon,lat) as the dictionary key
2. Prompt the user for a longlat pair (allowing an empty string to exit)
3. Convert the input longlat to floats and round the value to the nearest degree
4. Pack the longlat pair into a tuple
5. If the longlat pair is in the dictionary then print the temperature, otherwise show message
6. Repeat steps 2 to 5

Step 1 is too complicated to easily break down, or at least, it will take multiple sub-steps:

1. Read temperatures T for all locations and store in a dictionary using (lon,lat) as the dictionary key:
2. Initialize line counter
3. For every line in file:
    1. Split by blanks into 3 fields: long, lat and T
    2. Convert longlat, and temperature to a float
    3. Store longlat as a tuple
    4. Add entry to dictionary where the tuple is the key and the temperature is the value, so we can call `d[tuple]` to recover the value.
    5. increment line counter
4. Print number of lines read
5. Prompt user for lon,lat or empty string to exit
6. Convert input lon,lat to floats and round to nearest degree

7. Pack (lon,lat) in tuple
8. If (lon,lat) is in dictionary, print the temperature, otherwise show message
9. Repeat steps 2 to 5

```
In [ ]: # Initialize the dictionary to hold longitude,latitude, and temperature:
        lonlatT = {}

        # Initialize the counter for the dataset:
        npts = 0

        # Open the data file:
        f = open("LandTemp.txt","rt")

        for line in f:
           # For each line in the file, split the line at 'space' characters:
           fields = line.split()

           # We know the file structure, it is `lon   lat   T`. Convert these to floats:
           #lon = float(fields[0])
           #lat = float(fields[1])
           #T   = float(fields[2])

           #lonlatT[(lon,lat)] = T

           #  Note, this could all be compressed into fewer lines:
           for i in range(0,len(fields),1): fields[i] = float(fields[i])
           lonlatT[(fields[0], fields[1])] = fields[2]

           npts += 1
        f.close()

        print "Read", npts, "grid locations"

        while True :
           s = raw_input("\nEnter lon,lat (Return to exit) >")
           if len(s) < 1 : break

           try:
             s = s.split(',') #split input into lon,lat strings

             lon = round(float(s[0]))
             lat = round(float(s[1]))

             if (lon,lat) in lonlatT:
                print "T at",lon,lat, ":",lonlatT[ (lon,lat) ]

             else:
                print "Sorry,", s[0],s[1], "is not on land."

           except:
               print 'Try again.'
```

```
Read 18967 grid locations

Enter lon,lat (Return to exit) >-100,40
```

```
T at -100.0 40.0 : 11.5
```

3.3 File reading methods continued...

Sometime we have very large files, and we want to read only parts of them to make sure that they contain information we need. With some geospatial files, particularly large netCDF files, sizes can reach into gigabytes or terabytes. In this case we can use one of several methods for partial reading.

The `read` method uses the number of bytes as the limiting argument, reading only `nBytes`. If `nBytes` is negative, or undefined, then read the whole file. This works with both binary and text files:

```
In [1]: f = open("ClassRoster.txt")
        s = f.read(100)
        print s
        f.close()
```

```
Abel, John: Geography
Lewinski, Ralph: Communication Arts
Lassen, Bella: Environmental Studies
Poppi
```

In a file that is a text file it is possible to select by line as well. It's easy to see in the example below that we are selecting chunks of text along a single line. In the first example we keep selecting text by line. So even though 24 bytes should be enough to select "Geography" and some text from the second line in the file, the `readLine` method terminates at the end of the line, not beginning the next line until the subsequent call.

```
In [7]: f = open("ClassRoster.txt")
        s = f.readline(12)
        print s

        s = f.readline(24)
        print s

        s = f.readline(24)
        print s

        f.close()

        f = open("ClassRoster.txt")
        s = f.readline(24)
        print s
        f.close()

        f = open("ClassRoster.txt")
        s = f.readline(124)
        print s
        f.close()
```

```
Abel, John:
Geography

Lewinski, Ralph: Communi
Abel, John: Geography

Abel, John: Geography
```

In these examples if nBytes is not provided or is negative then the entire line is read, including the line terminator (
n).

The `readlines` method reads `nBytesHint` lines, and returns each line as a string element in a list. As in the methods above if no argument is provided to the method then the entire file is returned. The file `ClassRoster.txt` has 6 lines, so we should expect a list of length 6 to be returned:

```
In [9]: f = open("ClassRoster.txt")
        sList = f.readlines()
        print len(sList)
        f.close()
```

```
6
```

3.4 Methods for writing a file

The write methods are similar in structure to the read methods. The file will be written into your working directory unless an absolute path is provided. For this example we will create a new file using the `w` parameter, and then write out a single string to the file.

```
In [12]: f = open('testFile.txt', 'w+t')
         s = ['Seattle vs Dallas', 'New Orleans and New York', 'San Francisco and St. Louis']
         f.write(s[0])

         f.close()

         f = open('testFile.txt')
         print f.readlines()
```

```
['Seattle vs Dallas']
```

The `write` method for files works only with a single element. It doesn't work with a list. For that we need something analagous to `readlines`, and we find that we have a method called `writelines`:

```
In [13]: f = open('testFile.txt', 'w+t')
         s = ['Seattle vs Dallas', 'New Orleans and New York', 'San Francisco and St. Louis']
         f.writelines(s)

         f.close()

         f = open('testFile.txt')
         print f.readlines()
```

```
['Seattle vs DallasNew Orleans and New YorkSan Francisco and St. Louis']
```

Woah, what's going on here? This isn't what we put in! If we didn't know what the terms were supposed to be it would be hard for a program to figure out where each of the list elements starts and stops.

The reason it looks weird is because the write function is literal. It appends each text element to the existing end of file (EOF). If you want new lines you need to specify the end of line (EOL) symbol, `\n`. If we do that, you can see, we get proper recovery.

```
In [15]: f = open('testFile.txt', 'w+t')
         s = ['Seattle vs Dallas\n', 'New Orleans and New York\n', 'San Francisco and St. Louis\n']
         f.writelines(s)

         f.close()
```

```
        f = open('testFile.txt')
        print f.readlines()
        f.close()
```

```
['Seattle vs Dallas\n', 'New Orleans and New York\n', 'San Francisco and St. Louis\n']
```

If you want to write out numeric values with the various `write` methods, then you need to use the `str` function, but it's not always exactly what we want, or what we want to recover from a file:

```
In [19]: f = open('testFile.txt', 'w+t')
         a = [1, 2, 3, 4]
         f.writelines( str(a) )
         f.close()

         f = open('testFile.txt')
         print f.readlines()
         f.close()
```

```
['[1, 2, 3, 4]']
```

3.5 Example 10.4

Produce spreadsheet CSV file containing a table of sin(x), cos(x) to 3 digit accuracy for angles x between 0 and 360o at 15o increments. (CSV stands for comma separated values—spreadsheet columns are bracketed by commas. CSV files can be read by Excel and other spreadsheet programs. Unlike xlsx files, they contain nothing but text.)

```
In [18]: import math
         f = open("trig.csv","wt")

         columnHeads = 'Angle,  Sin(A), Cos(A)'

         f.write(columnHeads)

         for x in range(0,365,15):
             sinX = round(math.sin(math.radians(x)), 3)
             cosX = round(math.cos(math.radians(x)), 3)

             values = str(x)    + ',' + \
                      str(sinX) + ',' + \
                      str(cosX)

             f.write(values)
             f.write('\n')

         f.close()

         f = open('trig.csv')
         print f.readlines()
```

```
['Angle,  Sin(A), Cos(A)0,0.0,1.0\n', '15,0.259,0.966\n', '30,0.5,0.866\n', '45,0.707,0.707\n', '60,0.8
```

Note: All of the methods for reading and writing involve strings. We can read from a file into a string (or list of strings), and likewise are able to write strings—but that's all!

To write numeric values like `x` above, they must be converted to strings. The example uses the `string()` function for that conversion. As discussed in the next section, it isn't sufficient for anything beyond the most basic needs.

3.6 Exercise 10.2:

1. Modify the program in Example 10.3 so that temperature is stored and reported in oF rather than oC.
2. Read the temperature file and write out the records whose longitude is between 20 and 30 and latitude between 10 and 40.

The output format should be comma-delineated, for example:

```
22.00,33.00,36.20
22.00,32.00,42.00
```

Go to the Exercise 10.2. PDF on Learn@UW

4 Text Formatting

The `string()` function converts a value to a string, but we don't have any control how many characters are in the string, or how many digits to the right of the decimal will appear, etc. Thus we would have trouble, for example, if we wanted to write 3 columns of numbers and make sure the decimal points aligned. This is true whether we are writing to a file or to the console. For example:

```
In [1]: x = 7.23
        y = 10 * x
        z = 12 * x
        print str(x)
        print str(y)
        print str(z)

7.23
72.3
86.76
```

The string format operator (`%`) mentioned briefly in Topic 7 comes to the rescue. This operator (an operator is any symbol like `+`, `-`, `%` or `!`) builds a string out of two operands. One operand is a tuple containing a set of values to be converted into a string. The other operand is a string containing "instructions" about how the values are to be formatted. Thus the general syntax is:

```
In [ ]: newstring = 'format_string'  % (value_tuple)
```

The most confusing thing is that a `%` symbol often appears within a `format` string as well as between the format string and value tuple. Within a format string it is used specify a particular format.

For example:

```
In [21]: fmt = '%6.3f %6.3f'
         values = (math.pi, 2*math.pi)

         newstring = fmt   % values
         print newstring

3.142  6.283
```

Breaking this down: Both values (`math.pi` and `2*math.pi`) are floating point values (`%f`). They are printed in fields 6 characters wide (`%6.f`). There are 3 digits to the right of the decimal point (`%6.3f`).

4.1 Formatting Integers:

Suppose we have an integer x = 987. We will put it in a tuple (x,) and show the result for various formatting instructions.

```
In [23]: print "%d"    % (x,)
         print "%5d"   % (x,)
         print "%-5d"  % (x,)
         print "%5.5d" % (x,)
```

```
360
  360
360
00360
```

4.2 Formatting Floats

Suppose we have a float `x = 9.87`. We will put it in a tuple `(x,)` and show the result for various formatting instructions.

```
In [24]: x = 9.87

         print "%f"     % (x,)
         print "%5f"    % (x,)
         print "%5.2f"  % (x,)
         print "%e"     % (x,)
         print "%.2e"   % (x,)
         print "%10.2e" % (x,)
```

```
9.870000
9.870000
 9.87
9.870000e+00
9.87e+00
  9.87e+00
```

4.3 String format:

Suppose we have a string x = 'string'. We will put it in a tuple (x,) and format it as a string. We do this to control the number of characters displayed or the justification.

```
In [26]: x = 'abcd'
         print "%s"    % (x,)
         print "%10s"  % (x,)
         print "%-10s" % (x,)
```

```
abcd
      abcd
abcd
```

4.4 Example 10.5

Problem: Produce a nice table of sin(x), cos(x) to 3 digit accuracy for angles x between 0 and 360o at 15o increments. Make each column the same width with decimal points aligned.

```
In [38]: import math
         f = open("trig.txt","wt")

         columnHeads = ( "Angle", 'Sin(A)', 'Cos(A)' )

         headingFormat = "%10s %10s %10s\n"

         s = headingFormat % columnHeads

         f.write(s)

         for x in range(0,365,15):
             sinX = math.sin(math.radians(x))
             cosX = math.cos(math.radians(x))

             fmt = "%10d %10.3f %10.3f\n"
             values = (x, sinX, cosX)
             f.write(fmt % values )

         f.close()

         f = open("trig.txt")

         for i in range(0,10,1):
             print f.readline().strip('\n')
```

```
Angle     Sin(A)     Cos(A)
         0      0.000      1.000
        15      0.259      0.966
        30      0.500      0.866
        45      0.707      0.707
        60      0.866      0.500
        75      0.966      0.259
        90      1.000      0.000
       105      0.966     -0.259
       120      0.866     -0.500
```

5 File Exceptions

Run-time errors are common when working with files. A user might name a file that doesn't exist, or the file might not be formatted as expected. Use of `try`, `except` blocks is strongly recommended when working with files. These prevent the whole program from bombing just because a user doesn't know where their files are.

For example:

```
In [20]: filename = raw_input ('Enter a file name: ')
         try:
              f = open(filename, 'r')
         except:
              print "can't open",filename

         # do some stuff. . .
```

```
Enter a file name: whazzap.90s
can't open whazzap.90s
```

5.1 Exercise 10.3:

Test the above exception handling example. Feed the program files that don't exist, improperly formed paths, etc.

Go to the Exercise 10.3. PDF on Learn@UW

6 Command Line Arguments

It is very common to pass file names to a script as arguments. How can we do this with a Python script? Recall that standard Windows programs and batch files are executed by

```
In [ ]: ProgramName Argument1 Argument2 . . .
```

e.g.,

```
In [ ]: notepad MyFile.txt
        gdalinfo MadisonWest.tiff
        ogr2ogr -f "KML" parcels.kml parcels.shp
        myBatch Input.txt Output.txt
```

If the program name is in the path (or if you are in the proper working directory), the program will be run. The arguments will be made available to the program. That is, the OS will hand them to the program, assuming the program is expecting them.

Suppose we have a program in a file myProg.py. Recall that in general we can't just type "myProg.py" because it is not an executable. Instead we call the script through python:

```
Python myprog.py
```

or

```
Python myprog.py arg1 arg2 arg3 ...
```

In this second case, you can think of the filename `myprog.py` as the Shell argument `%1`, and each arg as `%2`, `%3`, etc. Wheras this behavior is built into batch files, a Python file needs to have the `sys` module loaded to capture the arguments.The module must be imported before it is used:

```
In [ ]: import sys
```

Once this line is added to your code there will be a variable `sys.argv` that will contain the command line arguments as a list of strings, that you would then process. This may be a little confusing during the debugging process, you never have to explicitly call `sys.argv`, it is passed whenever the file is executed from the Shell.

Note: The first item in the list (`sys.argv[0]`) is always the name of the python script that is being executed. Other arguments appear sequentially in the list as `sys.argv[1]`, `sys.argv[2]`, . . . For example, if I do this:

```
Python Resize.py  BigImage.tif   Thumbnail.tif
```

Python will run the program contained in `Resize.py`. Assuming that the program imports the `sys` module there will be a list named `sys.argv` within the program. The contents of that list will be:

```
In [ ]: ['Resize.py', 'BigImage.tif', 'Thumbnail.tif']
```

Note that the operating system/python copied what was entered on the command line into the list as strings. It has no idea if the arguments are file names or anything else. Even if one of the arguments is a number it will be of type `str` in the `sys.argv` list. Our program must make intelligent use of what was entered.

`len(sys.argv)` is the number of arguments, including the program name.

It might be possible to type >myprog.py arg1 arg2. In particular, if the .py extension is registered to Python, the Python will start.

6.1 Example 10.6

write a python script to multiply two numbers supplied by the user on the command line. To make this work you need to copy to a file and then run this script from the command line.

```
In [ ]: import sys
        try:
            x = float(sys.argv[1])
            y = float(sys.argv[2])
            print "The product of %f and %f is %f" % (x, y, x*y)
        except:
        print "Sorry, I need two numeric values."
```

6.2 Exercise 10.4:

Write code to take the radius of a circle as command line argument and print out the area of the circle.

Go to the Exercise 10.4. PDF on Learn@UW

7 Binary mode and binary data

Many GIS file formats have binary rather than character (text) data. Does Python's binary mode help? For example, can we read a shape file by f = open('file.shp','rb') ? Sadly, no. . .

Python's binary mode only affects the interpretation of newlines (
n). In particular, it controls whether a newline in a file is translated into two characters or one. In `text` mode a newline is converted to one character. In `binary` mode a newline appears as two characters. The details are found below, but these are beside the main point — see the conclusion for that.

For those that simply have to know, here is how it works in the Windows world: `\n` is two characters, a carriage return followed by a linefeed (ascii table positions 13 and 10). If you read a file in binary mode every "
n" will be stored as two characters in your string. If you read a file in text mode every "
n" will be stored as one character, a line feed (ascii 10)

7.1 Example 10.7

Print the line length and ascii number of the last three characters of a line so we can see how values are represented:

```
In [37]: print 'Reading file in binary mode'
         f = open("trig.txt",'rb')
         for i in range(0,5,1):
             line = f.readline()
             print len(line),'characters, last 3 are:',ord(line[-3]),ord(line[-2]),ord(line[-1])

         print '\nReading file in text mode'
         f = open("trig.txt",'rt')
         for i in range(0,5,1):
             line = f.readline()
             print len(line),'characters, last 3 are:',ord(line[-3]),ord(line[-2]),ord(line[-1])
```

```
Reading file in binary mode
34 characters, last 3 are: 41 13 10
34 characters, last 3 are: 48 13 10
34 characters, last 3 are: 54 13 10
34 characters, last 3 are: 54 13 10
34 characters, last 3 are: 55 13 10
```

```
Reading file in text mode
33 characters, last 3 are: 65 41 10
33 characters, last 3 are: 48 48 10
33 characters, last 3 are: 54 54 10
33 characters, last 3 are: 54 54 10
33 characters, last 3 are: 48 55 10
```

To interpret the output, recall the file looks like so Angle Sin(A) Cos(A) 0 0.000 1.000 15 0.259 0.966 30 0.500 0.866 45 0.707 0.707 60 0.866 0.500

7.2 Conclusions

Binary mode doesn't help us to read files that are explicitly in a binary format. To read binary data we need file processing that allows us to read a string of bytes and turn it into recognizable objects. *e.g.*, to read a binary integer, we need some way to read a sequence of 4 bytes directly into 4 bytes in memory representing an integer.

This capability is provided by the `struct` module, among others. This won't be discussed because we will use `gdal` and `ogr` modules to open GIS files. They may incorporate elements of the `struct` (or related) modules, but they add a layer over top of these "lower level" commands to directly load spatial data in to Python.