Using Python in Climate and Meteorology

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

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

Lives of Significance and Service
NORTH PARK UNIVERSITY
CHICAGO

Outline

- Why Python?
- Python tools for data analysis: CDAT.
- Using Python to make models more modular: The qtcm example.

Why Python?: Topics

- What is Python?
- Advantages and disadvantages.
- 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, as well as a fully native object-oriented (O-O) language.
- 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

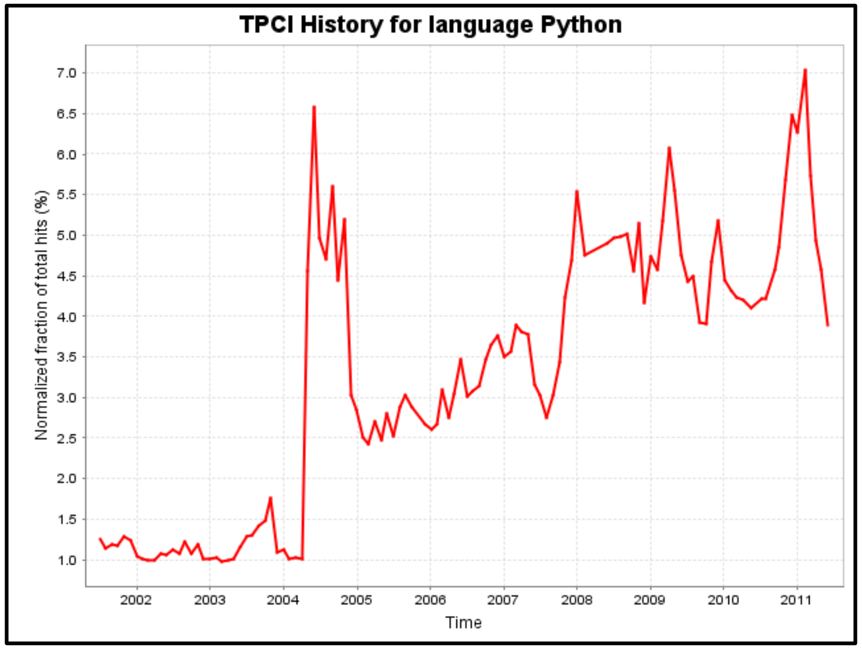
- Concise but natural syntax, both arrays and non-arrays, makes programs clearer. “Python is executable pseudocode. Perl is executable line noise.”
- Interpreted language makes development easier.
- Object-orientation makes code more robust/less brittle.
- Built-in set of data structures are very powerful and useful (e.g., dictionaries).
- Namespace management which prevents variable and function collisions.
- Tight interconnects with compiled languages (Fortran via f2py and C via SWIG), so you can interact with compiled code when speed is vital.
- Large user and developer base in industry as well as science (e.g., Google, LLNL PCMDI).

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 as a language was developed in the late 1980s, but even as late as 2004, could be considered relatively "esoteric."
- But by 2005, Python made a quantum jump in popularity, and has continued to grow.
- Now is ranked 8th in the June 2011 TIOBE Programming Community Index (which indicates the "popularity" of languages). Fortran is ranked 33rd, Matlab 23rd, and IDL is in the undifferentiated 50-100th group.



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

Why AOS Python is now gaining momentum (cont.)

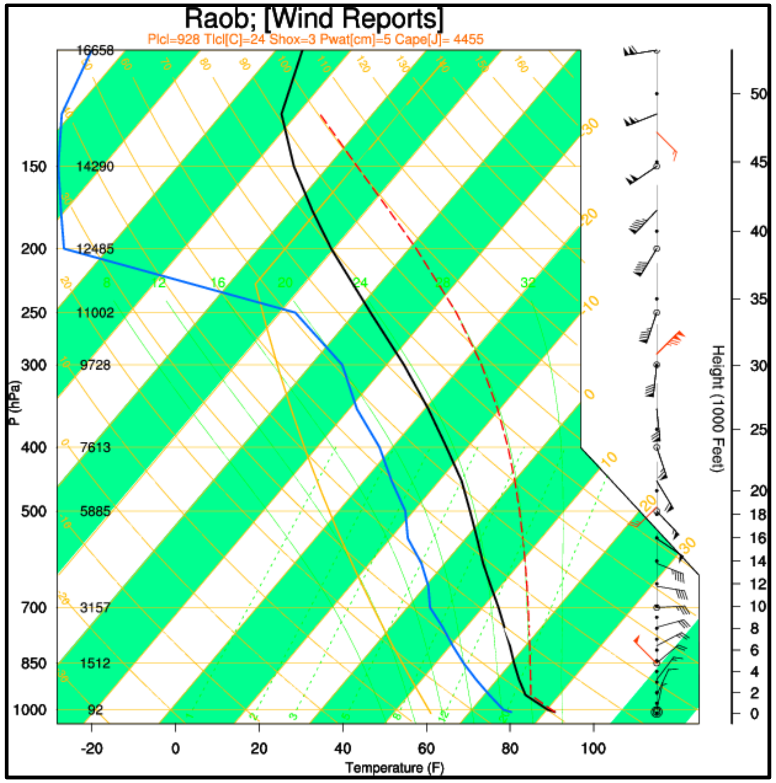
- Around the same time, key tools for AOS use became available:
  - In 2005, NumPy was developed which (finally) provided a standard array package.
  - SciPy was begun in 2001 and incorporated NumPy to create a one-stop shop for scientific computing.
  - CDAT 3.3 was released in 2002, which proved to be the version that really took off (the current version is 5.2).
  - PyNGL and PyNIO were developed in 2004 and 2005, respectively.
  - matplotlib added a contouring package in the mid-2000's.

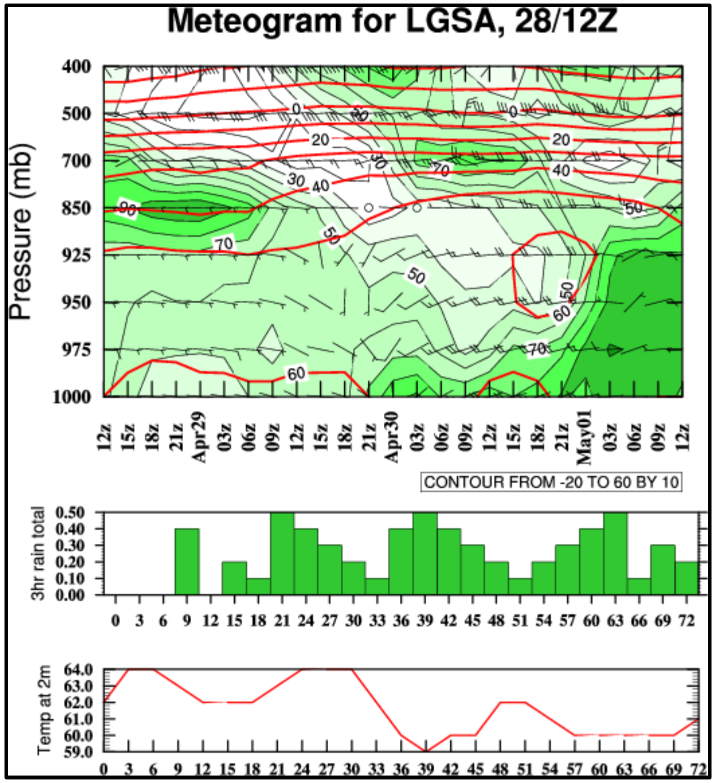
http://www.scipy.org/History_of_SciPy
http://www.pyngl.ucar.edu/Images/history.png

Why AOS Python is now gaining momentum (cont.)

- Institutional support in the AOS community now includes:
  - Lawrence Livermore National Laboratory's Program for Climate Model Diagnosis and Intercomparison (LLNL PCMDI) produces CDAT, etc.
  - National Center for Atmospheric Research Computational and Information Systems Laboratory (NCAR CISL) produces PyNGL and PyNIO.
  - Center for Ocean-Land-Atmosphere Studies/Institute of Global Environment and Society (COLA/IGES) produces PyGrADS.
  - American Meteorological Society (AMS): Sponsored an Advances in Using Python symposium at the 2011 AMS Annual Meeting (the symposia will reprise in 2012) as well as a Python short course.
- 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.

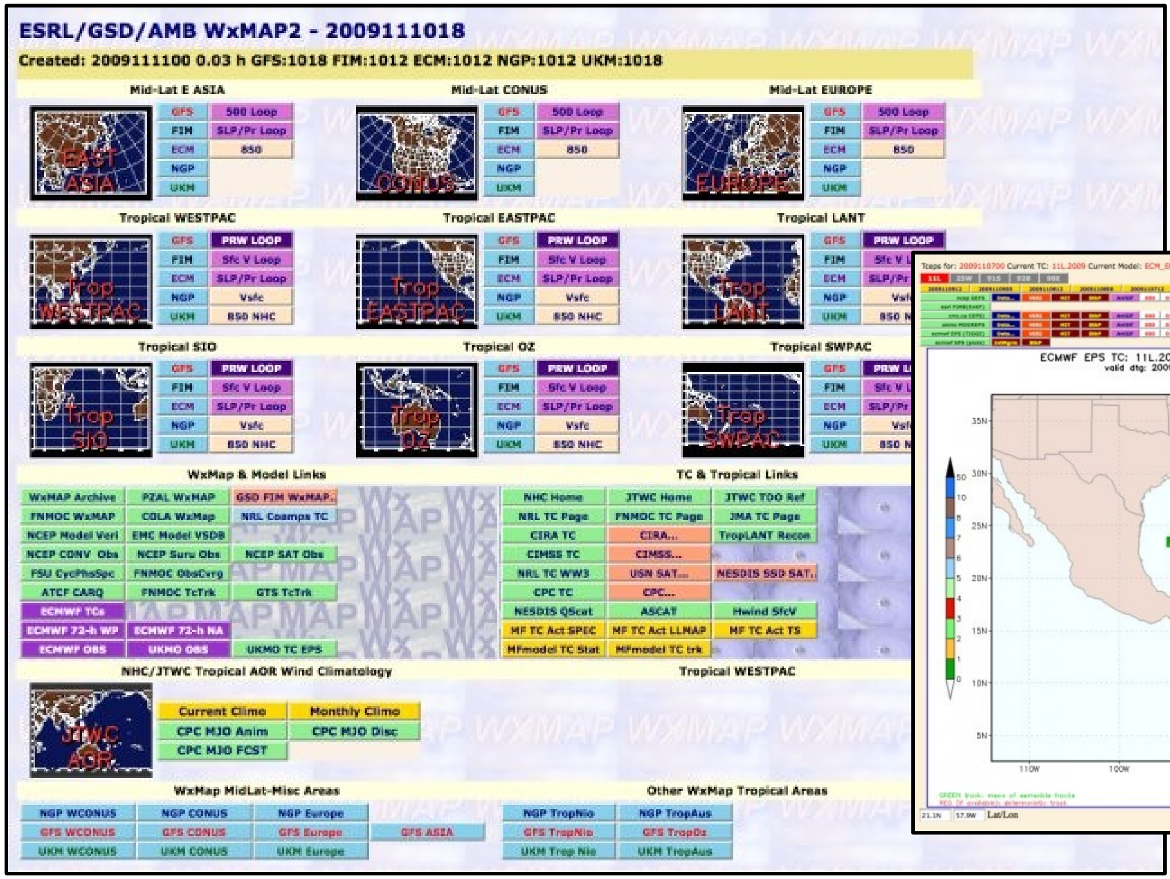
Example of visualization: Skew-T and meteograms



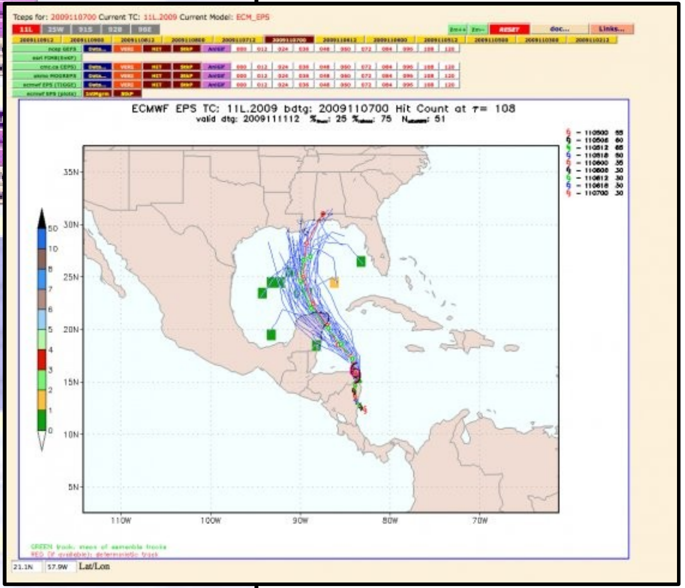


- All plots on this slide are produced by PyNGL and taken from their web site.
- See http://www.pyngl.ucar.edu/Examples/gallery.shtml for code.

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

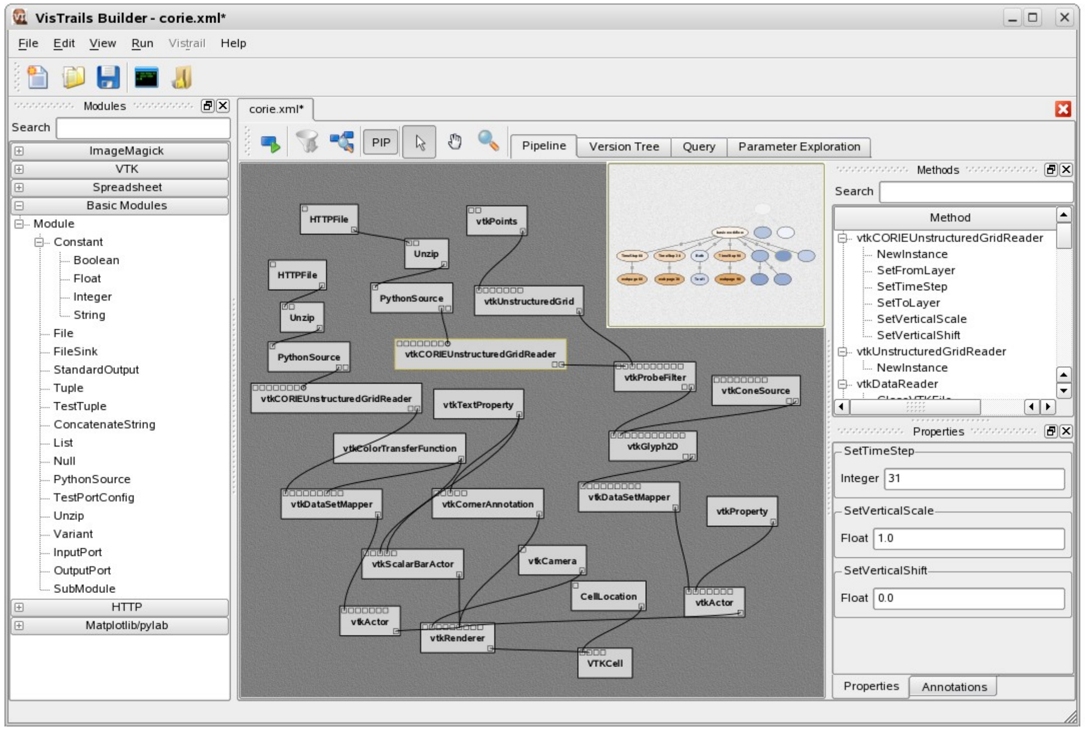

ESRL/GSD/AMB WxMAP2 - 2009111018

Created: 2009111100 0.03 h GFS:1018 FIM:1012 ECM:1012 NGP:1012 UKM:1018

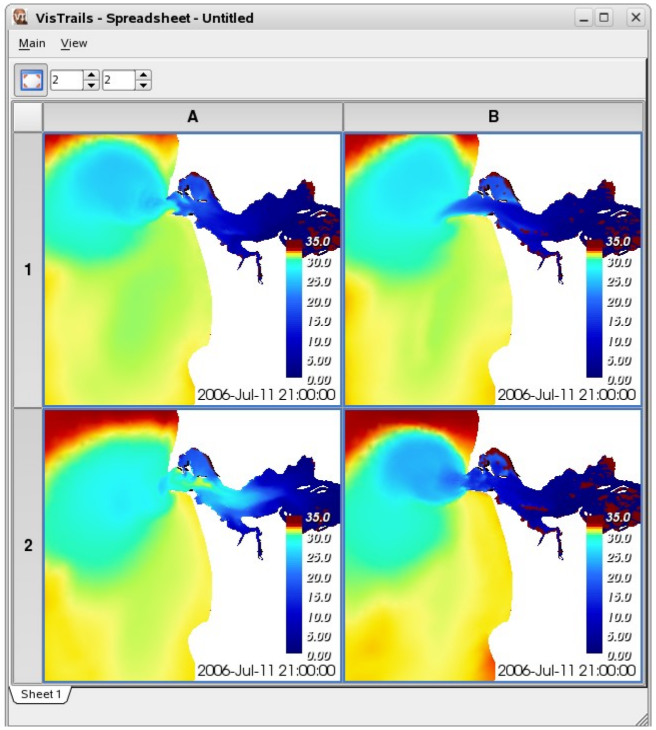

Tceps for: 2009110700 Current TC: 11L.2009 Current Model: ECM_EPS
RESET
ECMWF EPS TC: 11L.2009 bdtg: 2009110700 Hit Count at τ= 108
35N
30N
25N
20N
15N
10N
5N
110W
100W
90W
80W
70W
Lat/Lon

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

Example of analysis and visualization: Provenance management



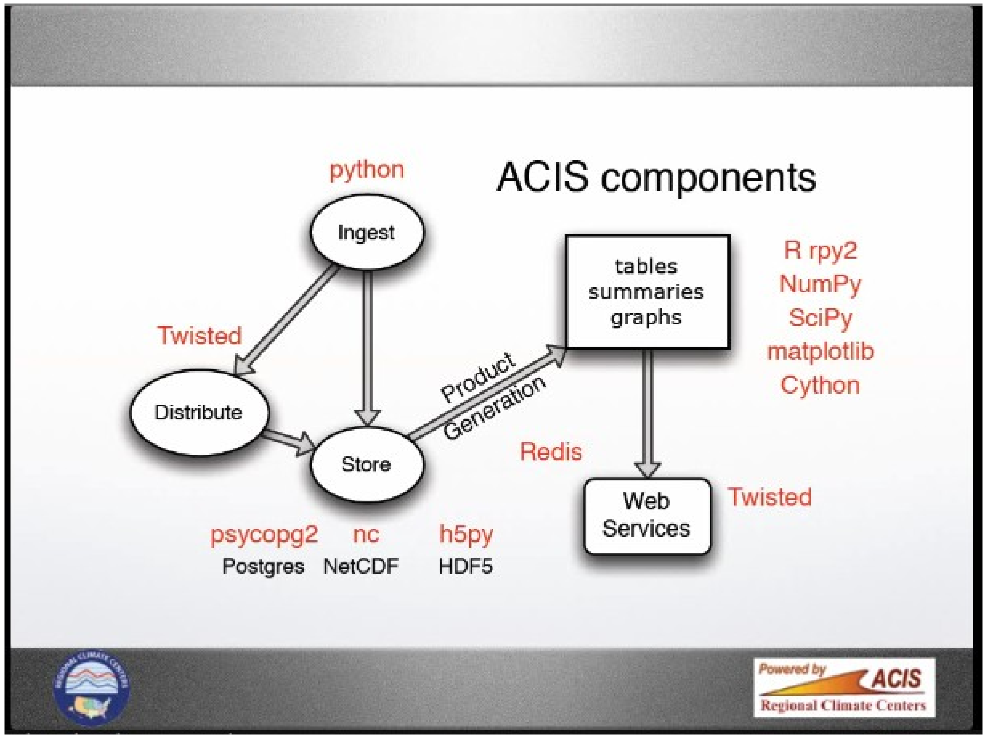
SetTimeStep
Integer 31
SetVerticalScale
Float 1.0
SetVerticalShift
Float 0.0



Session of the VisTrails visualization and data workflow and provenance management system; salinity data in the Columbia River estuary is graphed.

http://www.vistrails.org/index.php?title=File:Corie_example.png&oldid=616

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

Outline

- Why Python?
- Python tools for data analysis: CDAT.
- Using Python to make models more modular: The qtcm example.

Python tools for data analysis: Topics

We will look at only one tool (the Climate Data Analysis Tools or CDAT) and a few aspects of how that tool helps us with data analysis:

- What is CDAT?
- Dealing with missing data: Masked arrays and variables.
- Comparing different datasets: Time axis alignment.
- Dealing with different grids: Vertical interpolation, regridding.
- What is VCDAT?
- A walk through simple analysis using VCDAT.

What is CDAT?

- Written by LLNL PCMDI and designed for climate science data, CDAT was first released in 1997.
- 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 (cdms).
  - NumPy, masked array (ma), masked variable (MV2)
  - Visualization (vcs, Xmgrace, matplotlib, VTK, Visus, etc.
  - And more! (i.e., OPeNDAP, ESG, etc.).
- Graphical user interface (VCDAT).
- XML representation (CDML/NcML) for data sets.
- Community software (BSD open source license).
- URL: http://www-pcmdi.llnl.gov/software-portal.

Dealing with missing data: Why masked arrays and masked variables?

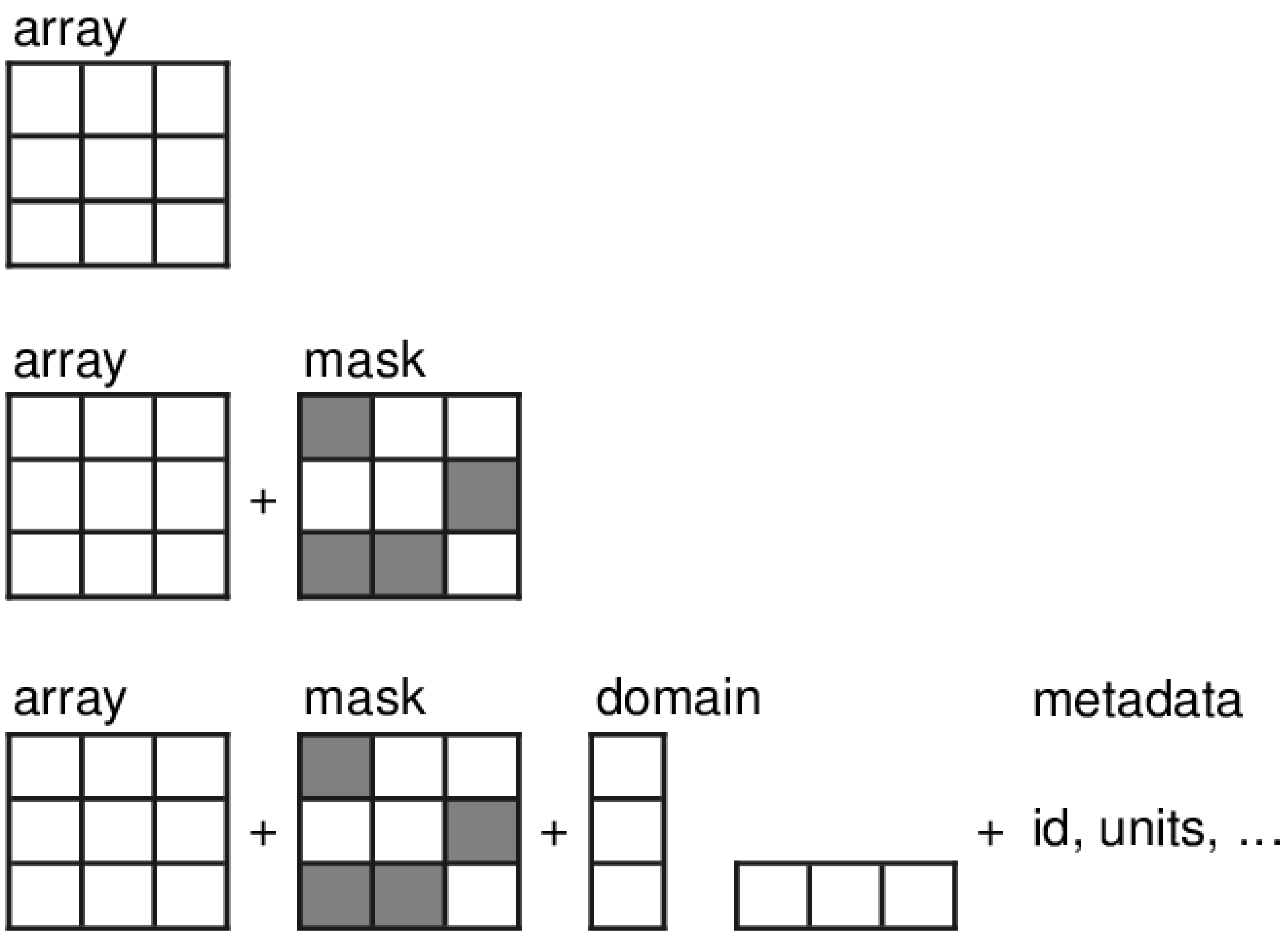
- Python supports array variables (via NumPy).
- All variables in Python are not technically variables, but objects:
  - Objects hold multiple pieces of data as well as functions that operate on that data.
  - For AOS applications, this means data and metadata (e.g., grid type, missing values, etc.) 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.

What are masked arrays and masked variables?

Arrays:
(numpy)

Masked Arrays:
(numpy.ma)

Masked Variables:
(MV2)



How do masked arrays and masked variables look and act in Python?

```
>>> import numpy as N
>>> a = N.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

Arrays: Every element has a value, and operations using the array are defined accordingly.

```
>>> import numpy.ma as ma
>>> b = ma.masked_greater(a, 4)
>>> b
masked_array(data =
 [[1 2 3]
 [4 -- --]],
             mask =
 [[False False False]
 [False  True  True]],
      fill_value = 999999)
>>> print a*b
[[1 4 9]
 [16 -- --]]
```

Masked arrays: A mask of bad values travels with the array. Those elements deemed bad are treated as if they did not exist. Operations using the array automatically utilize the mask of bad values.

How do masked arrays and masked variables look in Python (cont.)?

Additional info such as:

Metadata

Axes

```
>>> import MV2
>>> d = MV2.masked_greater(c,4)
>>> d.info()
*** Description of Slab variable_3 ***
id: variable_3
shape: (3, 2)
filename:
missing_value: 1e+20
comments:
grid_name: N/A
grid_type: N/A
time_statistic:
long_name:
units:
No grid present.
** Dimension 1 **
   id: axis_0
   Length: 3
   First:  0.0
   Last:   2.0
   Python id:  0x2729450
[... rest of output deleted for space ...]
```

Comparing different datasets: Time axis alignment

- Different datasets and model runs can have ways of specifying time, e.g., different:
  - Calendars: Gregorian, Julian, 360 days/year, etc.
  - Starting points: Since 1 B.C./A.D. 1, since Jan 1, 1970, etc.
- CDAT time axis values can be referenced in two ways:
  - Relative time: A number relative to a datum (e.g., 10 months since Jan 1, 1970).
  - Component time: A date on a specific calendar (e.g., 1970-10-01).
- If `t` is your time axis (via getTime):
  - To change to the same relative time datum, e.g.:
    ```
    t.toRelativeTime('months since 1800-01-01')
    ```
  - To change to component time based on a specified calendar, e.g.:
    ```
    t.asComponentTime(calendar=cdtime.DefaultCalendar)
    ```
- From there, you can subset either on the basis of the same relative time values or for a given component time date.

Dealing with different grids

- Vertical interpolation:

```
P = cdutil.vertical.reconstructPressureFromHybrid(Ps,
A, B, P0)
```

```
P = cdutil.vertical.linearInterpolation(var, depth,
levels)
```

```
P = cdutil.vertical.logLinearInterpolation(var,
depth, levels)
```

- Vertical regridding: Use the MV2 pressureRegrid method:

```
var_on_new_levels = var.pressureRegrid(levout)
```

- Horizontal regridding: Create a grid, then use the MV2 regridding method to create a variable on the new grid. E.g., from a rectangular grid to a 96x192 rectangular Gaussian grid:

```
var = f('temp')
n48_grid = cdms2.createGaussianGrid(96)
var48 = var.regrid(n48_grid)
```

What is VCDAT?

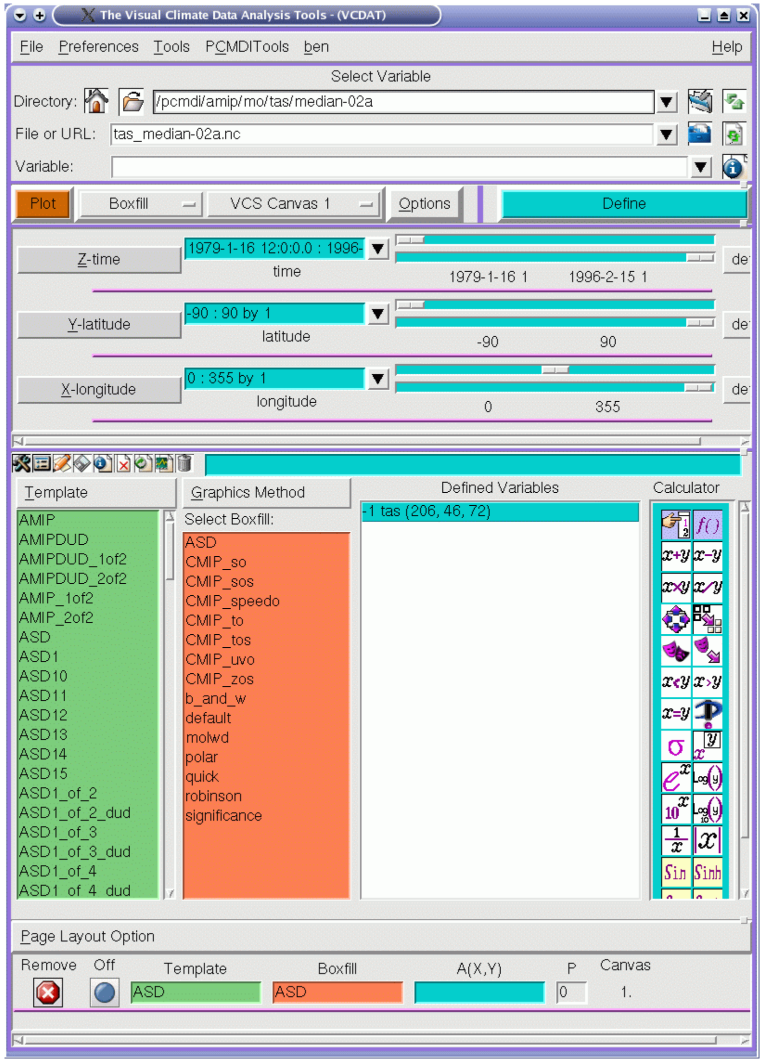
VCDAT lets you get familiar with many parts of CDAT without learning Python.

The executable is named `vcdat` and is found in `/opt/cdat/bin`.

Variable selection

Dimension selection and manipulation

Variable and graphics selection and manipulation



Directory: /pcmdi/amip/mo/tas/median-02a

File or URL: tas_median-02a.nc

Variable:

Z-time	1979-1-16 12:0:0.0 : 1996-	time	1979-1-16 1	1996-2-15 1
Y-latitude	-90 : 90 by 1	latitude	-90	90
X-longitude	0 : 355 by 1	longitude	0	355

Template: AMIP, AMIPDUD, AMIPDUD_1of2, AMIPDUD_2of2, AMIP_1of2, AMIP_2of2, ASD, ASD1, ASD10, ASD11, ASD12, ASD13, ASD14, ASD15, ASD1_of_2, ASD1_of_2_dud, ASD1_of_3, ASD1_of_3_dud, ASD1_of_4, ASD1_of_4_dud

Select Boxfill: ASD, CMIP_so, CMIP_sos, CMIP_speedo, CMIP_to, CMIP_tos, CMIP_uvo, CMIP_zos, b_and_w, default, molwd, polar, quick, robinson, significance

Defined Variables: -1 tas (206, 46, 72)

Template	Boxfill	A(X,Y)	P	Canvas
ASD	ASD		0	1.

Dean Williams and Charles Doutriaux (LLNL PCMDI)

A walk through simple analysis using VCDAT

- Reading in data (local and remote)
- Selecting a variable
- Selecting axis ranges
- Plotting a longitude-latitude slice
- Plotting a time-longitude slice
- Plotting time vs. a regional average
- Basic calculations using data (defined variables)
- Saving a plot
- As a tool for teaching CDAT and Python

Outline

- Why Python?
- Python tools for data analysis: CDAT.
- Using Python to make models more modular: The qtcm example.

Traditional AOS model coding techniques and tools

- Written in compiled languages (often Fortran):
  - Pros: Fast, "standard," free, many well-tested libraries.
  - Cons: Natively procedural, limited data structures (related data are not related to each other), non-interactive, portability issues (F90 standard does not specify memory allocation for non-standard data structures and each compiler implements different features differently).
- Language choice affects development cycle.
- Interfacing with operating system often unwieldy, through shell scripts, makefiles, etc.
- Substantial amounts of very old legacy code.
- Parallelization requires the climate scientist to deal with processor and memory management (MPI) issues.

Traditional practices yield problematic and unmodular code

- Brittle: Errors and collisions are common.
- Difficult for other users (even yourself a few months/years later!) to understand.
- Difficult to extend: Interfaces are poorly defined and structured between:
  - Submodels (e.g., between atmosphere and ocean)
  - Subroutines/procedures in a model
  - Models and the operating system
  - Models and the user (in terms of the user's thinking processes)
- Non-portable: Difficult for models and sub-models to talk to one another, sometimes even on the same platform.

Capabilities of Python to help modularize models

- Object-orientation: Makes it easier to develop robust and multi-application interfaces between tools, and, in theory, more closely approximate human mental representations of the world.
- Interpreted and dynamically-typed: Enables run time alteration of variables, as well as the reuse of code in multiple contexts (e.g., `c=a*b` works properly in a routine without change regardless of what type, size, and rank `a` and `b` are).
- Shared state: You can access virtually all variables. Note this is not some sort of giant common block, because namespace management and object decomposition automatically protect from collisions and accidental overwriting.
- Built-in set/collection data structures (i.e., lists, dictionaries) **greatly** improve the flexibility of the code.
- Clarity of syntax: Code is more robust and modules developed for different applications can be more easily reused.

Example of modularizing a model with Python: The qtcm atmospheric model

- Originally there was QTCM1, the Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model (Neelin & Zeng 1999 and Zeng et al. 1999).
- Intermediate-level atmospheric model
- Written in Fortran.
- Vertical temperature and moisture profiles based upon convective quasi-equilibrium assumption.
- Resolution 5.625 deg longitude, 3.75 deg latitude.
- Includes a simple radiative code and a Betts-Miller (1986) type convective scheme.
- Reasonable simulation of tropical climatology and also includes Madden-Julian oscillation (MJO)-like variability.

The qtcm atmospheric model (cont.)

- qtcm is a Python wrapping of QTCM1:
  - 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.
- http://www.geosci-model-dev.net/2/1/2009/gmd-2-1-2009.html

qtcm features: 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 in this run yield:

- The 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` keyword chooses the model version that gives control down to the atmospheric timestep.

qtcm features: Run sessions and a continuation run in qtcm

```
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 10
inputs['mrestart'] = 0
inputs['compiled_form'] = 'parts'

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)
```

- One run session is conducted with the model instance.
- The value of `u1`, the baroclinic zonal wind, is doubled.
- A continuation run is made for 30 more days.
- All this can be controlled at runtime, and interactively.

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

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

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

- Snapshots are variables (dictionary objects) 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).
- Again, objects enable us to control model configuration and execution in a clear but powerful way.

qtcm features: Runlists 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']
```

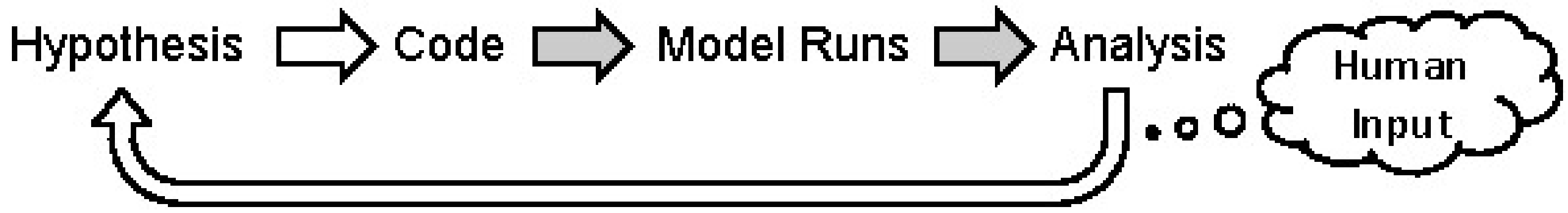
- Runlists specify a series of Python or Fortran methods, functions, subroutines (or other run lists) that will be executed when the list is passed into a call of the run_list method.
- Routines in run lists are identified by strings (instead of, for instance, as a memory pointer to a library archive object file) and so **what routines the model executes are fully changeable at run time.**
- The 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 benefits: Improving the modeling and analysis cycle for climate modeling studies

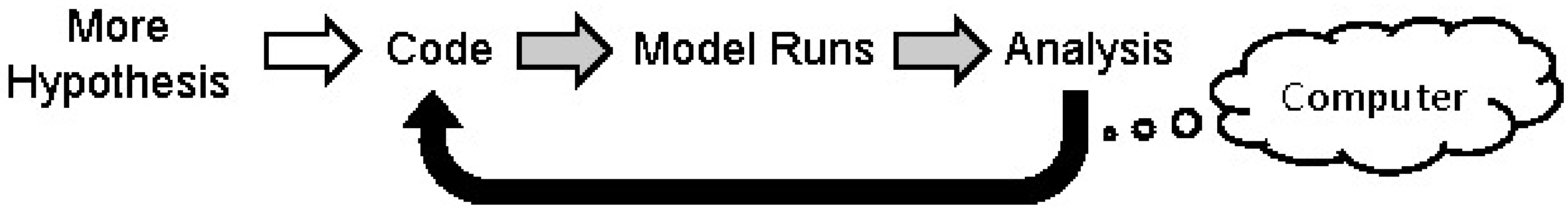
- The object decomposition provides a high level of flexibility for changing i/o, data, variables, subroutine execution order, and the routines themselves at run time.
- The performance penalty of this hybrid-languge model vs. the Fortran-only version of the model is 4–9%.
- But for this cost, modeling is now no longer a static exercise (i.e., set parameters, run, analyze output).
- With modeling more dynamic, the modeling study can adapt and change as the model runs.
- This means that we can unify not only the traditionally computer-controlled portions of a modeling workflow, but also parts of the traditionally human-controlled portions (hypothesis generation).

qtcm benefits: Improving the modeling and analysis cycle for climate modeling studies (cont.)

Traditional analysis sequence used in modeling studies:



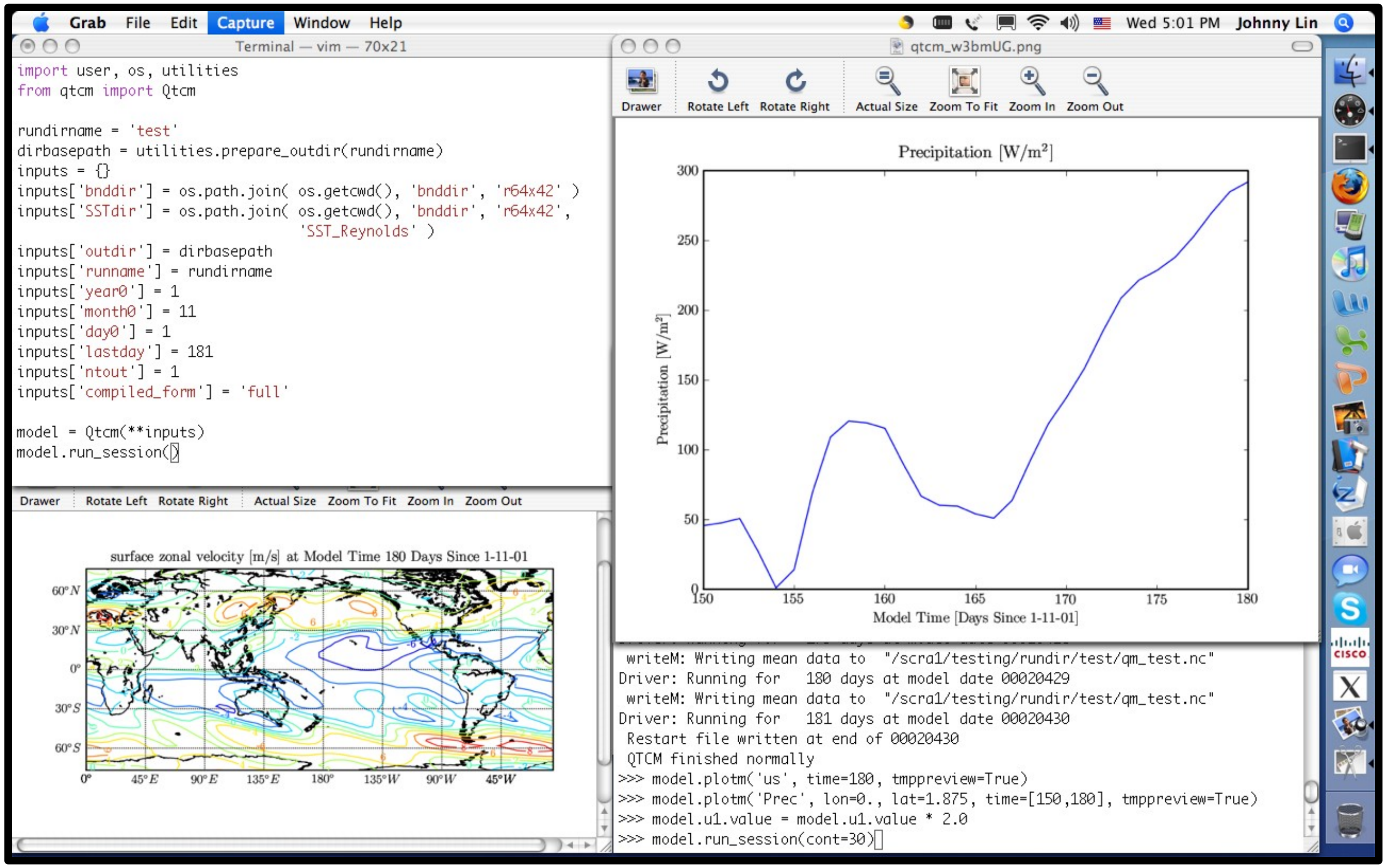
Transformed analysis sequence using qtcm-like tools:



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

Model output analysis can now automatically control future model runs. Try doing that with a kludge of shell scripts, pre-processors, Matlab scripts, etc.!

qtcm benefits: Modeling can be interactive



```
import user, os, utilities
from qtcm import Qtcm

rundirname = 'test'
dirbasepath = utilities.prepare_outdir(rundirname)
inputs = {}
inputs['bnddir'] = os.path.join( os.getcwd(), 'bnddir', 'r64x42' )
inputs['SSTdir'] = os.path.join( os.getcwd(), 'bnddir', 'r64x42',
                                 'SST_Reynolds' )
inputs['outdir'] = dirbasepath
inputs['runname'] = rundirname
inputs['year0'] = 1
inputs['month0'] = 11
inputs['day0'] = 1
inputs['lastday'] = 181
inputs['ntout'] = 1
inputs['compiled_form'] = 'full'

model = Qtcm(**inputs)
model.run_session()
```

```
 writeM: Writing mean data to  "/scra1/testing/rundir/test/qm_test.nc"
Driver: Running for   180 days at model date 00020429
 writeM: Writing mean data to  "/scra1/testing/rundir/test/qm_test.nc"
Driver: Running for   181 days at model date 00020430
 Restart file written at end of 00020430
 QTCM finished normally
>>> model.plotm('us', time=180, tmppreview=True)
>>> model.plotm('Prec', lon=0., lat=1.875, time=[150,180], tmppreview=True)
>>> model.u1.value = model.u1.value * 2.0
>>> model.run_session(cont=30)
```

- Because Python is interpreted, this permits model alteration at run time and thus interactive modeling. **All variables can be changed at run time** and in the model run.
- Visualization can also be done interactively.

Conclusions

- Python is a mature, comprehensive computational environment for all aspects of the atmospheric and oceanic sciences.
- AOS Python tools make data analysis much easier to do.
- Python offers tools to make models more flexible and capable of exploring previously difficult to access scientific problems.
- And it's (mostly) all free!