Lectures 3–4: Working with Files and Data, including geospatial methods and plotting

Andrew D. Wickert

May 20, 2015

The geosciences are full of a range of data types—from mapping and surveys to chemical analyses to "data" from computer model outputs (spanning a whole range of complexities) to remotely-sensed images and more. Working with all of these data types in an efficient way is one of the primary uses of computers in the geosciences. In this section, you will learn:

1. How computers store data
2. How to work with basic ASCII and binary data sets, using Numpy tools and basic file-reading operations.
3. How to work with spreadsheets (e.g., Excel) when programming
4. How to work with geospatial data both within Python on its own and within a GIS framework

1 Data storage and retrieval, data plotting, GIS, and Python modules

Data can be stored as text (typically ASCII) or binary values. Often, one file contains one data set. However, more advanced data storage formats like NetCDF and HDF are also widely-used—these act as containers for multiple related pieces of multidimensional data. NetCDF is more common and is often used for amospheric, oceanographic, glaciologic—generally climate-related data. atmospheric science data and models. It is now also being used by the Community Surface Dynamics Modeling System (CSDMS) as a standard format for geological model input and output. HDF was developed by NASA, and therefore is often used for satellite remotely-sensed data. Unless you do quite a lot of work with these kinds of data and models, you may not come across these—but it is important to know at least a little about them.

Along with the basics of data storage and retrieval, we are going to have to learn how to display the data. This will likely be one of the more useful skills taught in theis course, because you will learn how to write reusable computer code to generate publication-quality figures. Learnin gthe plotting commands takes a bit of time, as does writing the code—but in my experience, this saves hours in the long run. Having the ability to generate reusable plots also makes one more willing to revise data analyses, knowing that time will not be wasted *per se* on laboriously recreating plots by hand. It also lets one make a whole set of similar plots for displaying large amounts of data.

Oftentimes in the geosciences, we want to plot geospatial data. I will show how you can install mapping packages and use GIS tools to do just this. I will also point out and (in some cases) very briefly discuss the Python interfaces to GRASS GIS, Arc GIS, and QGIS, the three leading scientific geospatial platforms. These come with a set of premade tools to make it possible to run very involved analyses in just a few lines of code.

Using some of these tools involves downloading and installing @TODO: Modules section

2 ASCII

2.1 Theory

In the Introductory notes, you learned about the ASCII table for encoding text. Each ASCII character requires 7 bits to be encoded. This increases to 8 bits, or 1 byte, as a "stop" bit is added—this was historically used to note if there was an error in the transmission.

In ASCII, the number "45" would require 2 bytes of storage. "61.083" would require 6 bytes. "-15E-3", where "E" denotes that 15 is multipled by 10 to the following power (in this case, -3), would also require 6 bytes. In most situations, ASCII data take more storage space than binary data. They have the advantage, though, in that they are fully human-readable.

The great line-ending schism

A very important historical note is the controversy about line-endings. It may sound trivial, but it can turn a successful data import into a failure! Here it is:

- UNIX-based computers end lines of text with a newline character, `\n` (ASCII 10)
- DOS/Windows-based computers end lines of text with a carriage return, `\r\n` (ASCII 13, then ASCII 10)

This seemingly small detail has its basis in the mechanics of the transition from typewriters to computers: with typewriters, one must advance to a new line ("line feed" or "newline", ASCII 10) and push the roll of paper all the way back to the start of the line ("carriage return", ASCII 13). Computers can instead produce a new line all at once, because they are not limited to moving paper in physical space. This led to ASCII 10 being used on UNIX systems, and the typewriter-looking [ASCII 13, ASCII 10] being used on Windows systems, and a potential whole world of trouble! It can mean that, if you have a file with delimited data (e.g., csv – comma-separated values), that you cannot tell where one line ends and another begins if your computer looks for the wrong newline character! This is a problme that can often happen if, like most computer users, one is working on Windows but then needs to do a probject on a supercomputer (mostly UNIX). Many pieces of software are becoming smarter about this difference, but this is one of the great and unfortunate schisms in computing that happened in the early days, when we were really just learning what we are doing, and has been carried forward to the present.

2.2 Practice

OK, enough with the theory. Let's get our hands dirty! Figuratively, of course, in case you have been eating in a particularly messy way over your computer keyboard. (Aside: have you ever shaken out an old keyboard? Don't do it over your face. Especially not with your mouth open. No, I wouldn't know.)

As mentioned in the introductory section, there is a Matplotlib tutorial at http://matplotlib.org/users/pyplot_tutorial.html. And for those of you who like to learn about these sorts of things by following examples, Matplotlib has an excellent plotting gallery—a place where it shows you a whole range of graphics and how to create them, at http://matplotlib.org/gallery.html.

You should have already installed `numpy`. This is the numerical Python package and is important for doing work with all sorts of arrays. We are going to start by importing some data from an ASCII text file. In this case, it will be a simple transect of elevations across the park by where I grew up. We will use both the Numpy `genfromtxt` feature to create a numpy array and the `pandas read_csv` feature to generate a Pandas DataFrame. This example already includes plotting with matplotlib as does a section in the introductory notes. This, I hope, will be fairly self-explanatory.

If you are on Windows or Mac, your Python distribution should have come with Pandas. If you are on Linux (or Mac and are using a package manager) and do not yet have Pandas, you can type `sudo apt-get install python-pandas` or a similar command.

pip and easy_install

`pip` and `easy_install` are two tools to automatically add new Python packages. These are packages build by the community of Python users. You may need them for some of the packages to work with specific data sets in this guide. Indeed, there are so many useful such packages to do many of the activities that you might want to do that it is often good to do an Internet search for “python <task you want to do> before starting any work.

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

# 1 -- numpy
import numpy as np
from matplotlib import pyplot as plt

# Strip out header information
# Just remember that columns are: x, z, lat, lon
data = np.genfromtxt('../../data/BattleCreekProfile.txt', skip_header=1, \
                           delimiter=',')

# Plot as distance profile
plt.plot(data[:,0], data[:,1], 'k', linewidth=2)
plt.xlabel('Distance [km]')
plt.ylabel('Elevation [m]')
plt.title('NUMPY!')
plt.show()

# 2 -- pandas
import pandas as pd

# Import it as a data frame -- keep information in a coordinate system
data = pd.read_csv('../../data/BattleCreekProfile.txt')

# Plot -- hey look, distance is inferred!
plt.plot(data['Elevation (m)'], 'k', linewidth=2)
plt.xlabel('Distance [km]')
plt.ylabel('Elevation [m]')
plt.title('PANDAS!')
plt.show()

# Think about how useful this could be for time-series data!
# Indeed, this is the main use of Pandas, which is designed as a
# data-management library
```

code/FilesData/CSVexample.py

You can also save ASCII data using the np.savetxt feature. I mostly use it as follows:

```python
np.savetxt('OUTPUT_FILE_NAME.TXT', InputVariableName, fmt='%FORMAT_STRING')
```

Here, it is important to mention **formatting strings**. These follow the conventions from the C programming language. I will just introduce my most-used subset of them here.

Formatting strings contain a character. This determines how the incoming value is treated, but all output is as a string:

- d: integer
- f: floating point (“float”)

- s: string

These are then modified by numbers placed in front of the digits. Both the formatting string and the value that one would like to format as such must be preceded by a %, and the formatting string must be inside quotes to denote that it is a string. I will explain through a set of examples:

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

# Formatting strings

# d -- integer
# f -- floating point
# s -- plain string

# Let's see how these change the formatting of a number.

num = 14.81
print "Original number", num
print ""
print '%d:',
print ('%d' %num)
print ""
print '%f:',
print ('%f' %num)
print ""
print '%s:',
print ('%s' %num)
print ""


# Note that the '%d' option just cuts off the decimal without rounding the
# number! So it is doing floor division.

# The '%f' option has a pre-set decimal precision.

# The '%s' option just converts the number into a string, in the same way that
# str(num) would do so.


# Now let's start having a bit more control.

# %d
# The number of digits in a decimal nummber may be specified
# This can create spaces before that number
print '%5d %14.81 :'
print ('%5d' %14.81)
print ""
# But will not truncate the number: it simply sets the minimum number of digits
# that must be displayed.
print '%5d %6132614.81 :'
print ('%5d' %6132614.81)
print ""
# It can also be used to generate zeros before a number instead of the spaces.
# This is known as zero-padding (0-padding)
print '%05d %14.81 :'
print ('%05d' %14.81)
print ""

# %f
# Floating point numbers can be formatted as follows
# [TOTAL NUMBER OF CHARACTERS, INCLUDING DECIMAL].[NUMBER OF DIGITS AFTER DECIMAL POINT]
# -- OR --
# .[NUMBER OF DIGITS AFTER THE DECIMAL POINT]
# (with the number of digits before the decimal point being fit to the number
#  that you are formatting as a string)

# If there is not enough precision, it still is truncated. But in the
# floating-point case, it is rounded.
print '%4.1f %14.89 :'
print ('%4.1f' %14.89)
print ""
# This is the perfect size for this
print '%5.2f %14.89 :'
print ('%5.2f' %14.89)
print ""
# This automatically sets the proper space to the left of the decimal point
print '%.2f %14.89 :'
print ('%.2f' %14.89)
print ""
# Extra space and decimal places
print '%9.4f %14.81 :'
print ('%9.4f' %14.89)
print ""
# 0-padding with extra decimal places
print '%09.4f %14.81 :'
print ('%09.4f' %14.89)
```

code/FilesData/formattingStrings.py

Saving multiple files with a numbering scheme

In many cases, you may have data that belong in a particular order—whether they are a set of time-steps, a series of different analyses, or a number of different sample ID's that you want to automatically generate. For the case with real dates and times, you may use that date and time as part of the file name. And if you use it as **yyyymmdd**, it will alphabetically sort correctly! This becomes harder though if it is an arbitrary time step – like say, millions of years ago, seconds, or just an arbitrary set of **[0, 1, 2, ..., 51, etc.]**. If you want the data to be sorted in order when you do a simple alphabetical sorting – great for viewing in file browsers, loading into data analyses, or just for general orderly storage, you can run into the problem in which your data show up as **[0, 1, 10, 100, 2, ..., 51, etc.]**. This is certainly not what we want? So how do we fix it?

We use **zero-padding** (see code above on string formatting). So in our above example, if we can safely say that all the numbers are integers, and nothing will go over 9999, we can write an output file name as:

```
fnpadded = 'some_descriptive_text_' + '%04d' %time_step + '.ext'
```

And then you can use this to save text output as a floating point with 2 decimal places of precision, called for example plate_reconstruction, as follows:

```
np.savetxt(fnpadded, plate_reconstruction, fmt='%.2f')
```

Sometimes, ASCII data are not so easy to work with. This can happen when they are not in a simpe grid. In that case, we have to use lower-level Python commands. To learn basic file handling, you may see the help at http://www.pythonforbeginners.com/files/reading-and-writing-files-in-python.

One example of this sort of data is a set of GPS tracks provided by Ben P. These look something like:

```
0
728609.215175176 7272865.670114477 3850.1122000000032
728608.813141581 7272866.856349733 3850.1769000000058
...
728588.585519101 7272909.968807690 3851.2842999999993
END
1
729772.085292112 7284330.246414313 3857.9808000000048
...
729789.783887492 7284223.112814812 3858.0691999999981
END
...
```

The numbers and "END" lines separate individual tracks. However, these are not nice 3-column entries. How do we parse a file like this?

Using the more basic Python file read/write commands (see links above), one can input the lines of the file, parse them into lists, and then turn them into arrays. This is as follows:

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

import numpy as np
from matplotlib import pyplot as plt

# Open text file for reading
f = open('../../data/PG_astgtm2_dgps_tracks.txt', 'r')

# Prepare a list to contain each individual track
tracks = []

# Now there are two ways to do this. The first is easier, and requires more
# memory on your computer. In this, we will read in the whole file at once
# and then parse it.

# This reads all lines in the file, and creates a list in which
# each entry is a string that is that line.
text = f.readlines()


# Go back to the beginning of the file
f.seek(0)

# This list will contain the points in each individual track.
track = []
# and this will contain the track number as recorded by the GPS
track_numbers = []

# This next statement is *asking* for an infinite loop. That is why
# I am introducing a new flow-control statement, break, that will get
# us out of it.
while True:
  # readline() reads the next line; strip() removes newline characters
  line = f.readline().strip()
  if line:
    try:
      # if the line is an integer, it is a track number
      track_numbers.append(int(line))
    except:
      # Check if the line is ending a section; if it is, package the section
      # and ship it off to the "tracks" list in its own numpy array.
      if line == 'END':
        # Only do this if we need another track to be entered; there are
        # two "END"s at the end of the file, so this will prevent it from
        # adding an empty track there that does not correspond to the numbers
        if len(track_numbers) > len(tracks):
          tracks.append(np.array(track))
        # reset individual track list for the next one
        track = []
      else:
        # I will here in two steps split the line with data into a list and
        # turn it into a numpy array of floating point values.
        tmp0 = line.split(' ') # Split it at the spaces
        # Everything is still a string, so need to tell numpy to make the
        # array of floating-point values
        tmp1 = np.array(tmp0, dtype=float)
        # Now append it to track -- we will change this to an array in the
        # step before we append it to the "tracks" master list (above)
        track.append(tmp1)
  else:
    # This is how we get out of the potentially infinite loop: if the
    # line is empty
    break

# Now let's plot all of these tracks' x and y components
# with the default set of different colors per line
# Remember, [:,i] means ALL ROWS IN iTH COLUMN
fig = plt.figure()
for line in tracks:
  plt.plot(line[:,0], line[:,1]) # Easting, Northing
plt.title('GPS tracks', fontsize=20, fontweight='bold')
plt.ylabel('Northing', fontsize=20)
plt.xlabel('Easting', fontsize=20)
plt.tight_layout() # Formatting helper
plt.show()

# And let's now combine all of the tracks together into a single numpy array
# to create a large set of points
# Numpy arrays play nicely with numpy lists, so we just need to use the
# concatenate command to do this in one step!
# Other ways to combine numpy arrays include using "np.vstack" and "np.hstack"
alltracks = np.concatenate(tracks)

# And let's see if there is any statistical clustering of elevations
plt.hist(alltracks[:,-1], bins=100)
plt.title('GPS tracks')
plt.ylabel('Number of measurements', fontsize=20)
plt.xlabel('Elevation [m]', fontsize=20)
plt.tight_layout() # Formatting helper
plt.show()
# Yep, there are some distinct hypsometric peaks!
```

code/FilesData/readGPStracks.py

3 Binary

3.1 Theory – and how to get the most out of storage space

Binary data are represented as a set of ones and zeros. In the Introductory notes, you learned how binary works, and a bit about how numbers may be generated.

Binary data are usually more compact than ASCII data, especially if used correctly. For example, a number between 0 and 65535 may be represented by two bytes (16 bits); this can be seen in binary because $2^16 = 65536$, and we want to include 0, so have to shift the maximum value down one. This is called a 16-bit unsigned integer, because it has 16 bits of data, and does not include a + or - sign (and hence is always positive). A number like this would require 5 bytes in ASCII to represent.

If you are representing a large number of binary values (0 or 1) as 16-bit unsigned integers, you would be using 2 bytes of data per value, while ASCII would require only 1 byte. So in this case, ASCII would be better! But this is where it becomes important to *intelligentlly choose binary representations of data*. If we represented each of these values as binary logical values, each would only require 1 bit of storage space—an 8× improvement over ASCII and a 16× improvement over the 16-bit unsigned integer, which is really just too much storage space for binary data.

We often denote these values as follows:

- unsigned integer: uint
- signed integer (so can be + or −): int
- floating point: float

You might also see terms like single, double, char, word, etc.; these are less-descriptive terms that also relate to number of bytes in data. I use the three above to be more clear to those who have not memorized what all of these are.

These can also be used to denote how many bits are involved in each data type. For example:

- unsigned 16-bit integer: uint16
- signed 32-bit integer (so can be + or −): int32
- 64-bit floating point: float64

3.2 Practice – raw binary files

```
#! /usr/bin/env python
import numpy as np

# Create a 2x3 numpy array
a = np.array([[1,2,3],[4,5,6]])

# Save it as a straight binary output with differnet precisions.
# Check the filesize after each of these
filename = 'testout.bin'

# 8-bit unsigned integer
a.astype('uint8').tofile(filename)

# 64-bit floating point
a.astype('float64').tofile(filename)

# Now load the saved file
b = np.fromfile(filename, dtype='float64')
```

```
# Hm, we lost the information about the line ending! This is because it
# is just a string of binary without any shape data.

# Let's see what happens if we try to load it as a 8-bit integer (signed)
c = np.fromfile(filename, dtype='int8')
# WOW! It worked. But what do the values look like? Hm... so you
# could combine those sets of binary values together in clumps of 8, and
# convert those 64-bit clumps of binary data into the original set of
# values. Not bad, huh? That's binary for ya!
```

code/FilesData/BinaryIO.py

As a quick mental exercise, imagine that you have the numbers -15, 35, 119, and 43. What is the ideal number of bits with which to represent each number? How about 0, 50612, 151, 10512, 85, 3160? *Answers: 8 bits (int8), 16 bits (uint16).*

3.3 Binary containers

3.3.1 Numpy files

Standard binary data formats are all right. But there are formats that can remember rows and columns of data, as well as "container" formats that can contain multiple arrays. So let's look at the native ones in numpy first.

```
#! /usr/bin/env python
import numpy as np

# Create two 2x3 numpy arrays
# int8 is more than enough to represent these data
a = np.array([[1,2,3],[4,5,6]], dtype='int8')
# float16 is a really rarely-used format, but I am using it here just to
# illustrate how it stores these data
b = np.array([[1.6,0.1,3.512],[-27,5,6.5109]], dtype='float16')
# look at how imprecise b is!
print b

# Let's look at a magic trick that numpy does. You know already that int8 can
# only take numbers from -128 to +127. Well, what happens if we add 500 to a?
c = a + 500
print c
print c.dtype
# Hey -- it made it be a int16! That's pretty cool
# (Other languages like C would instead wrap around to -128 and continue
# forward, causing potential big problems)

# Now let's save an array into an *.npy file
np.save('testout.npy', a)
# And load it -- it keeps the structure of the rows and columns, great!
d = np.load('testout.npy')

# How about saving multiple arrays? we can use a compressed *.npz file
np.savez('testout.npz', a=a, b=b, c=c)
# The reason for the i=i format is because this is telling us to take array
# "c", for example, and to also call it "c" in the *.npz file.
# We could likewise change the names:
np.savez('testout.npz', x=a, y=b, z=c)

# Now let's load it
zfile = np.load('testout.npz')
print zfile['x'] == a
print zfile['y'] == b
print zfile['z'] == c
```

code/FilesData/npIO.py

3.3.2 Raster images

@TODO: add these later

3.3.3 Working with NetCDF and HDF files – in brief!

@TODO: add these later

NetCDF

HDF

4 Working with Spreadsheets (Excel or LibreOffice Calc)

So far, we've discussed all of these data formats that might be a bit more complex – and these are all important for work with computers and larger data sets. However, many of us in the Earth sciences deal with spreadsheet data. Up until now, most (all?) of you have worked only by hand with this data. But no longer!

There many packages to work with spreadsheets. I will show you only the builtin method with Pandas here, but you should know about at least the two following ones (available via pip):

- openpyxl (Excel)
- odfpy (LibreOffice)
- oosheet (LibreOffice calc – a spreadsheet-focused alternative to odfpy)

Here, we are going to plot an oxygen isotope history from a Greenland ice core:

```python
import pandas as pd
from matplotlib import pyplot as plt

# Import data
sheet_title = 'Renland d18O' # set outside for use in plotting
data = pd.read_excel('../../data/vinther2008renland-agassiz.xlsx', \
                     sheetname=sheet_title, header=72)

# Give headers better names
data.columns = ['years [b2k]', 'depth [m]', 'del18O [permil]']

# And plot
plt.plot(data['years [b2k]'], data['del18O [permil]'])
#plt.gca().invert_yaxis() # Grabs current axes and inverts x-axis
                          # uncomment if you prefer to see time this way.
# Automatically use column labels in the plot
plt.title(sheet_title, fontsize=16, fontweight='bold')
plt.xlabel(data.columns[0], fontsize=16)
plt.ylabel(data.columns[-1], fontsize=16)
plt.show()
```

```
# If we want better axis labels, we can use the LaTeX formatting:
plt.plot(data['years [b2k]'], data['del18O [permil]'])
plt.title('Renland', fontsize=16, fontweight='bold')
plt.xlabel('Years b2k', fontsize=16)
plt.ylabel('$\delta^{18}$O', fontsize=16)
plt.show()
```

code/FilesData/GreenlandXLSXpandas.py

5 Plotting geospatial data using Basemap

6 Special Python modules for data sets

7 Time series and the datetime lilbrary