Dynamic Data Structures and First-Class Citizens: Python Features That Can Make Data Analysis More Flexible and Powerful

Johnny Wei-Bing Lin
University of Washington Bothell and North Park University

July 27, 2016

Slides version date: July 30, 2016. Presented at the UCLA Department of Atmospheric and Oceanic Sciences, Los Angeles, CA.

Outline

What is Python?

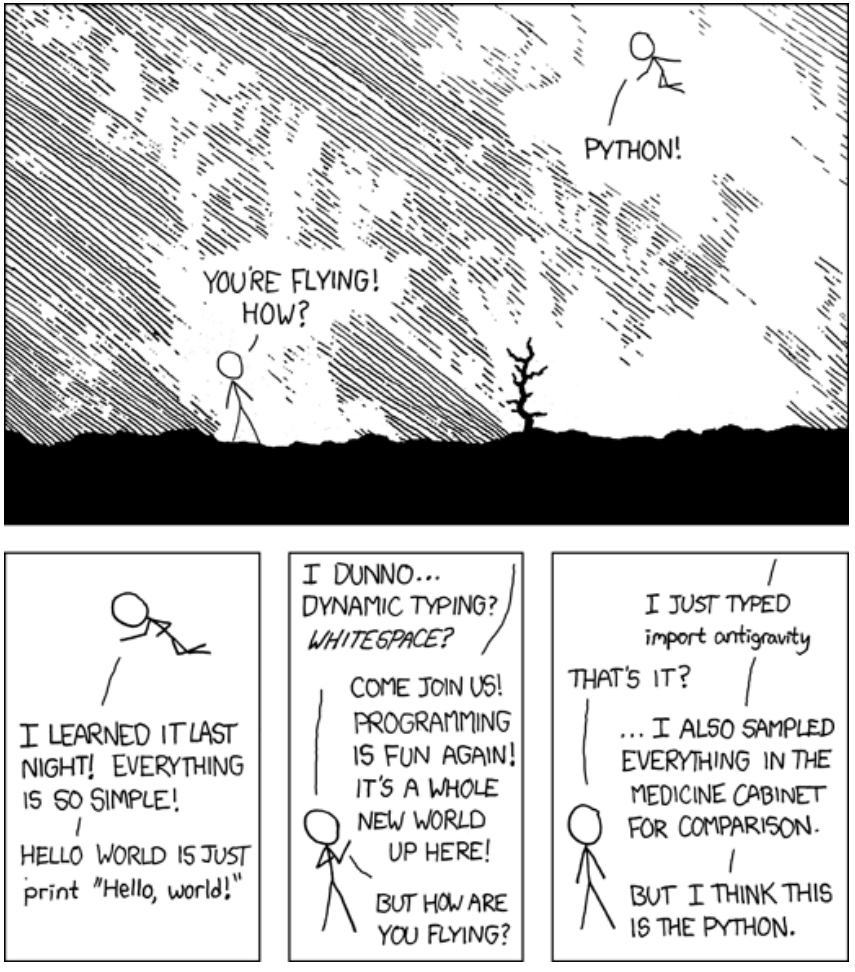
- Multi-paradigm: Scripting, procedural, and fully native object-oriented (O-O) language.
- Interpreted: Loosely/dynamically-typed and interactive.
- Array syntax: Similar to MATLAB, IDL, and Fortran 90 (no loops!).
- Platform independent, open-source, and **free!**

Python's advantages I

- Concise but natural syntax, both arrays and non-arrays, makes programs clearer. "Python is executable pseudocode. Perl is executable line noise."
- No compiling and no declaring variables makes development easier.
- Object-orientation makes code more robust/less brittle.

Python's advantages II

- Tight interconnects with compiled languages (Fortran via f2py and C via SWIG) when speed is vital.
- Many packages available that are developed by non-scientists (e.g., webservices).
- Built-in set of data structures are very powerful and useful (e.g., dictionaries).

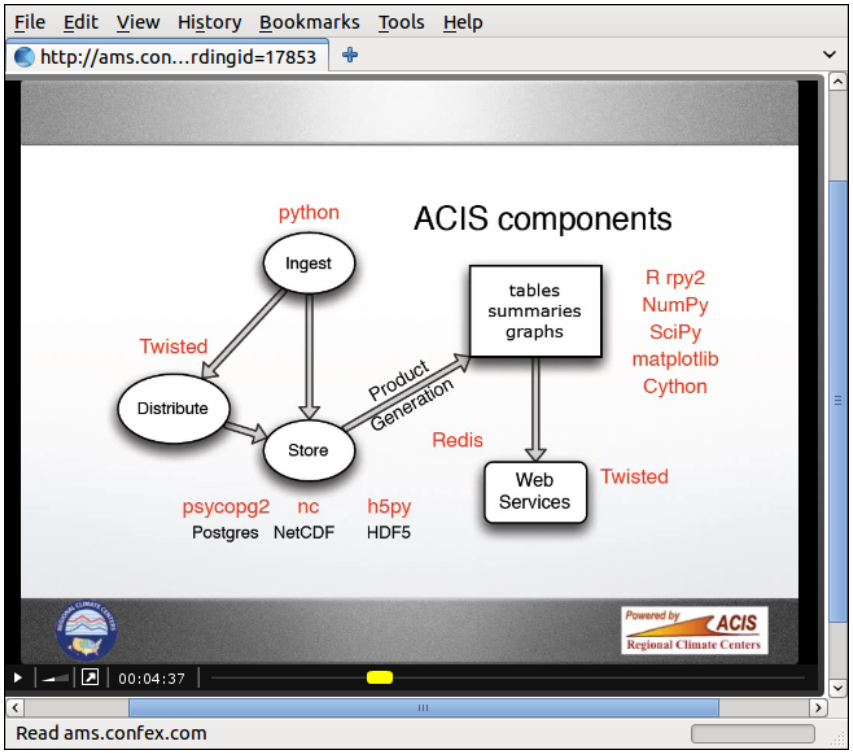


(http://xkcd.com/353/)

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).

Example of analysis, visualization, and workflow management and integration



- Problem: Many different components of the Applied Climate Information System.
- 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: AMS talk by Bill Noon, Northeast Regional Climate Center, Ithaca, NY, `http://ams.confex.com/ams/91Annual/flvgateway.cgi/id/17853?recordingid=17853`.

Prologue Data Analysis Exercise

Prologue data analysis exercise I

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. You also the following `readdata` function that reads each file and puts the file's contents into a 1-D NumPy array:

```
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
```

Prologue data analysis exercise II

Thus,

```
data1 = readdata('data0001.txt')
```

will give you a 1-D NumPy array `data1` that contains all the values in the file *data0001.txt*.

Prologue data analysis exercise III

Our assignment: Write a program that:

- Reads in the data from each file using the `readdata` function above and then
- Calculates the mean, median, and standard deviation of the values in each data file, for each data file, saving the values to variables for possible later use.

Review of Python Lists, Dictionaries, and Functions

Lists and tuples I

- Lists are ordered sequences.
- They are like arrays, except each of the items do not have to be of the same type. A given list element can be set to anything, even another list.
- Square brackets ("`[]`") start and stop (delimit) a list.
- Put a comma between list elements. If you have a one element list, put a comma after the element.
- List element addresses start with zero, so the first element of list `a` is `a[0]`, the second is `a[1]`, etc.
- Length of a list is obtained using the `len` function, e.g., `len(a)`.

Lists and tuples II

- Say you typed the following in the Python interpreter:

  ```
  a = [2, 3.2, 'hello', [-1.2, 'there', 5.5]]
  ```

  What is `len(a)`? What does `a[1]` equal to? `a[3]`? `a[3][1]`? Share your answers with your neighbor.
- Elements can also be addressed starting from the end. `a[-1]` is the last element in list `a`, is the next to last element, etc.
- Slicing a list:
  - Element addresses in a range are separated by a colon.
  - The lower limit is **inclusive,** and the upper limit is **exclusive.**
  - For the list `a` you just typed in, what would `a[1:3]` return?
- Lists are mutable (i.e., you can add and remove items, change the size of the list) while tuples are immutable.

Lists and tuples III

- Tuples use parenthesis as delimiters, e.g.:

```
b = (3.2, 'hello')
```

- Note: You can, to an extent, treat strings as lists. Thus, if `a = "hello"`, then `a[1:3]` will return `"el"`.

Dictionaries I

- Dictionaries are **unordered** lists whose elements are referenced by **keys** (not by position).
- Keys can be anything that can be uniquely named and sorted. In practice, keys are usually integers or strings. Values can be anything.
- Curly braces ("{}") delimit a dictionary. Dictionary elements are key:value pairs, separated by a colon.
- Dictionaries are very powerful. This one data structure revolutionized my code.

Dictionaries II

- Say you typed the following in the Python interpreter:

```
a = {'a':2, 'b':3.2, 'c':[-1.2, 'there', 5.5]}
```

- Dictionary elements are referenced like lists, except the key is given in place of the element address.
- For the previous dictionary:
  - What does `a['b']` return?
  - What does `a['c'][1]` return?

Functions I

- Functions in Python, in theory, work **both** like functions and subroutines in Fortran, in that:
  - Input comes via arguments.
  - Output occurs through:
    - A return variable (like Fortran functions) and/or
    - Through arguments (like Fortran subroutines)
- In practice, functions in Python are written to act like Fortran functions, with a single output returned. If you want multiple returns, it's easier to put them into a list or use objects.
- Function definitions:
  - Begin with `def`.
  - Contents of the function after the `def` line are indented in "x" spaces (where "x" is constant). Usually, people indent 4 spaces.

Functions II

- Arguments:
  - Python accepts both positional and keyword arguments:
  - Positional arguments are usually for *required* input.
  - Keyword arguments are usually for *optional* input.
  - Typically, keyword arguments are set to some default value.
- Example of a function definition and calls:

```
def area(radius, pi=3.14):
    area = pi * (radius**2)
    return area
a = area(3)
b = area(3, pi=3.1415)
```

- Python also has a nifty way of passing in lists of arguments and keywords (as a list/tuple and dictionary, respectively):

Functions III

```
args = [3,]
kwds = {'pi':3.1415}
a = area(*args, **kwds)
```

First-Class Citizenship in Python

What's a first-class citizen? I

Definition: From Wikipedia:[1]

> *In programming language design, a first-class citizen (also object, entity, or value) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as a parameter, returned from a function, and assigned to a variable.*

What's a first-class citizen? II

Examples of first-class citizens in Fortran 77:

- Yes: integer, real, character, arrays.
- No: Functions, subroutines, programs, libraries.

Examples of first-class citizens in Python: **Basically everything.**

[1] `https://en.wikipedia.org/w/index.php?title=First-class_citizen&oldid=653520885`

Functions can be set as variables in Python I

Consider this area function:

```
def area(radius, pi=3.14):
    return pi * (radius**2)
```

Say the following lines of code are executed:

```
print( area(3) )
myarea = area
print( myarea(3) )
set = { 'value':3, 'area':area, 'area2':myarea }
print( set['area'](3) )
print( set['area'](set['value']) )
print( set['area2'](set['value']) )
```

Question: Predict what will occur, then share with your answer with your neighbor.

Functions can be set as variables in Python (solution) I

The following is output:

```
28.26
28.26
28.26
28.26
28.26
```

All the calling references given above are the same!

Functions can be set as variables in Python (solution) II

Lessons from this example:

- Functions (and modules) are like any other object or variable and can be stored as a variable or in any appropriate data structure. **Any reference to a function** of whatever "kind" (e.g., a list entry) can be called (if callable).
- Dictionaries and lists (and calling) are **mutable at runtime.** Thus, you don't have to know ahead of the runtime what functions you will use. You can have **your program choose your functions automatically while the program is running.**

Functions can be passed as parameters in Python I

Consider the following Newtonian heating/cooling model of an object:[2]

$$\frac{dT}{dt} = k(T - T_{\text{env}})$$

where the T of the object is in K, t is in hrs, and the environmental temperature is T_{env}. k is a constant.
Below I code two different functions for dT/dt for two different sets of k and T_{env}, and then I write a function to solve for T using Euler's method (ugly, I know).

Functions can be passed as parameters in Python II

```
import numpy as N

def dTdt_one(T):
    k = -1.335
    return k * (T - 25.)

def dTdt_two(T):
    k = -2.0
    return k * (T - 35.)

def calculate_temps( dTdt, start=0.0, stop=5.0,
                     delta_t=0.001, T0=6.0 ):
    num_pts = N.ceil(((stop - start) / delta_t) + 1)
    times = (N.arange(num_pts) / (num_pts - 1.0) * end)
          + start
```

Functions can be passed as parameters in Python III

```
    temps = N.zeros(N.shape(times), dtype='d')
    temps[0] = T0

    for i in xrange(1, N.size(times)):
        Told = temps[i-1]
        temps[i] = Told + (dTdt(Told) * delta_t)

    return (times, temps)

calculate_temps(dTdt_one)
calculate_temps(dTdt_two)
```

Question: What will happen with the two `calculate_temps` calls?

---

[2] A problem in Shiflet & Shiflet (2014).

Functions can be passed as parameters in Python (solution) I

- This example is silly: No one writes a new function when you change some constants (k and T_{env}, in this case).
- But you can imagine defining a more complicated function for dT/dt. Or importing a function from another module: Since functions are variables, once you have a function name in your namespace, you can pass it in as a parameter.
- The key: `dT/dt` in `calculate_temps` is a parameter. **It is substituted at runtime** at the `calculate_temps` call.

Functions can be passed as parameters in Python (solution) II

- You do not have to hardwire in your function calls. I could have just as easily make the calls in a loop:

```
for i in list_of_dTdts:
    calculate_temps(i)
```

  where the list `list_of_dTdts` is mutable at runtime. So, as the program is running, it might make `list_of_dTdts` have 5 items one time then 1000 items the other, depending on what's going on in the rest of the program.

- If you're concerned about dealing with different function argument lists, just code your calls with the general `*args` and `**kwds` parameter lists feature in Python.

Solutions To Our Prologue Data Analysis Exercise: Applying Dynamic Structures and First-Class Citizens to Data Analysis

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).

Note: I assume `readdata` is defined earlier in the code.

Solution One: Fortran-like structure with several loops II

```
import numpy as N

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

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)
```

Solution One: Fortran-like structure with several loops III

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

Solution One: Fortran-like structure with several loops IV

- With programs written Fortran-style, 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 predeclare 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 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.

An Implication for Modeling

Make subroutine execution arbitrary at runtime I

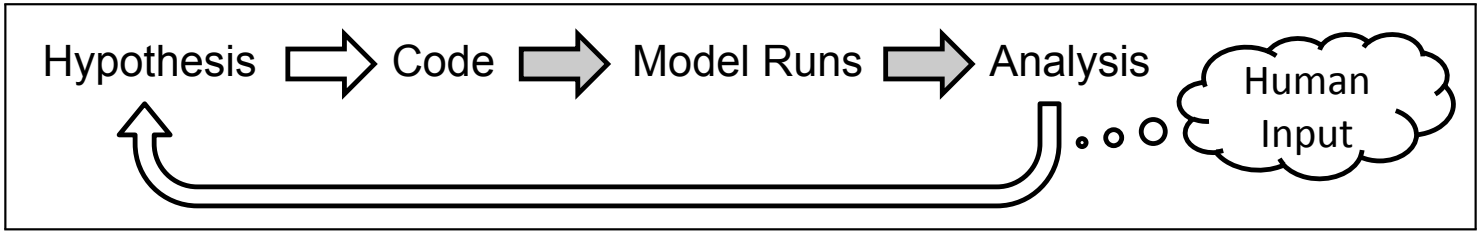
- Dictionaries and lists are mutable at runtime: They can **change order, content, and size as the program runs.**
- If we store functions in dictionaries and lists, we can run them in any order and have the program automatically change that order **as the program runs.**
- Example from the qtcm hybrid Python-Fortran package (Lin 2009):

```
>>> model = Qtcm(compiled_form='parts')
>>> print model.runlists['qtcminit']
['__qtcm.wrapcall.wparinit', '__qtcm.wrapcall.wbndinit',
'varinit', {'__qtcm.wrapcall.wtimemanager': [1,]},
'atm_physics1']
```

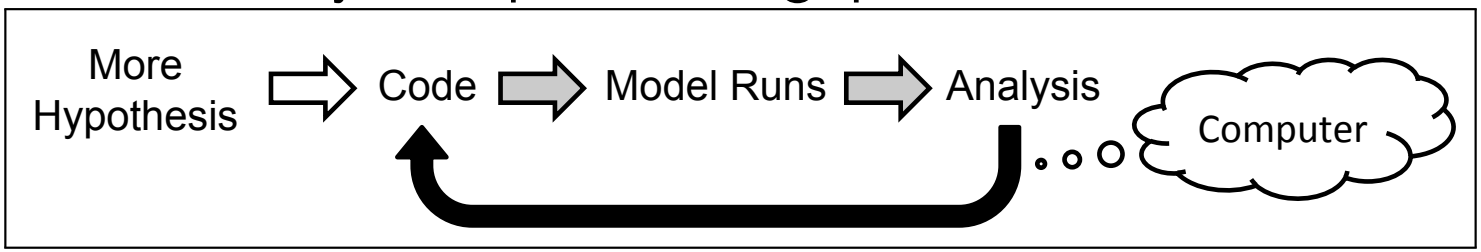
- Routines here are specified by strings that link to the function object. They are stored in a list I can change while the program runs.

Transforming the modeling and analysis cycle for climate modeling studies (Lin 2009)

Traditional analysis sequence used in modeling studies:



Transformed analysis sequence using qtcm-like tools:



Outlined arrows = mainly human input.
Gray-filled arrows = a mix of human and computer-controlled input.
Completely filled (black)-arrows = purely computer-controlled input.

For more information and an advertisement

- *Geosci. Model Dev.* paper on qtcm:
  `http://www.geosci-model-dev.net/2/1/2009/`
- The qtcm Python package website:
  `http://www.johnny-lin.com/py_pkgs/qtcm/`
- The Neelin-Zeng QTCM1 website:
  `http://www.atmos.ucla.edu/~csi/QTCM/qtcm.html`
- Interested in learning Python or growing the atmospheric-oceanic sciences Python community? Come join PyAOS:
  `http://pyaos.johnny-lin.com`
- Ad for my book: *The Nature of Environmental Stewardship*, (Pickwick Publications, 2016):
  `http://nature.johnny-lin.com`

Conclusions

- In a traditional Fortran data analysis program, filenames, variables, and functions are all static: They're specified by typing.
- 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.
- Python data structures and first-class citizenship to nearly all entities enable us to write dynamic data analysis (and modeling) programs.