Johnny Wei-Bing Lin

A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences

http://www.johnny-lin.com/pyintro

2012

Who would *not* want to pay money for this book?: if you do not need a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, have limited funds, or are interested in such a small portion of the book that it makes no sense to buy the whole thing. The book's web site (http://www.johnny-lin.com/pyintro) has available, for free, PDFs of every chapter as separate files.

Who would want to pay money for this book?: if you want a black-and-white paper copy of the book, a color PDF copy with functional hyperlinks, or you want to help support the author financially. You can buy a black-and-white paper copy of the book at http://www.johnny-lin.com/pyintro/buypaper.shtml and a hyperlink-enabled color PDF copy of the book at http://www.johnny-lin.com/pyintro/buypdf.shtml.

A special appeal to instructors: Instruction at for-profit institutions, as a commercial use, is not covered under the terms of the CC BY-NC-SA, and so instructors at those institutions should not make copies of the book for students beyond copying permitted under Fair Use. Instruction at not-for-profit institutions is not a commercial use, so instructors may legally make copies of this book for the students in their classes, under the terms of the CC BY-NC-SA, so long as no profit is made through the copy and sale (or Fair Use is not exceeded). However, most instruction at not-for-profit institutions still involves payment of tuition: lots of people are getting paid for their contributions. Please consider also paying the author of this book something for his contribution.

Chapter 6

A “Real” AOS Project: Putting Together a Basic Data Analysis Routine

At this point, we’ve covered enough of Python for you to do basically any atmospheric or oceanic sciences calculation you would normally use a data analysis language like IDL or Matlab for (excepting visualization, which we’ll cover in Ch. 9). So let’s put what we’ve learned to the test and do a “real” AOS data analysis project.

In Section 6.1, I present your mission (should you accept it ☺). In Sections 6.2–6.5, I give four different ways of solving the problem. Why four solutions? I want to use this real-world-like project to demonstrate how the modern features of Python enable you to write much more powerful and robust programs than are possible in traditional compiled and data analysis languages; you can write a Fortran-like program in Python, but if you do, you’ll miss features of Python that can help make your life as a scientist much easier. Finally, we finish with some exercises where we use these modern methods to extend our data analysis program.

6.1 The assignment

Let’s say you have three data files named *data0001.txt*, *data0002.txt*, and *data0003.txt*. Each data file contains a single column of data of differing lengths (on the order of thousands of points). The data files have no headers. Write a program that:

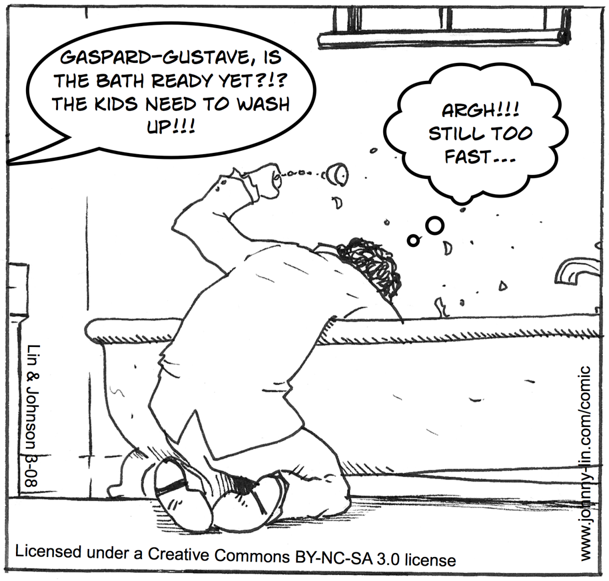
- Reads in the data from each file into its own NumPy array.

- Calculates the mean, median, and standard deviation of the values in each data file, saving the values to variables for possible later use.

While you can do this assignment without recourse to a real dataset, there are three data files so structured in the *course_files/datasets* directory. The data is random (Gaussian distributed), with the first dataset *data0001.txt* having a mean and standard deviation of 1, the second *data0002.txt* having a mean and standard deviation of 2, etc., so that you can see whether you're getting the right result. Specifically, NumPy will calculate the mean and standard deviation as:

```
data0001.txt:  0.962398498535 1.00287723892
data0002.txt:  2.02296936035  1.99446291623
data0003.txt:  3.08059179687  2.99082810178
```

Hints: The NumPy function for calculating the mean is `mean`, the median is `median`, and the standard deviation is `std`.


GASPARD-GUSTAVE, IS THE BATH READY YET?!? THE KIDS NEED TO WASH UP!!!
ARGH!!! STILL TOO FAST...
Lin & Johnson 3-08
www.johnny-lin.com/comic
Licensed under a Creative Commons BY-NC-SA 3.0 license

Bath Time at the Coriolis Household

6.2 Solution One: Fortran-like structure

In this solution, I've put all the file open, closing, read, and conversion into a function, so you don't have to type `open`, etc., three times. Then, I make use of NumPy's statistical functions to analyze the data and assign the results to variables. The way it's written, however, looks very Fortran-esque, with variables initialized and/or created explicitly.

```
1  import numpy as N
2
3  def readdata(filename):
4      fileobj = open(filename, 'r')
5      outputstr = fileobj.readlines()
6      fileobj.close()
7      outputarray = N.zeros(len(outputstr), dtype='f')
8      for i in xrange(len(outputstr)):
9          outputarray[i] = float(outputstr[i])
10     return outputarray
11
12 data1 = readdata('data0001.txt')
13 data2 = readdata('data0002.txt')
14 data3 = readdata('data0003.txt')
15
16 mean1 = N.mean(data1)
17 median1 = N.median(data1)
18 stddev1 = N.std(data1)
19
20 mean2 = N.mean(data2)
21 median2 = N.median(data2)
22 stddev2 = N.std(data2)
23
24 mean3 = N.mean(data3)
25 median3 = N.median(data3)
26 stddev3 = N.std(data3)
```

The program above works fine, but we haven't really taken much advantage of anything unique to Python. How might we change that? For instance, in `readdata`, instead of using a loop to go through each element and convert it to floating point, we could use array syntax and the `astype` method of NumPy array objects. The code to replace lines 7–9 would be:

Example of using `astype` for array type conversion.

```
outputarray = N.array(outputstr)
outputarray = outputarray.astype('f')
```

This doesn't really change much, however. The program is still written so that anytime you specify a variable, whether a filename or data variable, or an analysis function, *you type it in.* This is fine if you have only three files, but what if you have a thousand? Very quickly, this kind of programming becomes not-very-fun.

6.3 Solution Two: Store results in arrays

One approach seasoned Fortran programmers will take to making this code better is to put the results (mean, median, and standard deviation) into arrays and have the element's position in the array correspond to *data0001.txt*, etc. Then, you can use a `for` loop to go through each file, read in the data, and make the calculations. This means you don't have to type in the names of every mean, median, etc. variable to do the assignment. And, since we also have Python's powerful string type to create the filenames, this approach is even easier to do in Python than Fortran. The solution, then, is:

```
import numpy as N
num_files = 3
mean = N.zeros(num_files)
median = N.zeros(num_files)
stddev = N.zeros(num_files)

for i in xrange(num_files):
    filename = 'data' + ('000'+str(i+1))[-4:] + '.txt'
    data = readdata(filename)
    mean[i] = N.mean(data)
    median[i] = N.median(data)
    stddev[i] = N.std(data)
```

(I left out the definition of `readdata`, which is the same as in Solution One. This is also true for all the other solutions to come.)

This code is slightly more compact but scales up to any `num_files` number of files. But I'm still bothered by two things. First, what if the filenames aren't numbered? How then do you relate the element position of the `mean`, etc. arrays to the file the calculated quantity is based on? Variable names (e.g., `mean1`) *do* convey information and connect labels to values; by putting my results into generic arrays, I lose that information. Second, why should I predeclare the size of `mean`, etc.? If Python is dynamic, shouldn't I be able to arbitrarily change the size of `mean`, etc. on the fly as the code executes?

6.4 Solution Three: Store results in dictionaries

Before looking at this solution, we first need to ask how might dictionaries be useful for our problem. We previously said variable names connect labels to values, meaning that a string (the variable name) is associated with a value (scalar, array, etc.). In Python, there is a special construct that can associate a string with a value: a dictionary. From that perspective, setting a value to a key that is the variable name (or something similar), as you do in dictionaries, is effectively the same as setting a variable with an equal sign. However, dictionaries allow you to do this *dynamically* (i.e., you don't have to type in "variable equals value") and will accommodate *any* string, not just those numbered numerically.

Variable names connect labels to values.

Dictionaries can dynamically associate strings with values.

Here, then, is a solution that uses dictionaries to hold the statistical results. The keys for the dictionary entries are the filenames:

```
import numpy as N
mean = {}    #- Initialize as empty dictionaries
median = {}
stddev = {}
list_of_files = ['data0001.txt', 'data0002.txt',
                 'data0003.txt']

for ifile in list_of_files:
    data = readdata(ifile)
    mean[ifile] = N.mean(data)
    median[ifile] = N.median(data)
    stddev[ifile] = N.std(data)
```

So, in this solution, instead of creating the filename each iteration of the loop, I create a list of files and iterate over that. Here, it's hard-coded in, but this suggests if we could access a directory listing of data files, we could generate such a list automatically. I can, in fact, do this with the `glob` function of the glob module:[1]

Using `glob` to get a directory file listing.

```
import glob
list_of_files = glob.glob("data*.txt")
```

You can sort `list_of_files` using list methods or some other sorting function. (See the discussion on p. 111 which briefly introduces the built-in `sorted` function.)

[1]See http://docs.python.org/library/glob.html for details on the module (accessed August 16, 2012).

Another feature of this solution is that statistical values are now referenced intelligently. That is to say, if you want to access, say, the mean of *data0001.txt*, you type in `mean['data0001.txt']`. Thus, we've fixed the issue we had in Solution Two, where the element address of the variable `mean` had limited meaning if the dataset filenames were unnumbered. Cool!

Example of namespace protection in Python.

An aside: Again, you don't need the continuation backslash if you're continuing elements of a list (or similar entities) in the next line. Also, because of Python's namespace protections (see p. 41 for more details), we can have a variable named `mean` in our program that will not conflict with the NumPy function `mean`, because that function is referred to as `N.mean`.

6.5 Solution Four: Store results and functions in dictionaries

The last solution was pretty good, but here's one more twist: What if I wanted to calculate more than just the mean, median, and standard deviation? What if I wanted to calculate 10 metrics? 30? 100? Can I make my program flexible in that way?

Dictionaries can hold any object, even functions and other dictionaries.

The answer is, yes! And here too, Python dictionaries save the day: The key:value pairs enable you to put *anything* in as the value, even functions and other dictionaries. So, we'll refactor our solution to store the function objects themselves in a dictionary of functions, linked to the string keys `'mean'`, `'median'`, and `'stddev'`. We will also make a `results` dictionary that will hold the dictionaries of the mean, median, and standard deviation results. That is, `results` will be a dictionary of dictionaries. This solution is:

```
import numpy as N
import glob

metrics = {'mean':N.mean, 'median':N.median,
           'stddev':N.std}
list_of_files = glob.glob("data*.txt")

results = {}                      #- Initialize results
for imetric in metrics.keys():    #  dictionary for each
    results[imetric] = {}         #  statistical metric

for ifile in list_of_files:
    data = readdata(ifile)
    for imetric in metrics.keys():
        results[imetric][ifile] = metrics[imetric](data)
```

This program is now generally written to calculate mean, median, and standard deviation *for as many files as there are* in the working directory that match `"data*.txt"` and can be extended to calculate *as many statistical metrics as desired*. If you want to access some other files, just change the search pattern in `glob`. If you want to add another statistical metric, just add another entry in the `metrics` dictionary. So, you just change two lines: *nothing else in the program needs to change.* This is what I mean when I say that Python enables you to write code that is more concise, flexible, and robust than in traditional languages. By my lights, this isn't just cool, but *way* cool ☺.

6.6 Exercises on using dictionaries and extending your basic data analysis routine

▷ **Exercise 18 (Dynamically filling a dictionary):**
Assume you're given the following list of files:

```
list_of_files = ['data0001.txt', 'data0002.txt',
                 'data0003.txt']
```

- Create a dictionary `filenum` where the keys are the filenames and the value is the file number (i.e., *data0001.txt* has a file number of 1) as an integer.
- Make your code fill the dictionary automatically, assuming that you have a list `list_of_files`.
- Hints: To convert a string to an integer, use the `int` function on the string, and the list and array sub-range slicing syntax also works on strings.

Solution and discussion: Here's my program to fill `filenum`:

```
filenum = {}
list_of_files = ['data0001.txt', 'data0002.txt',
                 'data0003.txt']
for ifile in list_of_files:
    filenum[ifile] = int(ifile[4:8])
```

▷ **Exercise 19 (Extend your data analysis routine to calculate skew and kurtosis):**

For the basic data analysis routine assignment given in this chapter, extend the last solution so that it also calculates the skew and kurtosis of each file's data. (Hint: NumPy has functions `skew` and `kurtosis` that do the calculations.)

Solution and discussion: Here's my extended program:

```
import numpy as N
import glob

metrics = {'mean':N.mean, 'median':N.median,
           'stddev':N.std, 'skew':N.skew,
           'kurtosis':N.kurtosis}
list_of_files = glob.glob("data*.txt")

results = {}                    #- Initialize results
for imetric in metrics.keys():  #  dictionary for each
    results[imetric] = {}       #  statistical metric

for ifile in list_of_files:
    data = readdata(ifile)
    for imetric in metrics.keys():
        results[imetric][ifile] = metrics[imetric](data)
```

That was easy! (Again, I left out the definition of `readdata` from this code, because it's just the same as in Solution One.)

6.7 Summary

In a traditional Fortran data analysis program, filenames, variables, and functions are all static. That is to say, they're specified by typing. Python data structures enable us to write dynamic programs, because variables are dynamically typed. In particular, Python dictionaries enable you to dynamically associate a name with a variable or function (or anything else), which is essentially what variable assignment does. Thus, dictionaries enable you to add, remove, or change a "variable" on the fly. The results are programs that are more concise and flexible. And fewer lines of code means fewer places for bugs to hide. Way cool!