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 7

An Introduction to OOP Using Python: Part I—Basic Principles and Syntax

7.1 What is object-oriented programming

Object-oriented programming (OOP), deservedly or not, has something of a reputation as an obtuse and mysterious way of programming. You may have heard of it, and even heard that it is a powerful way of writing programs, but you probably haven't heard a clear and concise description of how it works to help you write better AOS programs. Unfortunately, I also cannot give you a clear and concise description of how OOP works to help you program.

The problem is not that I cannot describe to you what an object is or give you a definition of OOP, but rather that any description of the mechanics and use of OOP does not really capture how OOP makes your life easier as a scientist programmer. It's like thinking that a description of oil pigments and poplar surfaces will somehow enable you to "get" how the Mona Lisa works. For both OOP and art, you can't describe the forest in terms of the trees.

Really, the only way I know of to convey how OOP enables atmospheric and oceanic scientists to do better science using a computer is to give you many examples of its use. So, in this chapter, I'll do just that. After a brief description of the mechanics of OOP, we'll look at some simple examples and work through some more complex examples, including examples from the atmospheric and oceanic sciences. Through these examples, I hope to describe both how to write object-oriented programs as well as why object-oriented programs work the way they do.

7.1.1 Procedural vs. object-oriented programming

Procedural programs have data and functions as separate entities.

One good way of describing something new is to compare it with something old. Most atmospheric and oceanic scientists have had experience with procedural programming, so we'll start there. Procedural programs look at the world in terms of two entities, "data" and "functions." In a procedural context, the two entities are separate from each other. A function takes data as input and returns data as output. Additionally, there's nothing customizable about a function with respect to data. As a result, there are no barriers to using a function on various types of data, even inappropriately.

Real world objects have states and behaviors.

In the real world, however, we don't think of things or objects as having these two features (data and functions) as separate entities. That is, real world objects are not (usually) merely data nor merely functions. Real world objects instead have both "state" and "behaviors." For instance, people have state (tall, short, etc.) and behavior (playing basketball, running, etc.), often both at the same time, and, of course, in the same person.

The aim of object-oriented programming is to imitate this in terms of software, so that "objects" in software have two entities attached to them, states and behavior. This makes the conceptual leap from real-world to programs (hopefully) less of a leap and more of a step. As a result, we can more easily implement ideas into instructions a computer can understand.

7.1.2 The nuts and bolts of objects

Objects are made up of attributes and methods.

What do objects consist of? An object in programming is an entity or "variable" that has two entities attached to it: data and things that act on that data. The data are called attributes of the object, and the functions attached to the object that can act on the data are called methods of the object. Importantly, you *design* these methods to act on the attributes; they aren't random functions someone has attached to the object. In contrast, in procedural programming, variables have only one set of data, the value of the variable, with no functions attached to the variable.

Object instances are specific realizations of a class.

How are objects defined? In the real world, objects are usually examples or specific realizations of some class or type. For instance, individual people are specific realizations of the class of human beings. The specific realizations, or instances, differ from one another in details but have the same pattern. For people, we all have the same general shape, organ structure, etc. In OOP, the specific realizations are called object **instances,** while the common pattern is called a **class.** In Python, this common pattern or template is defined by the `class` statement.

So, in summary, objects are made up of attributes and methods, the structure of a common pattern for a set of objects is called its class, and specific realizations of that pattern are called "instances of that class."

Recall that all the Python "variables" we introduced earlier are actually objects. (In fact, basically everything in Python is an object.) Let's look at a number of different Python objects to illustrate how objects work.

7.2 Example of how objects work: Strings

Python strings (like nearly everything else in Python) are objects. Thus, built into Python, there (implicitly) is a class definition of the string class, and every time you create a string, you are using that definition as your template. That template defines both attributes and methods for all string objects, so whatever string you've created, you have that set of data and functions attached to your string which you can use. Let's look at a specific case:

Example 45 (Viewing attributes and methods attached to strings and trying out a few methods):

In the Python interpreter, type in:

```
a = "hello"
```

Now type: `dir(a)`. What do you see? Type `a.title()` and `a.upper()` and see what you get.

Solution and discussion: The `dir(a)` command gives a list of (nearly) all the attributes and methods attached to the object `a`, which is the string `"hello"`. Note that there is more data attached to the object than just the word "hello", e.g., the attributes `a.__doc__` and `a.__class__` also show up in the `dir` listing.

The `dir` command shows an object's attributes and methods.

Methods can act on the data in the object. Thus, `a.title()` applies the `title` method to the data of `a` and returns the string `"hello"` in title case (i.e., the first letter of the word capitalized); `a.upper()` applies the `upper` method to the data of `a` and returns the string all in uppercase. Notice these methods do not require additional input arguments between the parenthesis, because all the data needed is already in the object (i.e., `"hello"`).

Review of syntax for objects.

Let's do a quick review of syntax for objects. First, to refer to attributes or methods of an instance, you add a period after the object name and then put the attribute or method name. To set an attribute, the reference should be on the lefthand side of the equal sign; the opposite is the case to read an attribute. Method calls require you to have parentheses after the name, with or without arguments, just like a function call. Finally, methods can produce a return value (like a function), act on attributes of the object in-place, or both.

7.3 Exercise on how objects work: Strings

▷ **Exercise 20 (Strings and how objects work):**
In the Python interpreter, type in:

```
a = 'The rain in Spain.'
```

Given string a:

1. Create a new string b that is a but all in uppercase.
2. Is a changed when you create b?
3. How would you test to see whether b is in uppercase? That is, how could you return a boolean that is True or False depending on whether b is uppercase?
4. How would you calculate the number of occurrences of the letter "n" in a?

The upper, isupper, and count string methods.

Solution and discussion: Here are my solutions:

1. `b = a.upper()`
2. No, the upper method's return value is used to create b; the value of a is not changed in place.
3. Use the isupper method on the string object, i.e., b.isupper() will return True or False, accordingly.
4. `a.count('n')`

7.4 Example of how objects work: Arrays

While lists have their uses, in scientific computing, arrays are the central object. Most of our discussion of arrays has focused on functions that create and act on arrays. Arrays, however, are objects like any other object and have attributes and methods built-in to them; arrays are more than just a sequence of numbers. Let's look at an example list of all the attributes and methods of an array object:

Example 46 (Examining array object attributes and methods):

In the Python interpreter, type in:

```
a = N.reshape(N.arange(12), (4,3))
```

Now type: `dir(a)`. What do you see? Based on their names, and your understanding of what arrays are, what do you think some of these attributes and methods do?

Solution and discussion: The `dir` command should give you a list of a lot of stuff. I'm not going to list all the output here but instead will discuss the output in general terms.

We first notice that there are two types of attribute and method names: those with double-underscores in front and in back of the name and those without any pre- or post-pended double-underscores. We consider each type of name in turn.

Double-underscore attribute and method names.

A very few double-underscore names sound like data. The `a.__doc__` variable is one such attribute and refers to documentation of the object. Most of the double-underscore names suggest operations on or with arrays (e.g., add, div, etc.), which is what they are: Those names are of the methods of the array object that *define* what Python will do to your data when the interpreter sees a "+", "/", etc. Thus, if you want to redefine how operators operate on arrays, *you can do so.* It is just a matter of redefining that method of the object.

Single-underscore attribute and method names.

That being said, I do not, in general, recommend you do so. In Python, the double-underscore in front means that attribute or method is "very private." (A variable with a single underscore in front is private, but not as private as a double-underscore variable.) That is to say, it is an attribute or method that normal users should not access, let alone redefine. Python does not, however, do much to prevent you from doing so, so advanced users who need to access or redefine those attributes and methods can do so.

Public attributes and methods.

The non-double-underscore names are names of "public" attributes and methods, i.e., attributes and methods normal users are expected to access and (possibly) redefine. A number of the methods and attributes of `a` are duplicates of functions (or the output of functions) that act on arrays (e.g., `transpose`, `T`), so you can use either the method version or the function version.

And now let's look at some examples of accessing and using array object attributes and methods:

Example 47 (Using array attributes and methods):

In the Python interpreter, type in:

```
a = N.reshape(N.arange(12), (4,3))
print a.astype('c')
print a.shape
print a.cumsum()
print a.T
```

What do each of the `print` lines do? Are you accessing an attribute or method of the array?:

How to tell whether you are accessing an attribute or a method.

Solution and discussion: The giveaway as to whether we are accessing attributes or calling methods is whether there are parenthesis after the name; if not, it's an attribute, otherwise, it's a method. Of course, you could type the name of the method without parentheses following, but then the interpreter would just say you specified the method itself, as you did not *call* the method:

```
>>> print a.astype
<built-in method astype of numpy.ndarray object at
    0x20d5100>
```

(I manually added a linebreak in the above screenshot to fit it on the page.) That is to say, the above syntax prints the method itself; since you can't meaningfully print the method itself, Python's `print` command just says "this is a method."

The `astype` call produces a version of array `a` that converts the values of `a` into single-character strings. The `shape` attribute gives the shape of the array. The `cumsum` method returns a flattened version of the array where each element is the cumulative sum of all the elements before. Finally, the attribute `T` is the transpose of the array `a`.

Object versions of `astype`, `shape`, and `cumsum`.

While it's nice to have a bunch of array attributes and methods attached to the array object, in practice, I find I seldom access array attributes and find it easier to use NumPy functions instead of the corresponding array methods. One exception with regards to attributes is the `dtype.char` attribute; that's very useful since it tells you the type of the elements of the array (see Example 30 for more on `dtype.char`).

7.5 Exercise on how objects work: Arrays

▷ **Exercise 21 (More on using array attributes and methods):**

For all these exercises (except for the first one), do not use NumPy module functions; only use attributes or methods attached to the arrays. (Do these in order, since each builds on the preceding commands.)

1. Create a 3 column, 4 row *floating point* array named `a`. The array can have any numerical values you want, as long as all the elements are not all identical.
2. Create an array `b` that is a copy of `a` but is 1-D, not 2-D.
3. Turn `b` into a 6 column, 2 row array, *in place*.
4. Create an array `c` where you round all elements of `b` to 1 decimal place.

Solution and discussion: Here are array methods that one can use to accomplish the exercises:

The `reshape`, `ravel`, `resize`, and `round` function and methods.

1. `a = N.reshape(N.arange(12, dtype='f'), (3,4))`
2. `b = a.ravel()`
3. `b.resize((2,6))`
4. `c = b.round(1)`

Remember, methods need to be called or else they don't do anything; including the parentheses to specify the calling argument list tells the interpreter you're calling the method. In terms of the "output" of the method, some methods act like a function, returning their output as a return value. Other methods do their work "in-place," on the object the method is attached to; those methods do not typically have a return value.[1] The `resize` method is an example of a method that operates on the data in-place, which is why there is no equal sign (for assignment) associated with the method call. You can also make a method operate on an object in-place as well as output a return value.

7.6 Defining your own class

We had said that all objects are instances of a class, and in the preceding examples, we looked at what made up string and array instances, which tells us something about the class definitions for those two kinds of objects. How would we go about creating our own class definitions?

Defining a class using `class`.

Class definitions start with `class` statement. The block following the `class` line is the class definition. Within the definition, you refer to the instance of the class as `self`. So, for example, the instance attribute `data` is called `self.data` in the class definition, and the instance method named `calculate` is called `self.calculate` in the class definition (i.e., it is called by `self.calculate()`, if it does not take any arguments).

Defining methods and the `self` argument.

Methods are defined using the `def` statement. The first argument in any method is `self`; this syntax is how Python tells a method "make use of all the previously defined attributes and methods in this instance." However, you never type `self` when you call the method.

The `__init__` method.

Usually, the first method you define will be the `__init__` method. This method is called whenever you create an instance of the class, and so you usually put code that handles the arguments present when you create (or instantiate) an instance of a class and conducts any kind of initialization for the object instance. The arguments list of `__init__` is the list of arguments passed in to the constructor of the class, which is called when you use the class name with calling syntax.

Whew! This is all very abstract. We need an example! Here's one:

[1]This statement is not entirely correct. If you do set another variable, by assignment, to such a method call, that lefthand-side variable will typically be set to `None`.

Example 48 (Example of a class definition for a Book class):

This class provides a template for holding and manipulating information about a book. The class definition provides a single method (besides the initialization method) that returns a formatted bibliographic reference for the book. The code below gives the class definition and then creates two instances of the class (note line continuations are added to fit the code on the page):

```
 1  class Book(object):
 2      def __init__(self, authorlast, authorfirst, \
 3                   title, place, publisher, year):
 4          self.authorlast = authorlast
 5          self.authorfirst = authorfirst
 6          self.title = title
 7          self.place = place
 8          self.publisher = publisher
 9          self.year = year
10
11      def write_bib_entry(self):
12          return self.authorlast \
13              + ', ' + self.authorfirst \
14              + ', ' + self.title \
15              + ', ' + self.place \
16              + ':  ' + self.publisher + ', ' \
17              + self.year + '.'
18
19  beauty = Book( "Dubay", "Thomas" \
20               , "The Evidential Power of Beauty" \
21               , "San Francisco" \
22               , "Ignatius Press", "1999" )
23  pynut = Book( "Martelli", "Alex" \
24              , "Python in a Nutshell" \
25              , "Sebastopol, CA" \
26              , "O'Reilly Media, Inc.", "2003" )
```

Can you explain what each line of code does?

Solution and discussion: Line 1 begins the class definition. By convention, class names follow the CapWords convention (capitalize the first letter of every word). The argument in the class statement is a special object called

The object `object` and inheritance.

`object`. This has to do with the OOP idea of inheritance, which is a topic beyond the scope of this book. Suffice it to say that classes you create can inherit or incorporate attributes and methods from other classes. Base classes (class that do not depend on other classes) inherit from `object`, a special object in Python that provides the foundational tools for classes.

Notice how attributes and methods are defined, set, and used in the class definition: Periods separate the instance name `self` from the attribute and method name. So the instance attribute `title` is called `self.title` in the class definition. When you actually create an instance, the instance name is the name of the object (e.g., `beauty`, `pynut`), so the instance attribute `title` of the instance `beauty` is referred to as `beauty.title`, and every instance attribute is separate from every other instance attribute (e.g., `beauty.title` and `pynut.title` are separate variables, not aliases for one another).

Thus, in lines 4–9, I assign each of the positional input parameters in the `def __init__` line to an instance attribute of the same name. Once assigned, these attributes can be used anywhere in the class definition by reference to `self`, as in the definition of the `write_bib_entry` method.

Speaking of which, note that the `write_bib_entry` method is called with no input parameters, but in the class definition in lines 11–17, I still need to provide it with `self` as an input. That way, the method definition is able to make use of all the attributes and methods attached to `self`.

In lines 19–22, I create an instance `beauty` of the `Book` class. Note how the arguments that are passed in are the same arguments as in the `def __init__` argument list. In the last four lines, I create another instance of the `Book` class.

(The code of this example is in *course_files/code_files* in a file called *bibliog.py*.)

Now that we've seen an example of defining a class, let's look at an example of using instances of the `Book` class to help us better understand what this class does:

Example 49 (Using instances of `Book`):

Consider the `Book` definition given in Example 48. Here are some questions to test your understanding of what it does:

1. How would you print out the `author` attribute of the `pynut` instance (at the interpreter, after running the file)?

2. If you type `print beauty.write_bib_entry()` at the interpreter (after running the file), what will happen?

3. How would you change the publication year for the `beauty` book to `"2010"`?

Solution and discussion: My answers:

1. Type: `print pynut.author`. Remember that once an instance of `Book` is created, the attributes are attached to the actual instance of the class, not to `self`. The only time `self` exists is in the class definition.

2. You will print out the the bibliography formatted version of the information in `beauty`.

3. Type: `beauty.year = "2010"`. Remember that you can change instance attributes of classes you have designed just like you can change instance attributes of any class; just use assignment. (There is also a function called `setattr` that you can use to assign attributes. I'll talk about `setattr` in Section 8.2.)

7.7 Exercise on defining your own class

▷ **Exercise 22 (The `Book` class and creating an `Article` class):**
Here are the tasks:

1. Create another instance of the `Book` class using book of your choosing (or make up a book). Execute the `write_bib_entry` method for that instance to check if it looks like what you wanted.

2. Add a method `make_authoryear` to the class definition that will create an attribute `authoryear` and will set that attribute to a string that has the last name of the author and then the year in parenthesis. For instance, for the `beauty` instance, this method will set `authoryear` to `'Dubay (1999)'`. The method *should not have a return statement.*

3. Create an `Article` class that manages information about articles. It will be very similar to the class definition for `Book`, except publisher

and place information will be unneeded and article title, volume number, and pages will be needed. Make sure this class also has the methods `write_bib_entry` and `make_authoryear`.

Solution and discussion: Here are my answers:

1. Here's another instance of `Book`, with a call to the `write_bib_entry` method:

   ```
   madeup = Book("Doe", "John", "Good Book",
                 "Chicago", "Me Press", "2012")
   print madeup.write_bib_entry()
   ```

   This code will print the following to the screen:

   ```
   Doe, John, Good Book, Chicago:  Me Press, 2012.
   ```

2. The entire `Book` class definition, with the new method (and line continuations added to fit the code on the page), is:

   ```
   class Book(object):
       def __init__(self, authorlast, authorfirst, \
                    title, place, publisher, year):
           self.authorlast = authorlast
           self.authorfirst = authorfirst
           self.title = title
           self.place = place
           self.publisher = publisher
           self.year = year

       def make_authoryear(self):
           self.authoryear = self.authorlast \
                             + '(' + self.year +')'

       def write_bib_entry(self):
           return self.authorlast \
               + ', ' + self.authorfirst \
               + ', ' + self.title \
               + ', ' + self.place \
               + ':  ' + self.publisher + ', ' \
               + self.year + '.'
   ```

The new portion is lines 11–13. None of the rest of the class definition needs to change.

3. The class definition for `Article` (with line continuations added to fit the code on the page) is:

```
class Article(object):
    def __init__(self, authorlast, authorfirst, \
                 articletitle, journaltitle, \
                 volume, pages, year):
        self.authorlast = authorlast
        self.authorfirst = authorfirst
        self.articletitle = articletitle
        self.journaltitle = journaltitle
        self.volume = volume
        self.pages = pages
        self.year = year

    def make_authoryear(self):
        self.authoryear = self.authorlast \
                        + ’ (’ + self.year +’)’

    def write_bib_entry(self):
        return self.authorlast \
            + ’, ’ + self.authorfirst \
            + ’ (’ + self.year + ’):  ’ \
            + ’"’ + self.articletitle + ’," ’ \
            + self.journaltitle + ’, ’ \
            + self.volume + ’, ’ \
            + self.pages + ’.’
```

This code looks nearly the same as that for the `Book` class, with these exceptions: some attributes differ between the two classes (books, for instance, do not have journal titles) and the method `write_bib_entry` is different between the two classes (to accommodate the different formatting between article and book bibliography entries). See *bibliog.py* in *course_files/code_files* for the code.

7.8 Making classes work together to make complex programming easier

Summary of introduction to OOP.

So in our introduction to object-oriented programming (OOP), we found out that objects hold attributes (data) and methods (functions that act on data) together in one related entity. Realizations of an object are called instances. The template or form for an object is called a class, so realizations are instances of a class. In Python, the `class` statement defines the template for object instances. In the `class` statement, instances of the class are called `self`. Once a real instance of the class is created, the instance (object) name itself is "substituted" in for `self`.

But so what? It seems like classes are just a different way of organizing data and functions: Instead of putting them in libraries (or modules), you put them in a class. If you're thinking that this isn't that big of a deal, I would agree that it isn't a big deal, if all you do in a program is write a single class with a single instance of that class; in that case, OOP does not buy you very much.

The real power of OOP, rather, comes when objects are used in conjunction with other classes. By properly designing your set of classes, the object-oriented structure can make your code much simpler to write and understand, easier to debug, and less prone to error. In the remaining sections of the chapter, we'll look at two case studies illustrating the use of OOP in this manner. The first case study extends our `Book` and `Article` classes by examining the more general program of how to create a bibliography. In the second case study, we consider how to create a class for geosciences work that "manages" a surface domain.

7.9 Case study 1: The bibliography example

The `Book` and `Article` classes we wrote earlier manage information related to books and articles. In this case study, we make use of `Book` and `Article` to help us implement one common use of book and article information: the creation of a bibliography. In particular, we'll write a `Bibliography` class that will manage a bibliography, given instances of `Book` and `Article` objects.

7.9.1 Structuring the `Bibliography` class

Since a bibliography consists of a list of (usually formatted) book and article entries, we will want our `Bibliography` class to contain such a list. Thus, the Bibliography class has, as its main attribute, a list of entries which are instances of Book and Article classes. Remember, instances of Book and Article can be thought of as books and articles; the instances are the "objects" that specific books and articles are.

Next, we write methods for Bibliography that can manipulate the list of Book and Article instances. To that end, the first two methods we write for Bibliography will do the following: initialize an instance of the class; rearrange the list alphabetically based upon last name then first name. The initialization method is called __init__ (as always), and the rearranging method will be called sort_entries_alpha. Here is the code:

```
1  import operator
2
3  class Bibliography(object):
4      def __init__(self, entrieslist):
5          self.entrieslist = entrieslist
6
7      def sort_entries_alpha(self):
8          tmp = sorted(self.entrieslist,
9                       key=operator.attrgetter('authorlast',
10                                              'authorfirst'))
11         self.entrieslist = tmp
12         del tmp
```

Let's talk about what this code does. In the __init__ method, there is only a single argument, entrieslist. This is the list of Book and Article instances that are being passed into an instance of the Bibliography class. The __init__ method assigns the entrieslist argument to an attribute of the same name.

Lines 7–12 define the sort_entries_alpha method, which sorts the entrieslist attribute and replaces the old entrieslist attribute with the sorted version. The method uses the built-in sorted function, which takes a keyword parameter key that gives the key used for sorting the argument of sorted.

The attrgetter function and sorted.

How is that key generated? The attrgetter function, which is part of the operator module, gets the attributes of the names listed as arguments to attrgetter out of the elements of the item being sorted. (Note that the attribute names passed into attrgetter are strings, and thus you refer to the attributes of interest by their string names, not by typing in their names. This makes the program much easier to write.) In our example, attrgetter has two arguments; sorted indexes self.entrieslist by the attrgetter's first argument attribute name first then the second.

Note that at the end of the `sort_entries_alpha` method definition, I use the `del` command to make sure that `tmp` disappears. I need to do this because lists are mutable, and Python assignment is by reference not value (see p. 140 for more discussion on reference vs. value). If I do not remove `tmp`, the `tmp` might float around as a reference to the `entrieslist` attribute; it shouldn't, but I'm paranoid so I explicitly deallocate `tmp` to make sure.

Some final comments: First, if you would like to read more on sorting in Python, please see http://wiki.python.org/moin/HowTo/Sorting. The `sorted` function is very versatile.

Second, there are some diagnostics at the end of *bibliog.py* that are run if you type:

```
python bibliog.py
```

Basic testing of programs.

from the operating system command line. This is one way of writing a very basic test to make sure that a module works. (Python has a solid unit testing framework in the form of the unittest module, if you're interested in something more robust.) These diagnostics, however, are not implemented if you import *bibliog.py* as a module. This is due to the conditional:

```
if __name__ == '__main__':
```

which is true only if the module is being run as a main program, i.e., by the `python` command. If you import the module for use in another module, by using `import`, the variable `__name__` will not have the string value `'__main__'`, and the diagnostics will not execute.

7.9.2 What `sort_entries_alpha` illustrates about OOP

Let's pause to think for a moment about the method `sort_entries_alpha`. What have we just done? First, we sorted a list of items that are totally differently structured from each other based on two shared types of data (attributes). Second, we did the sort using a sorting function that does not care about the details of the items being sorted, only that they had these two shared types of data. In other words, the sorting function doesn't care about the source type (e.g., article, book), only that all source types have the attributes `authorlast` and `authorfirst`.

Comparing OOP vs. procedural for a sorting example.

This doesn't seem that big a deal, but think about how we would have had to do it in traditional procedural programming. First, each instance would have been an array, with a label of what kind of source it is, for instance:

```
nature_array = ["article", "Smith", "Jane",
                "My Nobel prize-winning paper",
                "Nature", "481", "234-236", "2012"]
```

The procedural sorting function you'd write would need know which elements you want to sort with (here the second and third elements of the array). *But the index for every array of data would potentially be different,* depending on where in the array that data is stored for that source type. Thus, in your sorting function, you'd need to run multiple `if` tests (based on the source type) to extract the correct field in the array to sort by. But, if you changed the key you're sorting by (e.g., from the author's name to the date of publication), you would have to *change the element index you're sorting against.* This means manually changing the code of the `if` tests in your sorting function.

It's easy to make such a manual code change and test that the change works, if you only have a few source types (e.g., articles and books), but what if you have tens or hundreds of source types? What a nightmare! And as you make all those code changes, think of the number of possible bugs you may introduce just from keystroke errors alone! But in object-oriented programming, you can switch the sorting key at will and have an infinite number of source types *without any additional code changes* (e.g., no `if` tests to change). This is the power of OOP over procedural programming: code structured using an OOP framework naturally results in programs that are much more flexible and extensible, resulting in dramatically fewer bugs.

7.9.3 Exercise in extending the `Bibliography` class

▷ **Exercise 23 (Writing out an alphabetically sorted bibliography):**

Since we programmed `Book` and `Article` with `write_bib_entry` methods, let's take advantage of that. Write a method `write_bibliog_alpha` for the `Bibliography` class we just created that actually writes out a bibliography (as a string) with blank lines between the entries, with the entries sorted alphabetically by author name. The bibliography should be returned using a `return` statement in the method. Some hints:

- Elements of a list do not have to all have the same type.
- `for` loops do not only loop through lists of numbers but through any iterable. This includes lists of *any sort*, including lists of objects (such as `Book` and `Article` instances.
- Strings are immutable, so you cannot append to an existing string. Instead, do a reassignment combined with concatenation (i.e., `a=a+b`).

- To initialize a string, in order to grow it in concatenation steps such as in a `for` loop, start by setting the string variable to an empty string (which is just `''`).

Solution and discussion: Here is the solution for the entire class, with the new method included:

```
1  import operator
2
3  class Bibliography(object):
4      def __init__(self, entrieslist):
5          self.entrieslist = entrieslist
6
7      def sort_entries_alpha(self):
8          tmp = sorted(self.entrieslist,
9                 key=operator.attrgetter('authorlast',
10                                        'authorfirst'))
11         self.entrieslist = tmp
12         del tmp
13
14     def write_bibliog_alpha(self):
15         self.sort_entries_alpha()
16         output = ''
17         for ientry in self.entrieslist:
18             output = output \
19                    + ientry.write_bib_entry() + '\n\n'
20         return output[:-2]
```

The only code that has changed compared to what we had previously is the `write_bibliog_alpha` method; let's talk about what it does. Line 14 defines the method; because `self` is the only argument, the method is called with an empty argument list. The next line calls the `sort_entries_alpha` method to make sure the list that is stored in the `entrieslist` attribute is alphabetized. Next, we initialize the output string `output` as an empty string. When the "+" operator is used, Python will then use string concatenation on it. Lines 17–19 run a `for` loop to go through all elements in the list `entrieslist`. The output of `write_bib_entry` is added one entry at a time, along with two linebreaks after it. Finally, the entire string is output except for the final two linebreaks. (Remember that strings can be manipulated using list slicing syntax.)

7.9.4 What the `write_bibliog_alpha` method illustrates about OOP

Here too, let's ask how would we have written a function that wrote out an alphabetized bibliography in procedural programming? Probably something like the following sketch:

```
def write_bibliog_function(arrayofentries):
    [open output file]

    for i in xrange(len(arrayofentries)):
        ientryarray = arrayofentries[i]
        if ientryarray[0] = "article":
           [call function for bibliography entry
             for an article, and save to output file]
        elif ientryarray[0] == "book":
           [call function for bibliography entry
             for an book, and save to output file]
        [...]

     [close output file]
```

This solution sketch illustrates how in procedural programming we are stuck writing `if` tests in the bibliography writing function to make sure we format each source entry correctly, depending on source type (e.g., article, book). In fact, for *every* function that deals with multiple source types, we need this tree of `if` tests. If you introduce another source type, you need to add another `if` test in *all functions* where you have this testing tree. This is a recipe for disaster: It is exceedingly easy to inadvertently add an `if` test in one function but forget to do so in another function, etc.

In contrast, with objects, adding another source type *requires no code changes or additions.* The new source type just needs a `write_bib_entry` method defined for it. And, since methods are *designed* to work with the attributes of their class, this method will be tailor-made for its data. So much easier!

7.10 Case study 2: Creating a class for geosciences work—Surface domain management

I think the bibliography example in Section 7.9 does a good job of illustrating what object-oriented programming gives you that procedural programming cannot. I also like the example because all of us have had to write a bibliography, and the idea of "sources" (books, articles) very nicely lends itself to being thought of as an "object." But can the OOP way of thinking help us in decomposing a geosciences problem? In this section, we consider a class for managing surface domains (i.e., a latitude-longitude domain). I present the task of defining the class as an exercise and give two possible solutions. The exercise and solutions, while valuable in and of themselves, offer a nice illustration of how OOP enables atmospheric and oceanic scientists to write more concise but flexible code for handling scientific calculations.

THIS REVISION IS MUCH BETTER THAN YOUR 37TH. WE'LL JUST SEND IT OUT FOR ANOTHER ROUND OF PEER REVIEW ...
I DON'T BRAKE FOR
EDITOR
Licensed under a Creative Commons BY-NC-SA 3.0 license
Johnson & Lin 4-08
www.johnny-lin.com/comic

Science Publication Hell

▷ **Exercise 24 (Defining a `SurfaceDomain` class):**

Define a class `SurfaceDomain` that describes surface domain instances. The domain is a land or ocean surface region whose spatial extent is described by a latitude-longitude grid. The class is instantiated when you provide a vector of longitudes and latitudes; the surface domain is a regular grid based on these vectors. Surface parameters (e.g., elevation, temperature, roughness, etc.) can then be given as instance attributes. The quantities are given on the domain grid.

In addition, in the class definition, provide an **instantiation** method that saves the input longitude and latitude vectors as appropriately named attributes and creates 2-D arrays of the shape of the domain grid which have the longitude and latitude values at each point and saves them as private attributes (i.e., their names begin with a single underscore).

Hint: An example may help with regards to what I'm asking for with respect to the 2-D arrays. If `lon=N.arange(5)` and `lat=N.arange(4)`, then the `_lonall` instance attribute would be:

```
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
```

and the `_latall` instance attribute would be:

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

Solution and discussion: The two solutions described below (with the second solution commented out) are in *course_files/code_files* in the file *surface_domain.py*). Here's the solution using `for` loops:

```
1  import numpy as N
2
3  class SurfaceDomain(object):
4      def __init__(self, lon, lat):
5          self.lon = N.array(lon)
6          self.lat = N.array(lat)
7
8          shape2d = (N.size(self.lat), N.size(self.lon))
9          self._lonall = N.zeros(shape2d, dtype='f')
10         self._latall = N.zeros(shape2d, dtype='f')
11         for i in xrange(shape2d[0]):
12             for j in xrange(shape2d[1]):
13                 self._lonall[i,j] = self.lon[j]
14                 self._latall[i,j] = self.lat[i]
```

Lines 5–6 guarantee that `lon` and `lat` are NumPy arrays, in case lists or tuples are passed in.

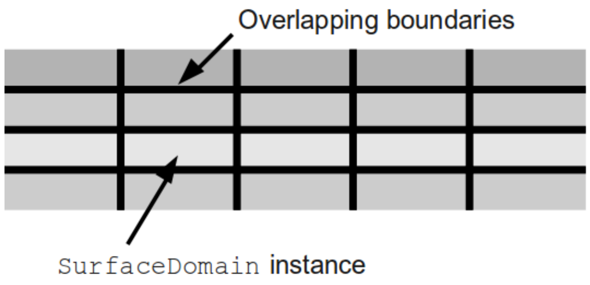
And here's a simpler and faster solution using the `meshgrid` function in NumPy instead of the `for` loops:

Using `meshgrid`.

```
import numpy as N

class SurfaceDomain(object):
    def __init__(self, lon, lat):
        self.lon = N.array(lon)
        self.lat = N.array(lat)
        [xall, yall] = N.meshgrid(self.lon, self.lat)
        self._lonall = xall
        self._latall = yall
        del xall, yall
```

So, what does this `SurfaceDomain` class illustrate about OOP applied to the geosciences? Pretend you have multiple `SurfaceDomain` instances that you want to communicate to each other, where the bounds of one are taken from (or interpolated with) the bounds of another, e.g., calculations for each domain instance are farmed out to a separate processor, and you're stitching domains together:



In the above schematic, gray areas are `SurfaceDomain` instances and the thick, dark lines are the overlapping boundaries between the domain instances.

Comparing OOP vs. procedural for a subdomain management example.

In procedural programming, to manage this set of overlapping domains, you might create a grand domain encompassing all points in all the domains to make an index that keeps track of which domains abut one another. The index records who contributes data to these boundary regions. Alternately, you might create a function that processes only the neighboring domains, but this function will be called from a scope that has access to all the domains (e.g., via a common block).

But, to manage this set of overlapping domains, you don't really need such a global view nor access to all domains. In fact, a global index or a common block means that if you change your domain layout, you have to hand-code a change to your index/common block. Rather, what you actually need is only to be able to interact with your neighbor. So why not just write a method that takes *your neighboring* `SurfaceDomain` *instances as arguments* and alters the boundaries accordingly? That is, why not add the following to the `SurfaceDomain` class definition:[2]

```
class SurfaceDomain(object):
    [...]
    def syncbounds(self, northobj, southobj,
                   eastobj, westobj):
        [...]
```

Such a method will propagate to all `SurfaceDomain` instances automatically, once written in the class definition. Thus, you only have to write one (relatively) small piece of code that can then affect any number of layouts of `SurfaceDomain` instances. Again, object-oriented programming enables you to push the level at which you code to solve a problem down to a lower-level than procedural programming easily allows. As a result, you can write smaller, better tested bit of code; this makes your code more robust and flexible.

7.11 Summary

You could, I think, fairly summarize this chapter as addressing one big question: Why should an atmospheric or oceanic scientist bother with object-oriented programming? In answer, I suggest two reasons. First, code written using OOP is less prone to error. OOP enables you to mostly eliminate lengthy argument lists, and it is much more difficult for a function to accidentally process data it should not process. Additionally, OOP deals with long series of conditional tests much more compactly; there is no need to duplicate `if` tests in multiple places. Finally, objects enable you to test smaller pieces of your program (e.g., individual attributes and methods), which makes your tests more productive and effective.

Second, programs written using OOP are more easily extended. New cases are easily added by creating new classes that have the interface methods defined for them. Additional functionality is also easily added by just adding new methods/attributes. Finally, any changes to class definitions automatically propagate to all instances of the class.

Procedural for short programs; OOP for everything else.

For short, quick-and-dirty programs, procedural programming is still the better option; there is no reason to spend the time coding the additional OOP infrastructure. But for many atmospheric and oceanic sciences applications,

[2]Christian Dieterich's PyOM pythonized OM3 ocean model does a similar kind of domain-splitting handling in Python.

things can very quickly become complex. As soon as that happens, the object decomposition can really help. Here's the rule-of-thumb I use: For a one-off, short program, I write it procedurally, but for any program I may extend someday (even if it is a tentative "may"), I write it using objects.