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 8

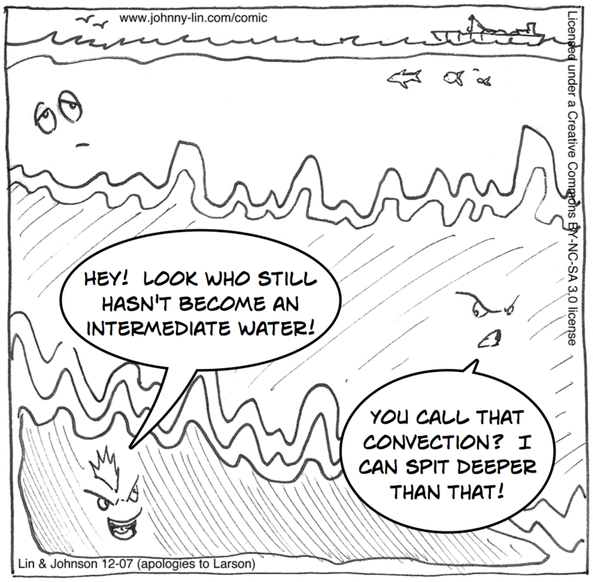
An Introduction to OOP Using Python: Part II—Application to Atmospheric Sciences Problems

Ch. 7, introduced us to the syntax of object-oriented programming (OOP), as well as an understanding of how we can use OOP to write AOS programs that are more flexible, extensible, and less error prone. In this chapter, we look at a few applications of OOP to AOS problems. In particular, we will examine how objects can be used to manage dataset metadata (including missing values), related but unknown data, and dynamically change subroutine execution order. Of these three topics, the first is addressed by two well-developed packages, NumPy and the Climate Data Management System (cdms). The second and third topics are implemented in two experimental packages, but they provide useful illustrations of how we can apply OOP to AOS problems.

8.1 Managing metadata and missing values

All datasets of real-world phenomena will have missing data: instruments will malfunction, people will make measurement errors, etc. Traditionally, missing data has been handled by assigning a value as the "missing value" and setting all elements of the dataset that are "bad" to that value. (Usually, the missing value is a value entirely out of the range of the expected values, e.g., −99999.0.) With OOP, objects enable us to do this in a more robust way.

Earlier, we saw that Python supports array variables (via NumPy), and we also described how 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, and for atmospheric and oceanic sciences (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. These array-like variables incorporate metadata attached to the arrays and define how that metadata can be used as part of analysis, visualization, etc.


www.johnny-lin.com/comic
Licensed under a Creative Commons BY-NC-SA 3.0 license
HEY! LOOK WHO STILL HASN'T BECOME AN INTERMEDIATE WATER!
YOU CALL THAT CONVECTION? I CAN SPIT DEEPER THAN THAT!
Lin & Johnson 12-07 (apologies to Larson)
Oceanographic Bullies

8.1.1 What are masked arrays and masked variables?

Recall that arrays are n-dimensional vectors or grids that hold numbers (or characters). Masked arrays, then, are arrays that also have a "mask" attribute which tells you which elements are bad, and masked variables are masked arrays that also give domain information and other metadata information. Let's look at each type of variable in detail.

Review of arrays.

In an array, every element has a value, and operations using the array are defined accordingly. Thus, for the following array:

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

the contents of the array are numbers, and operations such as addition, multiplication, etc. are defined as operating on those array elements, as we saw in Ch. 4.

Masked arrays.

Masked arrays are arrays with something extra. That something extra is a mask of bad values; this mask is an attribute of the array and thus automatically travels with the numerical (or character) values of the elements of the array. Elements of the array, whose corresponding elements in the mask are set to "bad," are treated as if they did not exist, and operations using the array automatically utilize the mask of bad values. Consider the array a and the masked array b:

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

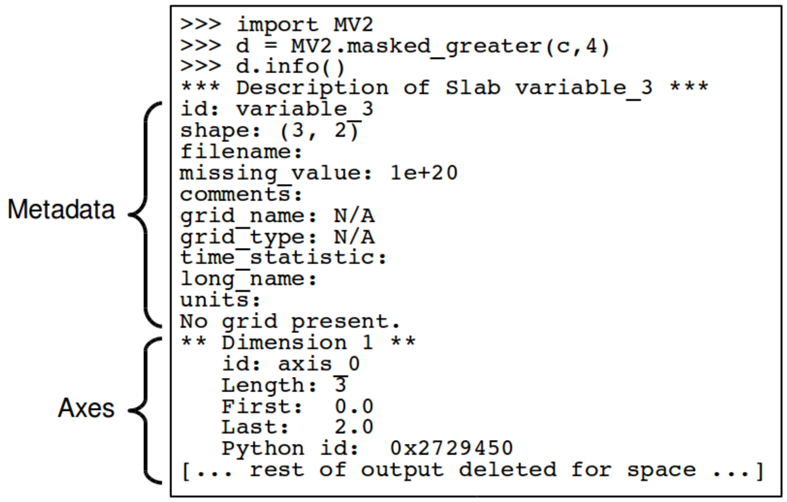
Masked array masks.

The mask is a boolean array whose elements are set to True if the value in the corresponding array is considered "bad." Thus, in the masked array b, the last two elements of the second row have mask values set to True, and when the data for the masked array is printed out for a human viewer, those elements display "--" instead of a number.

Masked array fill values.

We also note that the masked array b has an attribute called fill_value that is set to 999999. As we'll see in Example 52, this is the value used to fill-in all the "bad" elements when we "deconstruct" the masked array. That is to say, when we convert a masked array to a normal NumPy array, we need to put something in for all the "bad" elements (i.e., where the mask is True): the value of fill_value is what we put in for the "bad" elements.

Just as operators have been defined to operate in a special way when the operands are arrays (i.e., the + operator adds element-wise for arrays), operators have also been defined to operate in a special way for masked arrays.



```
>>> 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 ...]
```

Figure 8.1: Example of information attached to a masked variable. Adapted from a figure by Bob Drach, Dean Williams, and Charles Doutriaux. Used by permission.

Operations using masked arrays.

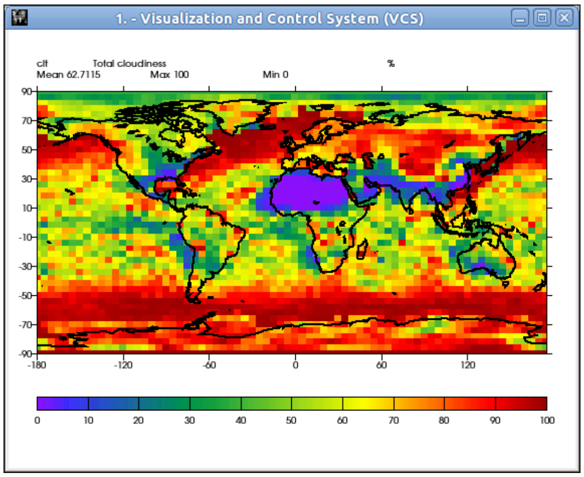
For masked arrays, operations using elements whose mask value is set to `True` will create results that also have a mask value set to `True`. Thus, in the example above, the product of array `a` and masked array `b` yields an array whose last two elements of the second row are also "bad," since those corresponding elements in masked array `b` are bad: a good value times a bad value gives a bad value. Thus, masked arrays transparently deal with missing data in real-world datasets.

Masked variables.

A masked variable is like a masked array but with additional information, such as axes and domain information, metadata, etc. Figure 8.1 shows an example of the additional information that can be attached to a masked variable.

The domain information and other metadata attached to a masked variable can be used in analysis and visualization routines. UV-CDAT functions, for instance, are pre-built to do just this. As an example, consider Figure 8.2 which shows the use of UV-CDAT's cdms2 module to read in the total cloudiness (clt) variable from a netCDF file and UV-CDAT's vcs module to render the plot using a single command. This is possible because the vcs `boxfill` method uses the information attached to the masked variable to properly title the graph, label the units, etc.

As a summary, Figure 8.3 gives a schematic that shows how each of these three types of "arrays" relate to each other. Arrays and masked arrays are both part of NumPy whereas masked variables are part of UV-CDAT.


1. - Visualization and Control System (VCS)
clt Total cloudiness %
Mean 62.7115 Max 100 Min 0

is made by:

```
>>> v =
vcs.init()
>>>
v.boxfill(clt)
```

Figure 8.2: Example showing plot of the total cloudiness (clt) variable read from a netCDF file and the code used to generate the plot, using UV-CDAT masked variables.

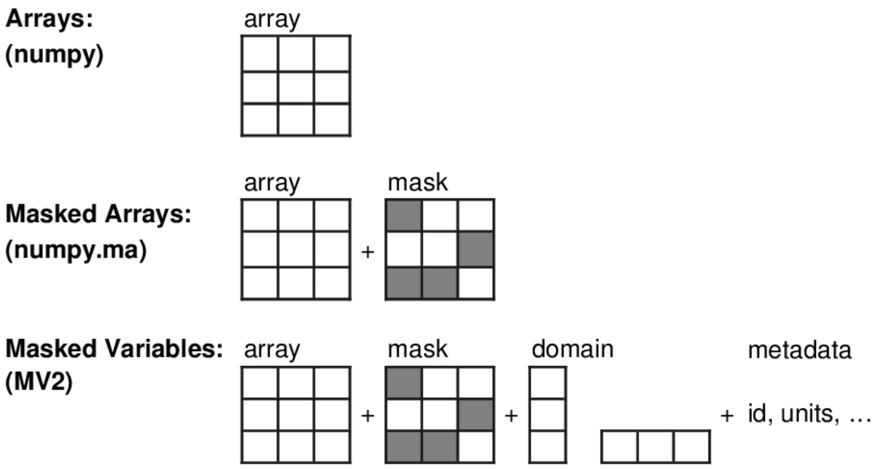


Figure 8.3: Schematic of arrays, masked arrays, and masked variables. Adapted from a drawing by Bob Drach, Dean Williams, and Charles Doutriaux. Used by permission.

(See Section 10.2 for more information on UV-CDAT.)

8.1.2 Constructing and deconstructing masked arrays

We covered construction of normal NumPy arrays in Ch. 4, so we won't revisit that topic here. Construction of masked variables is normally not something you would do in a program; if you already have the metadata available to you in the program, it may not make much sense to attach it to a masked variable instead of using it directly later on in the program. Many times, for most common uses, masked variables will be automatically constructed for you upon read from a self-describing input file format like netCDF. As a result, in this section, I will only cover the construction and deconstruction of masked arrays. For details on the construction and deconstruction of masked variables, please see the CDAT documentation.[1]

Masked array submodule and constructors.

NumPy provides a number of masked array constructors. I'll illustrate the use of a few of these constructors through examples. Details of these functions, as well as information on other masked array constructors, are found in the NumPy documentation.[2] In the examples below, all functions are part of the `numpy.ma` submodule and I assume I've already imported that submodule with `import numpy.ma as MA` and that NumPy is already imported as `import numpy as N`. Before you type in the example, try to guess what the output will be, based on the command syntax itself. Note that you can see what `a` looks like by typing in the array name `a` by itself, which will reveal the data, mask, and fill value.

Example 50 (Make a masked array by explicitly specifying a mask and fill value):

Type in the following in the Python interpreter:

```
a = MA.masked_array(data=[1,2,3],
        mask=[True, True, False], fill_value=1e20)
```

What does the variable `a` look like?

Solution and discussion: As expected, the first two array elements are now considered "bad." Here is the result of an array inquiry done in the Python interpreter:

[1] http://www2-pcmdi.llnl.gov/cdat/manuals (accessed August 17, 2012).

[2] http://docs.scipy.org/doc/numpy/reference/maskedarray.html (accessed August 17, 2012).

```
>>> a
masked_array(data = [-- -- 3],
              mask = [ True  True False],
        fill_value = 999999)
```

Often times, we determine which data values should be masked on the basis of whether or not the data values are beyond a given acceptable value or some other similar test. Thus, it makes sense to have functions to create masked arrays based on such tests. Here are a few examples of such functions:

Example 51 (Make a masked array by masking values based on conditions):

Type in the following in the Python interpreter:

Masked array construction using conditions.

```
a = MA.masked_greater([1,2,3,4], 3)
data = N.array([1,2,3,4,5])
b = MA.masked_where(N.logical_and(data>2, data<5), data)
```

What do the variables a and b look like?

Solution and discussion: The first line creates a masked array a where all values greater than 3 have mask values set to True. The third line creates a masked array b where all values greater than 2 and less than 5 have mask values set to True. Note that masked_where's first argument is a boolean array that specifies where the mask will be set to True. An inquiry into masked arrays a and b should return:

```
>>> a
masked_array(data = [1 2 3 --],
             mask = [False False False  True],
       fill_value = 999999)
>>> b
masked_array(data = [1 2 -- -- 5],
             mask = [False False  True  True False],
       fill_value = 999999)
```

There are times when we want just a regular NumPy array—perhaps to output to a file format that does not support object attributes—and so we need a way to convert a masked array into a normal array. In that conversion, the function will fill all "bad" elements with a special value that no "good" element would have. The masked array submodule function `filled` does the job. Let's look at an example:

Converting from a masked array to a regular array with `filled`.

Example 52 (Make a regular NumPy array from a masked array):
Type in the following in the Python interpreter:

```
b = MA.masked_array(data=[1.0, 2.0, 3.0],
        mask=[True, True, False], fill_value=-1e23)
a = MA.filled(b)
```

What do the variables `b` and `a` look like?

Solution and discussion: In this example, we start with a masked array and convert it into a normal NumPy array `a`. Thus, the variables `b` and `a` are:

```
>>> b
masked_array(data = [-- -- 3.0],
             mask = [ True  True False],
       fill_value = -1e+23)
>>> a
array([ -1.00000000e+23,  -1.00000000e+23,
         3.00000000e+00])
```

(I manually added a line break in the screenshot to make it fit on the page.) Note that we create our masked array with a fill value different than the default of 999999. Thus, the array `a` that results will have -1×10^{23} as the "missing value" value. Also note that if the type of `data` and the type of `fill_value` conflict, the default value of `fill_value` will be used despite the explicit specification of `fill_value` in the function call (if the default `fill_value` is of the same type as `data`). Thus:

Customized fill value for a masked array.

```
>>> b = MA.masked_array(data=[1, 2, 3],
            mask=[True, True, False], fill_value=-1e23)
>>> b
masked_array(data = [-- -- 3],
             mask = [ True  True False],
       fill_value = 999999)
```

yields a masked array `b` with a `fill_value` set to the default value, which is an integer.

By the way, the `filled` function also comes in a masked array method form, so instead of calling the function `filled`, i.e:

`Filled` is also a masked array method.

```
a = MA.filled(b)
```

you can call the method attached to the masked array, i.e.:

```
a = b.filled()
```

Remember, *bad values* (i.e., the missing values) have mask values set to `True` in a masked array.

8.1.3 Exercise using masked arrays

▷ **Exercise 25 (Creating and using a masked array of surface air temperature):**

Open the example netCDF NCEP/NCAR Reanalysis 1 netCDF dataset of monthly mean surface/near-surface air temperature (or the netCDF dataset you brought) and read in the values of the `air`, `lat`, and `lon` variables into NumPy arrays. Take *only the first time slice* of the air temperature data. (The example data is in *course_files/datasets* in the file *air.mon.mean.nc*.)

Create an array that masks out temperatures in that time slice in all locations greater than 45°N and less than 45°S. Convert all those temperature values to K (the dataset temperatures are in °C). Some hints:

- You can use the code in *exercise-netcdf.py* in *course_files/code_files* as a starting point.
- Use the `meshgrid` function in NumPy to make it easier to handle the latitude values in array syntax (you can, of course, always use `for` loops).
- The air temperature, directly from the netCDF file, has the shape (755, 73, 144) and thus is dimensioned time, latitude, longitude.
- You can test whether you masked it correctly by printing the values of your masked array at the poles and equator (i.e., if your masked array is called `ma_data`, you would print `ma_data[0,:]`, `ma_data[-1,:]`, and `ma_data[36,:]`).

Solution and discussion: Here's my solution:

```
1  import numpy as N
2  import numpy.ma as MA
3  import Scientific.IO.NetCDF as S
4
5  fileobj = S.NetCDFFile('air.mon.mean.nc', mode='r')
6  data = fileobj.variables['air'].getValue()[0,:,:]
7  lat = fileobj.variables['lat'].getValue()
8  lon = fileobj.variables['lon'].getValue()
9  fileobj.close()
10
11 [lonall, latall] = N.meshgrid(lon, lat)
12 ma_data = MA.masked_where( \
13         N.logical_or(latall>45,latall<-45), data )
14 ma_data = ma_data + 273.15
15
16 print 'North pole:  ', ma_data[0,:]
17 print 'South pole:  ', ma_data[-1,:]
18 print 'Equator:  ', ma_data[36,:]
```

The result of line 16 should show that all the points in the zeroth row of `ma_data` are "bad," as should the result of line 17 for the last row. All the points in line 18, which are the Equatorial points, should be "good" values, and in units of Kelvin.

See the code in *exercise-ma.py* in *course_files/code_files* for the above solution (with minor changes).

8.2 Managing related but unknown data: Seeing if attributes are defined

In the atmospheric and oceanic sciences, we are often interested in "secondary" quantities, for instance, virtual temperature, vorticity, etc., that are derived from "primary" quantities (like temperature, pressure, etc.) and other secondary quantities. In other words, final quantities often depend on both basic and intermediate quantities. For instance, density depends on virtual temperature which depends on temperature. Thus, many of these quantities are related to each other.

In traditional procedural programming, to calculate secondary variables, we would figure out all the quantities we want to calculate (both final and intermediate), allocate variables for all those quantities, then calculate our desired variables using the proper sequence of functions. But unless we know exactly what we want to calculate, we won't know what variables to allocate and what functions to call; we often get around that problem by allocating memory for every conceivable variable of interest. But why should we have to do this? Put another way, the problem with the procedural method is that we are limited to *static* analysis. Since computers are all about automation, why can't we have the computer automatically calculate what quantities it needs when it needs it; in the atmosphere and ocean, all these quantities are interrelated. This would enable *dynamic* analysis.

Using Python for dynamic variable management.

Python, it turns out, can do dynamic variable management. At any time in the program, objects can add and remove attributes and methods and check if an attribute or method exists. Let's take advantage of these capabilities and design a class to manage the multiple atmospheric quantities we're calculating: to make sure we have calculated what we need to calculate, when we need it. We define an object class `Atmosphere` where the following occurs:

- Atmospheric quantities are assigned to attributes of instances of the class.
- Methods to calculate atmospheric quantities:
  - Check to make sure the required quantity exists as an attribute.
  - If it doesn't exist, the method is executed to calculate that quantity.
  - After the quantity is calculated, it is set as an attribute of the object instance.

What might something with these traits look like in Python code? Here's a skeleton class definition:

```
class Atmosphere(object):
    def __init__(self, **kwds):
        for ikey in kwds.keys():
            setattr(self, ikey, kwds[ikey])

    def calc_rho(self):
        if not hasattr(self, 'T_v'):
            self.calc_virt_temp()
        elif not hasattr(self, 'p'):
            self.calc_press()
        else:
            raise ValueError, \
                "cannot obtain given initial quantities"

        self.rho = \
            [... find air density from self.T_v and
                 self.p ...]
```

The `setattr`, `hasattr`, `getattr`, and `delattr` functions.

Before talking about the code in specific, let me briefly describe what the `setattr`, `hasattr`, `getattr`, and `delattr` functions do (the last two are not used in the code above, but I describe them for completeness). As the names suggest, these functions manipulate or inquire of the attributes of objects. However, because they are functions, they enable you to interact with attributes without having to actually type out the name of the attribute. For instance, consider the act of setting an attribute which we've already seen can be done with assignment . So, if we have the following masked array a (as in Example 51):

```
import numpy.ma as MA
a = MA.masked_greater([1,2,3,4], 3)
```

we can manually change the fill value from its default value 999999 to something else by assignment:

```
>>> a.fill_value=-100
>>> a
masked_array(data = [1 -- --],
             mask = [False  True  True],
       fill_value = -100)
```

or we can use the function `setattr`:

```
>>> setattr(a, 'fill_value', 456)
>>> a
masked_array(data = [1 -- --],
             mask = [False  True  True],
       fill_value = 456)
```

The setattr function takes three arguments. The first is the object whose attribute you wish to set. The second is the name of the attribute you wish to set (as a string). The third argument is the new value of the attribute you are setting. Because setattr is a function, you can pass in the arguments as variables. You do not have to type in a period and equal sign, which the assignment syntax requires you to do. Functions can receive variables as arguments and so can be automated; typing a period and equal sign can only be applied to actually defined objects and so cannot be automated. Normally, methods are tailored for a class of objects and will not be set during run-time. However, you can add methods to an instance at run-time, if you wish, by setting an attribute to a function or method object.

The hasattr function tests whether a given object has an attribute or method of a given name. It takes two arguments, the first being the object under inquiry and the second being the name of the attribute or method you're checking for, as a string. True is returned if the object has the attribute you're checking for, False otherwise. Thus, for masked array a:

```
a = MA.masked_greater([1,2,3,4], 3)
print hasattr(a, 'fill_value')
print hasattr(a, 'foobar')
```

the first print line will print True while the second will print False.

The functions getattr and delattr have the same syntax: The first argument is the object in question while the second argument is the attribute to either get or delete. getattr returns the attribute or method of interest while delattr removes the attribute of interest from the given object. (Note that delattr cannot remove a method that is hard-coded into the class definition.)

With this as background, let's see what this code does. We pass in initial values for our atmospheric variables via the __init__ method, as normal, but in this case, all our initial values come through keyword parameters as given in the kwds dictionary (see Example 16 for more on passing a keyword parameters dictionary into a function, as opposed to referring to every keyword parameter manually). In our keyword parameters dictionary, we assume that the keyword names will be the names of the attributes that store those parameter values. Once passed in we set the keyword parameters to instance attributes of the same name as the keyword. For instance, if an `Atmosphere` object is instantiated by:

```
myatmos = Atmosphere(T=data1, q=data2)
```

(where `data1` and `data2` are the data, most likely arrays), then upon instantiation, `myatmos` will have the attribute `T` and the attribute `q` which refer to the data variables `data1` and `data2`, respectively. That is to say, `myatmos.T` refers to the `data1` data while `myatmos.q` refers to the `data2` data.

How does the `__init__` code do this? In lines 3–4 in the class definition, we loop through the keys in `kwds`, which are strings, and use the built-in function `setattr` to set the values of the dictionary entries to attributes of the instance (i.e., `self`), with the names given by the corresponding keys (i.e., `ikey`). Note how *we do not have to type in the variables to set them!* The function `setattr` does this for us. Thus, in our class definition, we do not need to know ahead of time which atmospheric quantities will be initially defined; all that can be determined at **runtime,** and our code will be the same.

How do methods that calculate quantities work with the attributes that hold atmospheric data? Lines 6–17 in the class definition define the method `calc_rho` which calculates air density using an algorithm that requires virtual temperature and pressure be already defined. So, `calc_rho` first checks if those attributes exist (the built-in `hasattr` function checks to see if an attribute is defined in the instance `self`), and if not, `calc_rho` calls the methods (defined elsewhere in the class) that calculate those atmospheric quantities. Those methods, in turn, are structured just like `calc_rho` and will do the same thing (check for an atmospheric quantity attribute, and if not found, calculate that quantity). Eventually, you'll calculate what you need given what you have; if not, you'll get an error (as in the `raise` statement of lines 12–13). Once all necessary variables are calculated, lines 16–17 calculates the air density and line 15 sets the result to an instance attribute of `self` called `rho`.

So, let's step back and think about what we've just done. First, because the class `Atmosphere` stores all primary and secondary atmospheric quantities needed to arrive at a quantity of interest, and the algorithms of `Atmosphere` are (hopefully) consistent with each other, all of the atmospheric quantities in an `Atmosphere` instance will be consistent with one another. Second, by using the `hasattr` function, the class automatically ensures all necessary secondary quantities are available if needed for the current calculation. In fact, the class *will find a way* to calculate what you asked it to, if the algorithms in the class will allow you to make the calculation you want using the initial values you gave. Lastly, the class can be used with any set of initial values that are input. The ability to inquire of and manipulate the attributes and methods of an object through functions enables us to write code in which the names of the initial atmospheric quantities are not known ahead of time. Our code is more flexible (and, in this case, concise) as a result.

8.3 Exercise to add to the `Atmosphere` class

▷ **Exercise 26 (Adding the method `calc_virt_temp`):**

Write the skeleton definition for a method `calc_virt_temp` (to be added to the `Atmosphere` class) that calculates the virtual temperature given mixing ratio (`r`) and temperature (`T`). Have this method call a method to calculate mixing ratio (`calc_mix_ratio`) if mixing ratio is not yet an object attribute. (We'll assume temperature has to be given.)

Solution and discussion: Here's the `Atmosphere` class with the skeleton definition for `calc_virt_temp` added:

```
class Atmosphere(object):
    def __init__(self, **kwds):
        for ikey in kwds.keys():
            setattr(self, ikey, kwds[ikey])

    def calc_rho(self):
        if not hasattr(self, 'T_v'):
            self.calc_virt_temp()
        elif not hasattr(self, 'p'):
            self.calc_press()
        else:
            raise ValueError, \
                "cannot obtain given initial quantities"

        self.rho = \
            [... find air density using self.T_v and
                 self.p ...]

    def calc_virt_temp(self):
        if not hasattr(self, 'r'):
            self.calc_mix_ratio()
        else:
            raise ValueError, \
                "cannot obtain given initial quantities"

        self.T_v = \
            [... find virtual temperature using
                 self.r and self.T ...]
```

I once wrote a package atmqty that does what `Atmosphere` does. It was one of the earlier things I wrote and needs a major rewrite, but you might find some of the routines and the structure to be informative.[3] Also, the object-oriented approach `Atmosphere` uses was essentially the way R. Saravanan in the late 1990's (then at NCAR) structured his Hyperslab OPerator Suite (HOPS) toolkit for manipulating climate model output. Written for the Interactive Data Language (IDL) and Yorick, Saravanan's work was really ahead of its time in the atmospheric and oceanic sciences community.[4]

One final note: In this section, we discussed dynamic variable management via object attributes and methods. But this may sound familiar to you—

[3]See http://www.johnny-lin.com/py_pkgs/atmqty/doc for details (accessed August 17, 2012.)

[4]See http://www.cgd.ucar.edu/cdp/svn/hyperslab.html for details (accessed April 5, 2012.)

aren't these the same things that a dictionary can do? Through this example, we've stumbled upon a secret in Python. Not only is everything an object in Python, but (nearly) everything in Python is managed by dictionaries. All objects have a private attribute `__dict__`, a data structure that manages the attributes and methods namespace just like a dictionary because it is a dictionary! And so, if you really need to, you can access that dictionary like any other dictionary. (I do not, however, recommend this.)[5] This is a nice illustration of how compact is the definition of Python: a relatively small set of data structures and principles are repeatedly reused in many aspects of the language's definition. This makes the language easier to use, because you have fewer "special structures" to try and remember.

Nearly everything in Python is managed by dictionaries.

8.4 Dynamically changing subroutine execution order (optional)

(This section is a bit more advanced, so if you feel like it's a little too much, just skip it. The main idea is that by using lists and an object encapsulation, you can dynamically change subroutine execution order in a Python program. This opens up AOS models to easily answer whole new classes of scientific problems.)

In traditional procedural programming, the execution order of subroutines is fixed, because subroutines are called by typing in the subroutine name (along with a `call` statement, in Fortran). Even branching (via `if` statements) is fixed in that the node cannot move from the place where you typed it in.

In procedural programming, subroutine execution order is fixed.

In contrast, we saw that Python's list structure is an ordered set that is mutable and can be changed *while the program is running*. Why, then, don't we use a list to manage subroutine execution order? Then, if we want to alter execution order, we just reorder, insert, and/or delete elements from the list.

Python lists are runtime mutable. Use them to manage subroutine execution order.

We'll embed such a list of subroutines—a "runlist"—as an attribute of the same name in a class `Model` where each of the subroutines is a method of the class and a method `execute_runlist` will go through the list of subroutines, executing them in order. A skeleton definition for such a class, for an oceanic general circulation model (GCM), might look like the following

[5] In general, you would do well to limit your interaction with `__dict__` to the built-in functions (e.g., `hasattr`) designed for such interactions. I confess, in my earlier days in using Python, I wrote a lot of code that directly accessed `__dict__`, but I now repent of what I did.

(note the runlist is not a complete listing of all routines in the model, but I list just a few to illustrate the idea):

```
1  class Model(object):
2      def __init__(self, *args, **kwds):
3          [...]
4          self.runlist = ['surface_fluxes', 'bottom_fluxes',
5                          'density', 'advection',
6                          'vertical_mixing', 'tracers']
7          [...]
8      def execute_runlist(self):
9          for imethodname in self.runlist:
10             f = getattr(self, imethodname)
11             f()
12     def surface_fluxes(self):
13         [... calculate surface fluxes ...]
14     def bottom_fluxes(self):
15         [... calculate bottom boundary fluxes ...]
16     [...]
```

Most of this code are placeholders (denoted by the square bracketed ellipses), but the `execute_runlist` method definition (lines 9–11) is complete (barring error checking) and bears comment. That method iterates through the `runlist` attribute list of subroutine (i.e., method) names, uses each name to retrieve the method itself, then executes the method. The variable `f` in the code is *the actual method* given by a string in the `runlist` attribute; the `getattr` function will give you the item attached to an object, regardless of whether it is an attribute or method (thus, `getattr` is somewhat misnamed). In this sense, objects actually only have attributes; it's just some attributes are data while others are functions that act on data. Once `f` is assigned to a method, the syntax `f()` calls the function, just like any other function call. (As we saw in Section 6.5, functions are objects in Python like any other object, and they can be stored, assigned, etc. So the `f()` call is no different than if I had typed `self.surface_fluxes()`, `self.bottom_fluxes`, etc.)

There are a variety of possible ways to use flexibility in subroutine execution order; here's one. Sometimes, the execution order of climate model subroutines affects model results. Thus, you might want to do a series of runs where subroutine execution order is shuffled. To do this using traditionally procedural languages, you would have to create separate versions of the source code and manually change the order of subroutine calling in each version of the code (then recompile, run, etc.). Using the `Model` class above, you would just create multiple instances of Model and create different versions of the runlist attribute where the order of the items in the list are shuffled.

How to do the list shuffling?[6] One way is to make use of the function permutations (from the itertools module) which will create an iterator that will step you through all permutations of the argument of permutations. Thus, this code:

Stepping through permutations.

```
a = itertools.permutations([0,1,2])
for i in a:
    print i
```

will print out all the different orderings of the list [0,1,2]:

```
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)
```

(Tuples result, so we will remember to use the list conversion function to give ourselves a list.)

We can apply this permutation function to the runlist attribute; instead of permuting a list of numbers, we will permute a list of strings. Each permutation will be set to the runlist attribute of the Model instance and executed. The code to do this would (basically) be:

```
import itertools
mymodel = Model([... input arguments ...])
runlist_copy = list(mymodel.runlist)
permute = itertools.permutations(runlist_copy)
for irunlist in permute:
    mymodel.runlist = list(irunlist)
    mymodel.execute_runlist()
```

Again, what have we done? By using lists and other Python helper functions on a model encapsulated in a class, we've created a series of model

[6]For more on shuffling and permutations in Python, see http://stackoverflow.com/questions/104420/how-to-generate-all-permutations-of-a-list-in-python (accessed August 10, 2012).

runs each of which executes the model's subroutines using one of the possible permutations of subroutine execution order. The lines of code needed to make this series of model runs is trivial (just 7). OOP, plus Python's powerful data structures and amazing library of modules, enables AOS users to easily use atmosphere and ocean models in ways that traditional methods of programming make difficult (or even impossible).

Python assignment is usually by reference rather than value.

An aside on assignment by reference vs. value: In line 3 above, I create a copy of the `runlist` attribute to make sure the `permutations` function is not acting on a list that will be changing in the loop. I do this because Python, for most variable types, including lists, does assignment by reference rather than value. Thus, the assignment in line 6 will propagate to all references to `mymodel.runlist`. By using the `list` function on `mymodel.runlist` in line 3, I make sure that `runlist_copy` is separate in memory from `mymodel.runlist`. Here's another example to make clearer the distinction between assignment by reference vs. value:

Example 53 (Assignment by reference vs. value):

Assignment by reference means that the assignment creates a pointer or alias to the memory location given by another variable while assignment by value means that the assignment creates a copy of that other variable and points to that copy. Consider the following lines of code:

```
>>> a = [1,2,3]
>>> b = a
>>> b[1] = 6.5
```

where I create a list `a`, assign the variable `b` to `a`, and then replace the oneth element of `b` with another value. Because the assignment of variable `b` to `a` is done by reference, not value, my replacement of the oneth element of `b` *also changes* the corresponding element of `a`. A print of `a` and `b` will show this:

```
>>> print b
[1, 6.5, 3]
>>> print a
[1, 6.5, 3]
```

Copying using deepcopy.

In other words, the `b = a` assignment did not create a copy of `a` but creates a pointer to the memory location of `a` and assigns the name `b` to that pointer. If what you wanted was for `b` to be an actual copy of `b`, you can use the `deepcopy` function of the copy module. Thus, this code:

```
>>> import copy
>>> a = [1,2,3]
>>> b = copy.deepcopy(a)
>>> b[1] = 6.5
>>> print b
[1, 6.5, 3]
>>> print a
[1, 2, 3]
```

as you can see, assigns b to a copy of a so any changes in b are separate from a, and vice versa.

Most datatypes in Python assign by reference. Simple datatypes like integers, floats, strings, etc. assign by value. Thus, for an integer scalar:

```
>>> a = 3
>>> b = a
>>> a = 6
>>> print b
3
```

we see that a change in a does not propagate to the variable b. (By the way, if you want to find the memory location of an object, you can use the id function. Two objects that both point to the same memory location should have the same id value.)

As a final aside: The use of "runlists" is only one way that an object encapsulation of atmosphere and ocean models can make those models more usable and powerful. I wrote a paper in *Geosci. Model Dev.* (Lin, 2009) that described such an object encapsulation for an intermediate-level tropical atmospheric circulation model and also demonstrated a hybrid Python-Fortran implementation of an atmospheric model; see http://www.geosci-model-dev.net/2/1/2009/gmd-2-1-2009.html if you're interested.

8.5 Summary

There's no denying it: object-oriented programming (OOP) is hard to learn. Anecdotal reports suggest even professional programmers need to work on around three OOP projects before they become proficient in OOP (Curtis, 1995). The dynamic nature of objects, however, permits one to do analysis in ways that would be much harder to do using traditional procedural programming. In this chapter, we saw three such AOS examples: Simpler handling of missing values and metadata, dynamic variable management, and dynamic subroutine execution ordering. OOP is not just a way of reorganizing data and functions, but a way of making more kinds of analysis possible for scientists to do. While Python works fine as a procedural language—so you can write Python programs similar to the way you would write Fortran, IDL, Matlab, etc. programs—the object-oriented aspects of Python provide some of the greatest benefit for AOS users. It's a steep learning curve, but well worth it.