Python in the Atmospheric Sciences: An Overview and Apologia for "New" Ways of Using Computers in Science

Johnny Wei-Bing Lin
Physics Department, North Park University
www.johnny-lin.com

Lives of Significance and Service
NORTH PARK UNIVERSITY CHICAGO

Slides version date: July 9, 2012. Author email: johnny@johnny-lin.com. Presented July 9th at the 2012 Unidata Workshop, Boulder, CO. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

Outline

- An overview of Python in the atmospheric sciences.
- An ultra-brief demonstration of UV-CDAT.
- An example of a basic Python data analysis routine: Why we should use modern data structures.
- An apologia for “new” ways of using computers in science.

An Overview of Python in the Atmospheric Sciences

Johnny Wei-Bing Lin
Physics Department, North Park University
www.johnny-lin.com

Lives of Significance and Service
NORTH PARK UNIVERSITY
CHICAGO

Slides version date: June 1, 2012. Author email: johnny@johnny-lin.com. Presented July 9th at the 2012 Unidata Workshop, Boulder, CO. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

Outline

- What is Python?
- Why Python is now gaining momentum in the atmospheric-oceanic sciences (AOS) community.
- Examples of how Python is used as an analysis, visualization, and workflow management tool.

What is Python?

- Structure: Is a scripting, procedural, fully native object-oriented (O-O) language. (Can also do some functional programming.)
- Interpreted: Loosely/dynamically-typed and interactive.
- Data structures: Robust built-in set and users are free to define additional structures.
- Array syntax: Similar to Matlab, IDL, and Fortran 90 (no loops!).
- Platform independent, open-source, and **free!**

Python's advantages

- Concise but natural syntax makes programs clearer. “Python is executable pseudocode. Perl is executable line noise.”
- Interpreted language makes development easier.
- Object-oriented nature makes code more robust/less brittle.
- Built-in set of data structures are very powerful and useful (e.g., dictionaries).
- Tight interconnects with compiled languages (Fortran via f2py and C via SWIG), so you can interact with compiled code when speed is vital.

Python's disadvantages

- Runs much slower than compiled code (but there are tools to overcome this).
- Relatively sparse collection of scientific libraries compared to Fortran (but this is growing).

Why AOS Python is now gaining momentum

- Python developed in the late 1980s; even as late as 2004, could be considered relatively “esoteric.”
- 2005, Python made a quantum jump in popularity.
- Now is ranked 8th in the May 2012 TIOBE Programming Community Index (which indicates the “popularity” of languages).

Image of TIOBE Index timeseries for Python is deleted for copyright reasons. Please see the URL for the image.

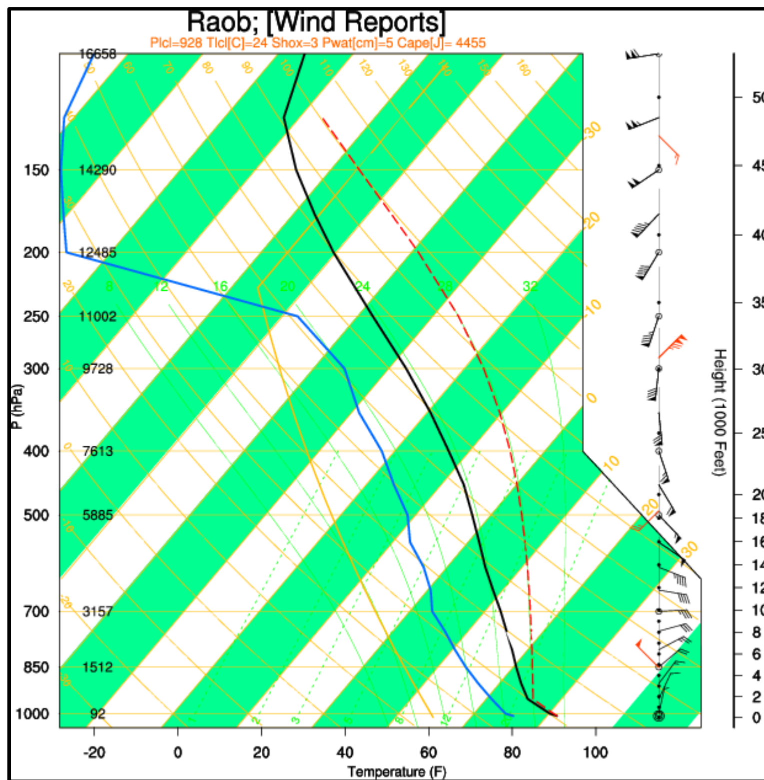
http://www.paulgraham.com/pypar.html, http://www.tiobe.com/index.php/paperinfo/tpci/Python.html, http://www.tiobe.com/content/paperinfo/tpci/index.html

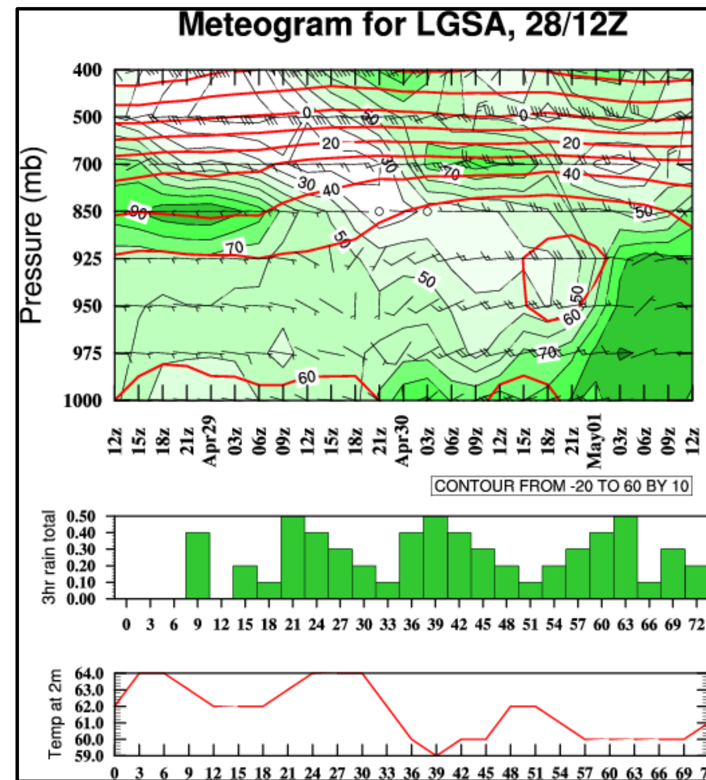
Why AOS Python is now gaining momentum (cont.)

- Around 2005, key tools for AOS use became available: NumPy, PyNGL, PyNIO, matplotlib added a contouring package.
- AOS institutional support in the AOS community now includes: LLNL PCMDI, NCAR CISL, AMS (annual meeting symposia and short courses).
- AOS Python users can now be found practically anywhere.
- An overview of AOS Python resources is found at the PyAOS website: http://pyaos.johnny-lin.com.

http://www.pyngl.ucar.edu/Images/history.png

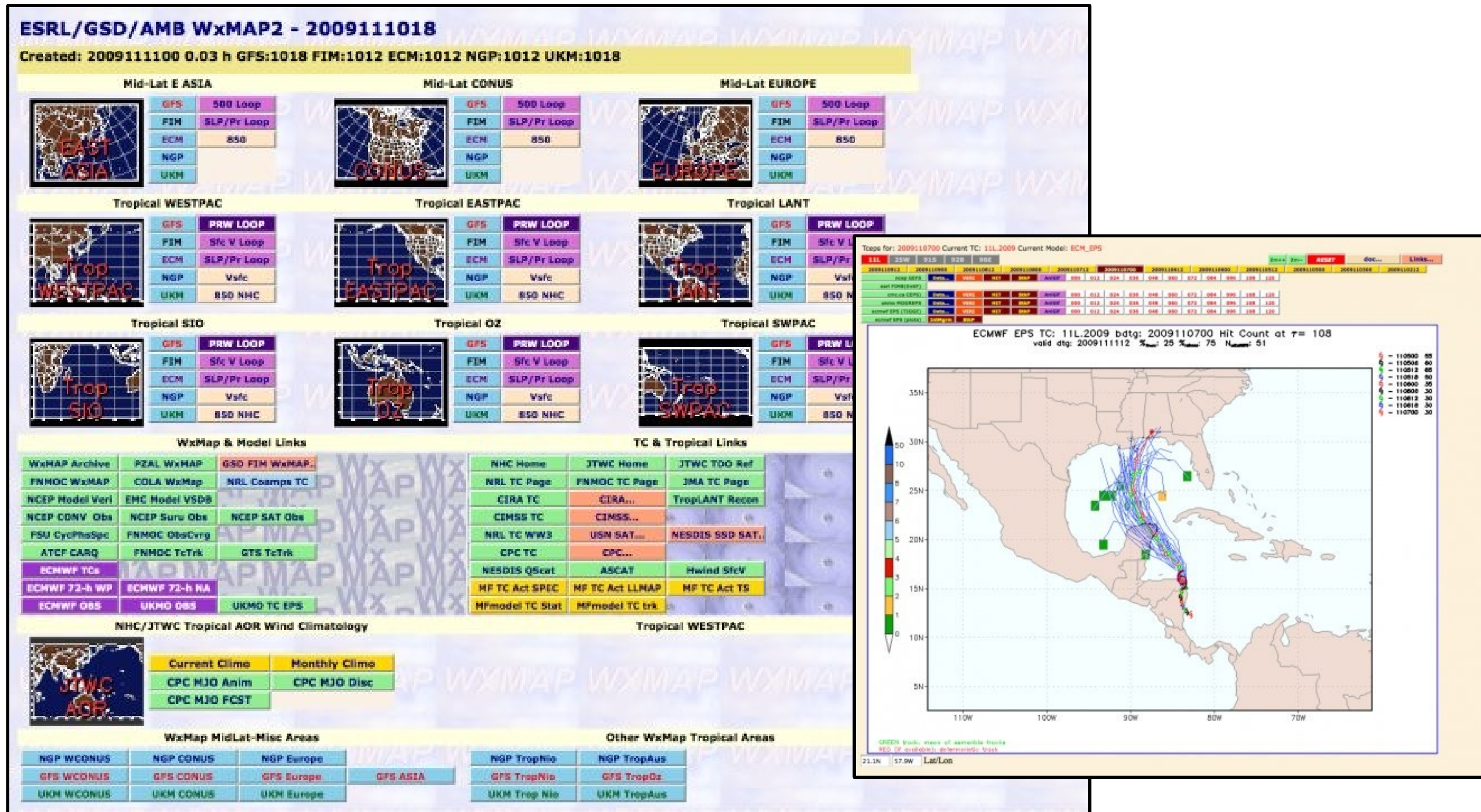
Example of visualization: Skew-T and meteograms





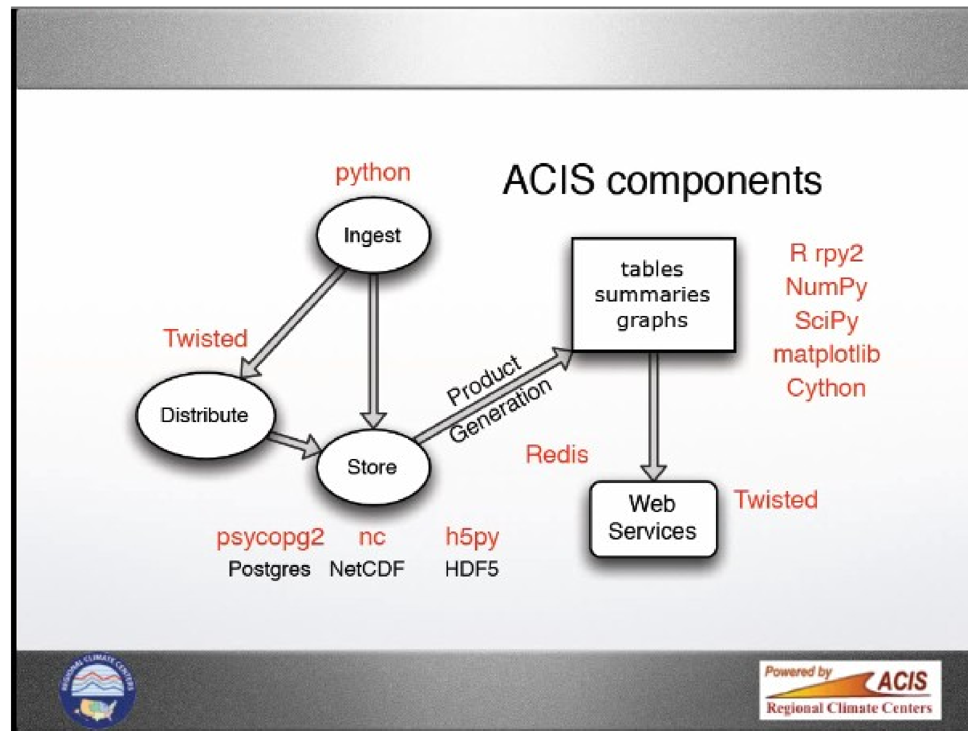
- All plots on this slide are produced by PyNGL and taken from their web site.
- See http://www.pyngl.ucar.edu/Examples/gallery.shtml for code.

Example of visualization and delivery of weather maps of NWP model results



- Screenshots taken from the WxMAP2 package web site.
- http://sourceforge.net/projects/wxmap2/

Example of analysis, visualization, and workflow management and integration



- Problem: Many different components of the Applied Climate Information System: Data ingest, distribution, storage, analysis, web services.
- Solution: Do it all in Python: A single environment of shared state vs. a crazy mix of shell scripts, compiled code, Matlab/IDL scripts, and web server makes for a more powerful, flexible, and maintainable system.

Image from: 2011 AMS talk by Bill Noon, Northwest Regional Climate Center, Ithaca, NY, http://ams.confex.com/ams/91Annual/flvgateway.cgi/id/17853?recordingid=17853

Conclusions

- Python is a mature, comprehensive computational environment for all aspects of the atmospheric and oceanic sciences (AOS).
- Python is growing in its adoption by the AOS community.
- And it's (mostly) all free!

An Ultra-Brief Demonstration of UV-CDAT

Johnny Wei-Bing Lin
Physics Department, North Park University
www.johnny-lin.com

Acknowledgments: Many of these slides are copied or adapted from a set by Dean Williams and Charles Doutriaux (LLNL PCMDI). Thanks also to the online CDAT/UV-CDAT documentation and Alex DeCaria (Millersville Univ.).

Lives of Significance and Service
NORTH PARK UNIVERSITY CHICAGO

Slides version date: June 1, 2012. Author email: johnny@johnny-lin.com. Presented July 9th at the 2012 Unidata Workshop, Boulder, CO. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

Outline

- What is UV-CDAT?
- Masked arrays and masked variables.
- A demonstration of the UV-CDAT GUI.

What is UV-CDAT?

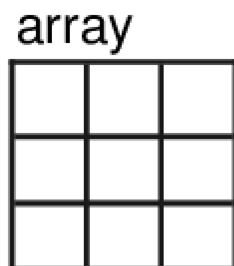
- Written by LLNL PCMDI under a BSD open source license.
- Unified environment based on the object-oriented Python computer language.
- Integrated with packages that are useful to the atmospheric sciences community:
  - Climate Data Management System (cdms2): netCDF file access, regridding, etc.
  - NumPy, masked array (ma), masked variable (MV2).
  - Visualization (vcs, Xmgrace, matplotlib, VTK, Visus, etc.).
  - And more! (e.g., time axis alignment, OPeNDAP, ESG, etc.).
- Graphical user interface (VCDAT).
- XML representation (CDML/NcML) for data sets.
- URL: http://www-pcmdi.llnl.gov/software-portal.

Masked arrays and masked variables

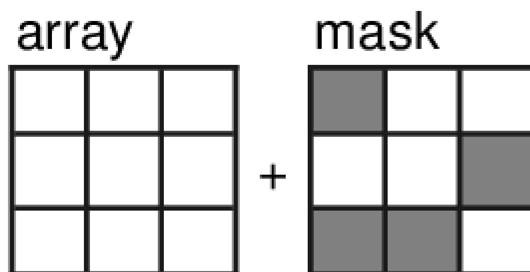
- Python supports array variables (via NumPy).
- All variables in Python are not technically variables, but objects:
  - Objects hold multiple pieces of data as well as functions that operate on that data.
  - For AOS applications, this means data and metadata (e.g., grid type, missing values, etc.) can both be attached to the "variable."
- Using this capability, we can define not only arrays, but two more array-like variables: masked arrays and masked variables.
- Metadata attached to the arrays can be used as part of analysis, visualization, etc.
- UV-CDAT automatically uses the metadata to reduce analysis operations to one-liners or drag-and-drop.

Schematic of arrays, masked arrays, and masked variables

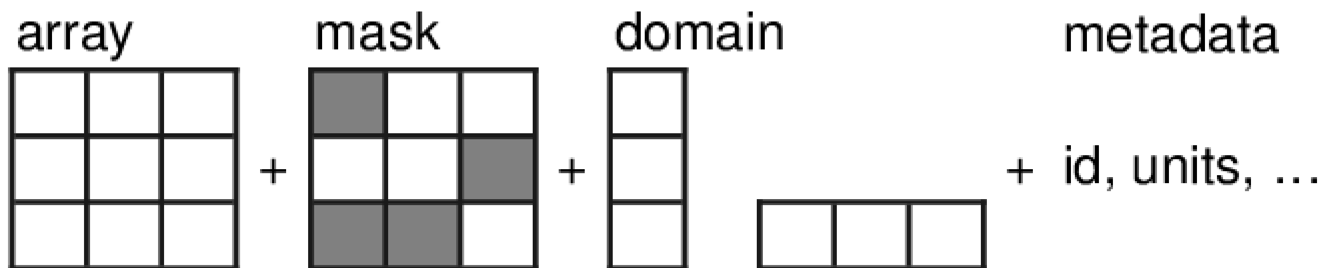
Arrays:
(numpy)



Masked Arrays:
(numpy.ma)



Masked Variables:
(MV2)



A demonstration of the UV-CDAT GUI

- Read in data (local).
- Select a region of interest.
- Select a time slice of the variable.
- Plot a longitude-latitude slice.
- As a tool for teaching UV-CDAT and Python.

An Example of a Basic Python Data Analysis Routine: Why We Should Use Modern Data Structures

Johnny Wei-Bing Lin
Physics Department, North Park University

July 9, 2012

Lives of Significance and Service
NORTH PARK UNIVERSITY
CHICAGO

Slides version date: June 1, 2012. Presented July 9th at the 2012 Unidata Workshop, Boulder, CO. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

Outline

Some Python collection data types

A data analysis problem

Solution One: Fortran-like structure with several loops

Solution Two: Store results in arrays

Solution Three: Store results in dictionaries

Solution Four: Store results and functions in dictionaries

Conclusions

Some Python collection data types I

- Lists: Ordered sequences referenced by index:

```
>>> a = [1, 4.5, 'hello']
>>> print a[1]
4.5
>>> print a[2]
hello
```

Some Python collection data types II

- Arrays: Multidimensional ordered sequences where all elements have the same type:

```
>>> import numpy as N
>>> a = [1, 4.5, 'hello']
>>> b = N.array(a)
>>> print b
['1' '4.5' 'hello']
>>> c = N.zeros(N.shape(b))
>>> print c
[ 0.  0.  0.]
```

Some Python collection data types III

- Dictionaries: Unordered sequences referenced by keys. Keys are anything that can be uniquely sorted; often strings or integers:

```
>>> import numpy as N
>>> a = {'a':1, 3:4.5, 'c':'hello'}
>>> print a[3]
4.5
>>> print a['a']
1
>>> a['dd'] = 'bye'
>>> print a
{'a': 1, 'c': 'hello', 3: 4.5, 'dd': 'bye'}
```

A data analysis problem

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). Write a program that:

- Reads in the data for 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.

Solution One: Fortran-like structure with several loops I

On the next slide is a solution that puts all the file open, closing, read, and conversion into a function, so you don't have to type `open`, etc., three times. The way it's written, however, looks very Fortran-esque, with variables initialized and/or created explicitly (e.g., from a function call).

Solution One: Fortran-like structure with several loops II

```
import numpy as N

def readdata(filename):
    fileobj = open(filename, 'r')
    outputstr = fileobj.readlines()
    fileobj.close()
    outputarray = N.zeros(len(outputstr), dtype='f')
    for i in xrange(len(outputstr)):
        outputarray[i] = float(outputstr[i])
    return outputarray

data1 = readdata('data0001.txt')
data2 = readdata('data0002.txt')
data3 = readdata('data0003.txt')
```

Solution One: Fortran-like structure with several loops III

```
mean1 = N.mean(data1)
median1 = N.median(data1)
stddev1 = N.std(data1)

mean2 = N.mean(data2)
median2 = N.median(data2)
stddev2 = N.std(data2)

mean3 = N.mean(data3)
median3 = N.median(data3)
stddev3 = N.std(data3)
```

Solution One: Fortran-like structure with several loops IV

- We haven't really taken much advantage of anything unique to Python. The program is 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.

Solution Two: Store results in arrays I

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, reading in the data, and making the calculations:

- This means you don't have to type in the names of every mean, etc. variable to do the assignment.
- Using Python's powerful string type to create the filenames makes this approach even easier.

Solution Two: Store results in arrays II

```
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)
```

Solution Two: Store results in arrays III

This code is more compact and scales up to any `num_files` number of files. But I'm still bothered by two things:

- What if the filenames aren't numbered? How then do you relate the element position of the `mean`, etc. arrays to the file the quantity is calculated using? Variable names (e.g., `mean1` *do* convey information and connect that label to a value.
- Why should I pre-declare the size of `mean`, etc.? If Python is dynamic, shouldn't I be able to arbitarily change the size of `mean`, etc. on the fly as the code executes?

Solution Three: Store results in dictionaries I

How are dictionaries useful here?:

- We previously said variable names connect labels to values. What does that mean? That a string (the variable name) is associated with a value (scalar, array, etc.).
- What do we know in Python that can associate a string with a value? A dictionary.
- So, setting a value to a key that is the variable name (or something similar) is effectively the same as setting a variable.
- But this can be done dynamically (i.e., you don't have to type it in) and can accomodate *any* string, not just those numbered numerically.

Solution Three: Store results in dictionaries II

Here 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)
```

Solution Three: Store results in dictionaries III

Comments on 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 get access a directory listing of data files, we could generate the list automatically. I can, in fact, do this in Python with the `glob` module:

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

  You can sort `list_of_files` using list methods or some other sorting function.
- Statistical values are referenced intelligently: To access, say, the mean of *data0001.txt*, type in `mean['data0001.txt']`.

Solution Four: Store results and functions in dictionaries I

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?

Yes! Dictionaries are the key: The key:value pairs enable you to put *anything* in as the value, even functions and other dictionaries. So:

- Store the function objects themselves in a dictionary of functions, linked to the keys `'mean'`, `'median'`, and `'stddev'`.
- Make a `results` dictionary that will hold the dictionaries of the mean, median, and standard deviation results. That is, `results` is a dictionary of dictionaries.

Solution Four: Store results and functions in dictionaries II

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

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 there are in the working directory that match `"data*.txt"` and can be extended to calculate as many statistical metrics as desired.

Conclusions

- In a traditional Fortran data analysis program, filenames, variables, and functions are all static: They're specified by typing.
- Python data structures enable us to write dynamic programs, because variables are dynamically typed.
- 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.

Acknowledgments: Thanks to Yun-Lan Chen and her Central Weather Bureau (Taiwan) colleagues and PyAOS commenter "N eil" (`http://pyaos.johnny-lin.com/?p=755&cpage=1#comment-119`) for discussions and suggestions.

An Apologia for “New” Ways of Using Computers in Science

Johnny Wei-Bing Lin
Physics Department, North Park University
www.johnny-lin.com

Acknowledgments: Portions are taken from a talk co-authored with Tyler Erickson (MTRI). Thanks to Ricky Rood and Jeremy Bassis at the University of Michigan for discussions.

Lives of Significance and Service
NORTH PARK UNIVERSITY CHICAGO

Slides version date: July 9, 2012. Author email: johnny@johnny-lin.com. Presented July 9th at the 2012 Unidata Workshop, Boulder, CO. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

How we've traditionally used computers to do science

- We mainly care about two things, when it comes to computers:
  - Speed: Run longer simulations of larger and more complex models.
  - Results: Just make it work to get a scientific result.
- We've often ignored best-practices from software engineering and the open-source community. Thus, we have:
  - Code that seldom gets used (and often cannot be used) by anyone besides the original author.
  - Code that receives limited testing and is brittle.
  - Science that is functionally irreproducible.
- Computationally, we are insular.

How insular are we?

- We are so insular that we use languages no one else uses:
  - Outside users cannot use or test our code.
  - Code innovations created by others are unavailable to us: Fewer synergies are possible.
- Computational power and tools have exploded outside of our community: We can't access the results of that explosion.

Language	Rank	Rating
C	1	17.346%
Java	2	16.599%
C++	3	9.825%

Language	Rank	Rating
Matlab	23	0.485%
Fortran	26	0.411%
IDL	Not in top 100	N/A

(top) The 3 most popular languages. (bott) Popularity of some languages used in the computational earth sciences. Data from the TIOBE Programming Community Index for May 2012.

An apologia for “new” ways of using computers in science

- We've seen how Python's modern constructs help us write code that is:
  - Clearer.
  - More powerful.
  - Can use packages developed by non-scientists (e.g., webservices).
- Such code improvements aren't just “nice extras”: They lead to **better science and more science:**
  - More reliable and reproducible results.
  - Clear code $\rightarrow$ ask and answer additional science questions (Nick Barnes, 2012).

An apologia for “new” ways of using computers in science (cont.)

- Valuing and writing clear and flexible code is one best-practice we need to adopt.
- Three critical strategies from software engineering and open-source best practices we also need to adopt:
  - Unit testing and code review (also TDD, Agile, etc.).

Commercial example: Flickr and testing

2009: 3 billion photos, 40,000 photos per second
2012: 5 billion+ photos ...

Image of the Flickr main page
deleted for copyright reasons.
Please see URL below for image.

http://www.flickr.com; Allspaw & Hammond (2009),
http://code.flickr.com/blog/2009/06/26/slides-from-velocity-2009/.
Hat tip: Neal Ford

Commercial example: Flickr and testing (cont.)

http://code.flickr.com/ (May 21, 2012)

Image of code contributors deleted for copyright reasons. Please see URL below for image, at the bottom of the web page.

Last week averaged **12+ deploys per day**

"One step build and deploy"

Allspaw & Hammond (2009), http://code.flickr.com/blog/2009/06/26/slides-from-velocity-2009/

An apologia for “new” ways of using computers in science (cont.)

- Valuing and writing clear and flexible code is one best-practice we need to adopt.
- Three critical strategies from software engineering and open-source best practices we also need to adopt:
  - Unit testing and code review (also TDD, Agile, etc.).
  - Social coding: Community development method that supports code improvement by lowering the barriers to access and changing (e.g., GitHub).

    > *“The advantages of multiple codebases are similar to the advantages of mutation: they can dramatically accelerate the evolutionary process by parallelizing the development path.”* (Stephen O'Grady, 2010)

  - Open API: Synergies come from tools that can easily talk to each other.
- Achieving these goals requires we take a code *management,* not just writing, approach.

Seven issues in code management

1) Distribution: How can you make the code available to others?
2) Documentation: How do you describe the code so that others can understand it?
3) Advertising: How do you make sure others can “find” the code?
   - Discover the code exists
   - Realize the code can be applied to their particular problem
4) Instruction: How do you make sure others have the skills that are needed to use the code?
5) Evaluation: How do you learn how your code compares to others people's code?
6) Improvement and feedback: Are their mechanisms to enable users to take your code, use it, improve it, and return those results to the community?
7) Sustainability: Are there (dis)incentives to make code management more (difficult)easy to implement?

Conclusions

- Python exemplifies clear code that results in better and more science.
- Adopting best-practices from software engineering and the open-source community can help us do better and more science.
- AMS 2013 Python Symposium and Short Courses:

  http://annual.ametsoc.org/2013/
- PyAOS: Tips, announcements, and growing a Python community in the atmospheric and oceanic sciences:

  http://pyaos.johnny-lin.com