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

Johnny Wei-Bing Lin
Computing and Software Systems Division,
University of Washington Bothell
Physics and Engineering Department, North Park University

February 2, 2016

Lives of Significance and Service
NORTH PARK UNIVERSITY CHICAGO

W

Slides version date: February 2, 2016. Presented at the Python in the Geosciences seminar, University of Washington Seattle, WA. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License.

Outline

Review of Python Lists, Dictionaries, and Functions

First-Class Citizenship in Python

Re-imagining Model Management

Conclusions

For more information

Review of Python Lists, Dictionaries, and Functions

Lists and Dictionaries

- 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.
- 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.

Review of Functions

- Functions in Python accept input via arguments and outputs via a return variable.
- Python also has a nifty way of passing in lists of arguments and keywords (as a list/tuple and dictionary, respectively):

```
args = [3,]
kwds = {'pi':3.14}
a = area(*args, **kwds)
print(a)
```

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']) )
```

What will occur?

Functions Can Be Set as Variables in Python II

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 III

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 programming is running.**

Functions Can Be Passed as Parameters in Python I

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

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

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)
```

Functions Can Be Passed as Parameters in Python IV

What this example illustrates:

- The parameter `dTdt` in `calculate_temps` can be anything, including a function.
- The parameter `dTdt` is **substituted at runtime** at the `calculate_temps` call.
- 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.

Functions Can Be Passed as Parameters in Python V

- 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.

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

Re-imagining Model Management

Traditional vs. Modern Model Management

With traditional compiled languages:

- Static variable namespace management: Values are assigned at compile time. You hard code it in (or store numbers in arrays, or read it from a file).
- Static subroutine execution order: Set at compile time. Again, you hard code it in.

But with Python's dynamic data structures and first-class citizens structure:

- Neither needs to be static.
- We can use dictionaries and/or Python objects to manage both.

The `qtcm` hybrid Python-Fortran intermediate-level atmospheric model implements these features.

The Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (QTCM1)

Neelin & Zeng (1999) and Zeng et al. (1999)

- Intermediate-level atmospheric model.
- Vertical temperature and moisture profiles based upon convective quasi-equilibrium assumption.
- Betts & Miller (1986) moist convective adjustment scheme.
- Includes radiative-convective feedback package.
- Resolution 5.625 deg longitude, 3.75 deg latitude.
- Reasonable simulation of tropical climatology, and also includes Madden-Julian oscillation (MJO)-like variability. Used in MJO studies, ENSO studies, etc.
- Written in Fortran.

Overview of the Python qtcm package

- Software infrastructure:
  - Fortran: Numerics of QTCM1
  - Python: User-interface wrapper that manages variables, routine execution order, runs, and model instances.
  - Connectivity: Through the program f2py:
    - Almost automatically makes the Fortran routines and memory space available to Python.
    - You can set Fortran variables at the Python level, even at run time.
- Two main classes of objects:
  - `Field`: Key model variable and parameters.
  - `Qtcm`: A model instance.

A simple qtcm run

```
from qtcm import Qtcm
inputs = {}
inputs['runname'] = 'test'
inputs['landon'] = 0
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 30
inputs['mrestart'] = 0
inputs['compiled_form'] = \
    'parts'
model = Qtcm(**inputs)
model.run_session()
```

- Configuration keywords:
  - Output filenames will contain the string given by `runname`.
  - Aquaplanet (set by `landon`).
  - Start from Nov 1, Year 1. Run for 30 days.
  - Start from a newly initialized model state.
- Run the model using the `run_session` method.
- `compiled_form` chooses the model version.

Run sessions and a continuation run in qtcm

```
model = Qtcm(**inputs)
model.run_session()
model.u1.value = model.u1.value * 2.0
model.init_with_instance_state = True
model.run_session(cont=30)
```

- Make one run session, double the value of `u1`, make a continuation run for 30 more days.
- All can be controlled interactively at runtime. (I could have used `getattr('u1')` to manage the `u1` attribute, for instance.)

Multiple qtcm model runs using a snapshot from a previous run session

```
model.run_session()
mysnap = model.snapshot

model1.sync_set_py_values_to_snapshot(snapshot=mysnap)
model2.sync_set_py_values_to_snapshot(snapshot=mysnap)
model1.run_session()
model2.run_session()
```

- Snapshots are dictionaries that act as restart files.
- `model1` and `model2` are separate instances of the `Qtcm` class and are truly independent (they share no variables or memory).

Runlists in qtcm make the model very modular

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

- Run lists specify a series of Python or Fortran methods, functions, subroutines (or other run lists) to execute when the list is passed into a call of the `run_list` method.
- Routines in run lists are identified by strings. What routines the model executes are fully changeable at run time.
- Example shows a list with two Fortran subroutines without input parameters, a Python method without input parameters, a Fortran subroutine with an input parameter, and another run list.

qtcm performance is competitive with the Fortran-only QTCM1

System	Fortran-Only QTCM1	Python qtcm Package
Mac OS X: MacBook 1.83 GHz Intel Core Duo running Mac OS X 10.4.10	152.59	158.94
Ubuntu GNU/Linux: Dell PowerEdge 860 with 2.66 GHz Quad Core Intel Xeon processors (64 bit) running Ubuntu 8.04.1 LTS	43.73	47.45

Performance penalty of hybrid-languge model vs. the Fortran-only version of the model is 4–9%.

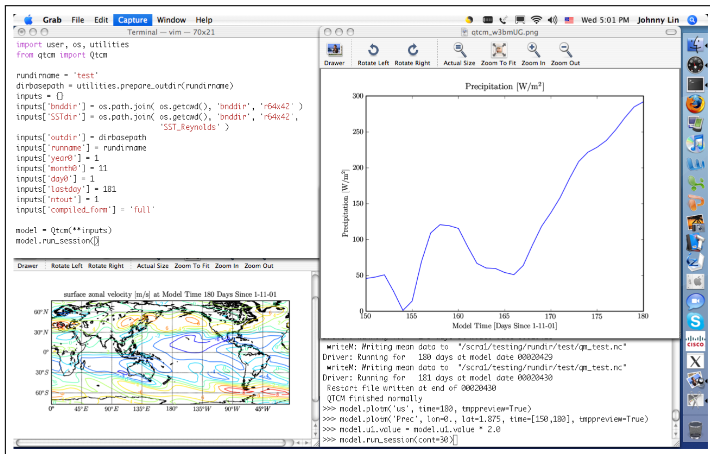
Wall-clock times (sec) for the average of three 365 day aquaplanet runs using climatological sea surface temperature as the lower boundary forcing (Lin 2008). All runs are executed as single threads.

Examples of qtcm uses: Conditionally explore parameter space

Explore different values of mixed-layer depth (`ziml`) over a set of 30-day runs, as a function of maximum zonal wind associated with the first baroclinic mode (`u1`) magnitude, until you find a case where the maximum of `u1` is greater than 10 m/s.

```
import os
import numpy as N
maxu1 = 0.0
while maxu1 < 10.0:
    iziml = 0.1 * maxu1
    iname = ziml- + str(iziml) + m
    ipath = os.path.join(proc, iname)
    os.makedirs(ipath)
    model = Qtcm(**inputs)
    try:
        model.sync_set_py_values_to_snapshot(snapshot=mysnapshot)
        model.init_with_instance_state = True
    except:
        model.init_with_instance_state = False
    model.ziml.value = iziml
    model.runname.value = iname
    model.outdir.value = ipath
    model.run_session()
    maxu1 = N.max(N.abs(model.u1.value))
    mysnapshot = model.snapshot
    del model
```

Examples of qtcm uses: Interactive modeling



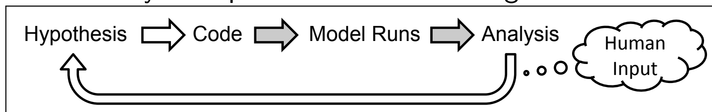
The graphs are created interactively by the user.

The effects of climate model programming structure on the modeling and analysis cycle

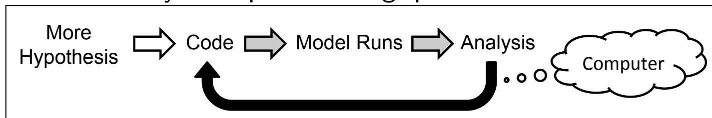
- Modeling has traditionally been a static exercise (i.e., set parameters, run, analyze output).
- The flexibility of changing i/o, data, variables, subroutine execution order, and the routines themselves at run time means modeling no longer needs to be static.
- Modeling is now more dynamic: The modeling study can adapt and change as the model runs.

Transforming the modeling and analysis cycle for climate modeling studies

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.

Automating model output analysis makes more science possible

- Model output analysis can now automatically control future model runs. Try doing that with a kludge of shell scripts, pre-processors, Matlab scripts, etc.!
- Certain science questions that used to be difficult to access are now more possible to access:
  - For certain questions, code more closely matches user thought processes.
  - Automation enables more comprehensive searching of the solution space.
  - Each increase in code complexity can be more productive with a lower per line error rate.

Conclusions

- Dynamic data structures and first-class citizenship enable dynamic namespace subroutine execution management.
- A model utilizing these features in Python, especially in an OOP framework, can make modeling easier and more reliable and also enable researchers to investigate previously inaccessible (or difficult to access) questions.

For more information

- *Geosci. Model Dev.* paper on qtcm (many portions of this presentation copied/adapted from this paper):
  `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`
- More on OOP and the atmospheric sciences (chapter 8):
  `http://www.johnny-lin.com/pyintro`
- Interested in growing the atmospheric-oceanic sciences Python community? Come join PyAOS:
  `http://pyaos.johnny-lin.com`