Johnny Wei-Bing Lin

A Hands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences

http://www.johnny-lin.com/pyintro

2012

 Printed version: ISBN 978-1-300-07616-2. PDF versions: No ISBNs are assigned.

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.

Regardless of whether or not you paid money for your copy of the book, you are free to use any and all parts of the book under the terms of the CC BY-NC-SA.

Chapter 3

Basic Data and Control Structures

Python, like any other programming language, has variables and all the standard control structures. As a multi-paradigm language, however, Python has data and control structures not commonly found in languages traditionally used by AOS users. In this chapter, I will describe Python's basic data and control structures that support procedural programming. By the end of this chapter, you should be able to write Fortran programs in Python ☺.

3.1 Overview of basic variables and operators

Python is dynamically typed.

Unlike languages like Fortran, Python is dynamically typed, meaning that variables take on the type of whatever they are set to when they are assigned. Thus, `a=5` makes the variable `a` an integer, but `a=5.0` makes the variable `a` a floating point number. Additionally, because assignment can happen anytime during the program, this means you can change the type of the variable without changing the variable name.

The built-in variable types are as you would guess, along with a few others. Here's a partial list of some of the most important basic types:

- Integer (short and long) and floating point (float)
- Strings
- Booleans
- NoneType
- Lists and tuples
- Dictionaries

The first three items are probably familiar to you, but NoneType, lists, tuples, and dictionaries might not be. I'll be talking about all these types as we go along in this chapter.

Arithmetic and comparison operators.

Arithmetic operators are as you would guess: (+, -, /, *, ** for addition, subtraction, division, multiplication, and exponentiation, respectively), as are comparison operators (>, <, >=, <=, !=, == for greater than, less than, greater than or equal to, less than or equal to, not equal, and equal, respectively).

Python is case-sensitive.

Please note that Python is case-sensitive, so "N" and "n" are different.

Example 4 (Create and use some numerical variables):

Open up a Python interpreter by typing `python` in a terminal window or use the Python Shell in IDLE. Type these lines in in the interpreter:

```
a = 3.5
b = -2.1
c = 3
d = 4
a*b
b+c
a/c
c/d
```

What did you find?

Solution and discussion: You should have gotten something like this:

```
>>> a = 3.5
>>> b = -2.1
>>> c = 3
>>> d = 4
>>> a*b
-7.350000000000005
>>> b+c
0.8999999999999999
>>> a/c
1.1666666666666667
>>> c/d
0
```

Remember Python is dynamically typed: It automatically decides what type a variable is based on the value/operation. Thus, `a` and `b` are floats and `c` and `d` are integers.

For operations, Python will generally make the output type the type that retains the most information. E.g., if one of the variables is float, a float variable is returned. However, if both variables are integers, integer division is performed, where the remainder is discarded. Thus `a/c` returns what you expect since `a` is float, but `c/d` does integer division and returns only the quotient (as an integer). Note that in Python 2.x, integers can be short or long; short has a size limit but long does not. In Python 3.x, all integers are long.

Python usually upcasts type if needed.

Here's a question: Why is the answer to `a*b` not exactly −7.35? Remember that floating point numbers on any binary computer are, in general, not represented exactly.[1] (This is why you should never do logical equality comparisons between floating point numbers; instead, you should compare whether two floating point numbers are "close to" each other. The NumPy array package has a function `allclose` that does this.) The default formatting setting for the `print` command, will sometimes print out enough of the portion after the decimal point to show that.

Binary floating point representations are inexact and the `allclose` function.

Let's take a look in more detail at the non-numeric built-in data types I listed before, i.e., strings, booleans, NoneType, lists and tuples, and dictionaries.

3.2 Strings

String variables are created by setting text in either paired single or double quotes (it doesn't normally matter which, as long as they are consistently paired), e.g.: `a = "hello"`.

Creating strings.

Some "special" strings include:

Special strings.

- `"\n"`: **newline character**
- `"\t"`: tab character
- `"\\"`: backslash

[1] See Bruce Bush's article "The Perils of Floating Point," http://www.lahey.com/float.htm (accessed March 17, 2012).

Triple quotes.

Python has a special construct called "triple quotes," i.e., quotation marks or apostrophes placed one after the other (`"""`), which delimit strings that are set to whatever is typed in between the triple quotes, (more or less) verbatim. This includes newline characters (but not backslashes), so this is an easy way to make strings with complex formatting.

Connecting strings.

Finally, Python uses the addition operator (+) to join strings together.

Example 5 (An operation with strings):

Try typing this in a Python interpreter:

```
a = "hello"
b = "there"
a + b
```

What did you get? Also try: `print a + b`.

Solution and discussion: The first two lines set `a` and `b` as string variables with values set to the given strings. Because the addition sign concatenates two strings together, `a + b` will return the string `'hellothere'`. The `print` command gives you the same thing, but it does not include the quotes which show that the result of `a + b` is a string.

3.3 Booleans

Boolean variables are variables that can have only one of two values, one of which is considered "true" and the other of which is considered "false." In some languages, the integer value zero is considered false and the integer value one is considered true. Older versions of Python also followed that convention (and this still works arithmetically); in recent versions of Python, there are two special values called `True` and `False` that serve as the values a boolean variable can take. (Note the capitalization matters.) Logical operators that operate on boolean variables are mainly as expected: `and`, `or`, `not`, etc.

`True` and `False` are Python's boolean values.

Example 6 (Operations with boolean variables):

Try this in a Python interpreter:

```
a = True
b = False
print a and b
print a or b
print 4 > 5
```

What did you get?

Solution and discussion: The first two lines assign `a` and `b` as boolean variables. The first two `print` statements return `False` and `True`, respectively. Remember that `and` requires both operands to be `True` in order to return `True`, while `or` only requires one of the operands be `True` to return `True`. Note that comparison operators (i.e., `4 > 5`) yield booleans, so the final `print` line returns `False`.

3.4 NoneType

This is a data type you probably have not seen before. A variable of NoneType can have only a single value, the value `None`. (Yes, the word "None," capitalized as shown, is defined as an actual value in Python, just like `True` and `False`.)

The `None` value.

Example 7 (Operations with NoneType):
Try this in a Python interpreter:

```
a = None
print a is None
print a == 4
```

What did you get?

Solution and discussion: The first `print` statement will return `True` while the second `print` statement will return `False`.

The `is` operator compares "equality" not in the sense of value (like `==` does) but in the sense of memory location. You can type in "`a == None`",

Logical equality and `is`.

the better syntax for comparing to `None` is "`a is None`".[2] The `a == 4` test is false because the number 4 is not equal to `None`.

Using `None` to safely initialize a parameter.

So what is the use of a variable of NoneType? I use it to "safely" initialize a parameter. That is to say, I initialize a variable to `None`, and if later on my program tries to do an operation with the variable before the variable has been reassigned to a non-NoneType variable, Python will give an error. This is a simple way to make sure I did not forget to set the variable to a real value. Remember variables are dynamically typed, so replacing a NoneType variable with some other value later on is no problem!

3.5 Lists and tuples

Lists are mutable ordered sequences.

Lists are ordered sequences. They are like arrays (in Fortran, IDL, etc.), except each of the items in the list do not have to be of the same type. A given list element can also be set to anything, even another list. Square brackets ("`[]`") **delimit** (i.e., start and stop) a list, and commas between list elements separate elements from one another. If you have a one element list, put a comma after the element.

List element indices start with 0.

List element addresses start with zero, so the first element of list `a` is `a[0]`, the second is `a[1]`, etc. IDL follows this convention but Fortran does not. Because the ordinal value (i.e., first, second, third, etc.) of an element differs from the address of an element (i.e., zero, one, two, etc.), when we refer to an element by its address we will append a "th" to the end of the address. That is, the "zeroth" element by address is the first element by position in the list, the "oneth" element by address is the second element by position, the "twoth" element by address is the third element by position, and so on.

The `len` function returns the length of lists and tuples.

Finally, the length of a list can be obtained using the `len` function, e.g., `len(a)` to find the length of the list `a`.

Example 8 (A list):

Type in the following in the Python interpreter:

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

[2] The reason is a little esoteric; see the web page http://jaredgrubb.blogspot.com/2009/04/python-is-none-vs-none.html if you're interested in the details (accessed August 16, 2012).

What is `len(a)`? What does `a[1]` equal to? How about `a[3]`? `a[3][1]`?

Solution and discussion: The `len(a)` is 4, `a[1]` equals 3.2, `a[3]` equals the list `[-1.2, 'there', 5.5]`, and `a[3][1]` equals the string `'there'`. I find the easiest way to read a complex reference like `a[3][1]` is from left to right, that is, "in the threeth element of the list `a`, take the oneth element."

Referencing list elements that are lists.

In Python, list elements can also be addressed starting from the end; thus, `a[-1]` is the last element in list `a`, `a[-2]` is the next to last element, etc.

Indexing from the end of a sequence.

You can create new lists that are slices of an existing list. Slicing follows these rules:

Slicing rules.

- Element addresses in a range are separated by a colon.
- The lower limit of the range is *inclusive,* and the upper limit of the range is *exclusive.*

Example 9 (Slicing a list):

Consider again the list `a` that you just typed in for Example 8. What would `a[1:3]` return?

Solution and discussion: You should get the following if you print out the list slice `a[1:3]`:

```
>>> print a[1:3]
[3.2, 'hello']
```

Because the upper-limit is exclusive in the slice, the threeth element (i.e., the fourth element) is not part of the slice; only the oneth and twoth (i.e., second and third) elements are part of the slice.

Lists are **mutable** (i.e., you can add and remove items, change the size of the list). One way of changing elements in a list is by assignment (just like you would change an element in a Fortran, IDL, etc. array):

Example 10 (Changing list element values by assignment):

Let's go back to the list in Example 8:

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

How would we go about replacing the value of the second element with the string `'goodbye'`?

Solution and discussion: We refer to the second element as `a[1]`, so using variable assignment, we change that element by:

```
a[1] = 'goodbye'
```

The list `a` is now:

```
[2, 'goodbye', 'hello', [-1.2, 'there', 5.5]]
```

Python lists, however, also have special "built-in" functions that allow you to insert items into the list, pop off items from the list, etc. We'll discuss the nature of those functions (which are called **methods**; this relates to object-oriented programming) in detail in Ch. 7. Even without that discussion, however, it is still fruitful to consider a few examples of using list methods to alter lists:

Example 11 (Changing lists using list methods):
Assume we have the list we defined in Example 8:

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

What do the following commands give you when typed into the Python interpreter?:

- `a.insert(2,'everyone')`
- `a.remove(2)`
- `a.append(4.5)`

The `insert`, `remove`, and `append` methods for lists.

Solution and discussion: The first command `insert` inserts the string `'everyone'` into the list after the twoth (i.e., third) element of the list. The second command `remove` removes the first occurrence of the value given in the argument. The final command `append` adds the argument to the end of the list.

For the list `a`, if we printed out the contents of `a` after each of the above three lines were executed one after the other, we would get:

```
[2, 3.2, 'everyone', 'hello', [-1.2, 'there', 5.5]]
[3.2, 'everyone', 'hello', [-1.2, 'there', 5.5]]
[3.2, 'everyone', 'hello', [-1.2, 'there', 5.5], 4.5]
```

Tuples are nearly identical to lists with the exception that tuples cannot be changed (i.e., they are **immutable**). That is to say, if you try to insert an element in a tuple, Python will return an error. Tuples are defined exactly as lists except you use parenthesis as delimiters instead of square brackets, e.g., `b = (3.2, 'hello')`.

Tuples are immutable ordered sequences.

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

Slicing strings as if each character were a list element.

3.6 Exercises with lists and tuples

Remember that exercises are no less necessary than examples to attempt! The only real difference between exercises and examples is that the former are more complex than the latter; pedagogically speaking, both types of problems are used in a similar way, and in my discussion of both examples and exercises, I will often introduce new topics.

▷ **Exercise 4 (Making and changing a list):**

1. Take your street address and make it a list variable `myaddress` where each token is an element. Make numbers numbers and words strings.
2. What would be the code to set the sum of the numerical portions of your address list to a variable called `address_sum`?
3. What would be the code to change one of the string elements of the list to another string (e.g., if your address had "West" in it, how would you change that string to "North")?

Solution and discussion: We give the solutions for each of the questions above:

1. For my work address, the `myaddress` list is:

```
myaddress = [3225, 'West', 'Foster', 'Avenue',
             'Chicago', 'IL', 60625]
```

Line continuation in Python.

Note that when you type in a list in Python, you can break the list after the completion of an element and continue the list on the next line, and Python will automatically know the list is being continued (leading blank spaces are ignored). In general, however, you continue a line of code in Python by putting a backslash ("\") at the end of a line, with nothing after the backslash. Thus, you can also enter the above list by typing in:

```
myaddress = [3225, 'West', 'Foster', 'Avenue', \
             'Chicago', 'IL', 60625]
```

2. This sets the sum of the numerical portions to `address_sum`:

```
address_sum = myaddress[0] + myaddress[-1]
```

3. Code to change "West" to "North":

```
myaddress[1] = "North"
```

▷ **Exercise 5 (Altering the order of a list's elements):**
Take the list you created in Exercise 4 and change the street portion of `myaddress` to have the street first and the building number at the end. Hints: Make use of assignments and slicing.

Solution and discussion: To change the street portion of `myaddress` and view the result:

```
a = myaddress[0]
b = myaddress[1:3]
myaddress[0:2] = b
myaddress[2] = a
print myaddress
```

Assigning sublists.

Note that you can assign sublists of a list in one fell swoop if the value on the right can be parsed element-wise (e.g., is also a list of the same length).

3.7 Dictionaries

Definition of a dictionary.

Like lists and tuples, dictionaries are also collections of elements, but dictionaries, instead of being ordered, are *unordered* lists whose elements are referenced by *keys*, not by position. Keys can be anything that can be uniquely

named and sorted. In practice, keys are usually integers or strings. Values can be anything. (And when I say "anything," I mean *anything*, just like lists and tuples. We'll see in Ch. 6 a little of the broad range of values that dictionaries can hold and how that is a useful feature.) Dictionaries are very powerful; this one data structure revolutionized my code.

Curly braces ("{}") delimit a dictionary. The elements of a dictionary are "key:value" pairs, separated by a colon. Dictionary elements are referenced like lists, except the key is given in place of the element address. The example below will make this all clearer:

Example 12 (A dictionary):

Type the following in the Python interpreter:

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

For the dictionary `a`:

- What does `a['b']` return?
- What does `a['c'][1]` return?

Solution and discussion: `a['b']` returns the floating point number `3.2`. `a['c']` returns the list `[-1.2, 'there', 5.5]`, so `a['c'][1]` returns the oneth element of that list, the string `'there'`.

Like lists, dictionaries come with "built-in" functions (methods) that enable you to find out all the keys in the dictionary, find out all the values in the dictionary, etc. In Ch. 7, when we introduce OOP, we'll discuss the nature of methods in detail, but even without that discussion, it is still useful to consider a few examples of dictionary methods:

Example 13 (A few dictionary methods):

Assume we have the dictionary from Example 12 already defined in the Python interpreter:

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

If you typed the following into the Python interpreter, what would you get for each line?:

- `d = a.keys()`
- `d = a.values()`
- `a.has_key('c')`

The `keys`, `values`, and `has_key` methods.

Solution and discussion: The first line executes the command `keys`, which returns a list of all the keys of `a`, and sets that list to the variable `d`. The second command does this same thing as the first command, except `d` is a list of the values in the dictionary `a`. The third command tests if dictionary `a` has an element with the key `'c'`, returning `True` if true and `False` if not. For the dictionary `a`, the first line returns the list `['a', 'c', 'b']` and sets that to the variable `d` while the second line returns `True`.

Do *not* assume dictionaries are stored in any particular order.

Note that the `keys` and `values` methods do *not* return a sorted list of items. Because dictionaries are *unordered* collections, you *must not* assume the key:value pairs are stored in the dictionary in any particular order. If you want to access the dictionary values in a specific order, you should first order the dictionary's keys (or, in some cases, values) in the desired order using a sorting function like `sorted`. (Section 7.9.1 gives an example of the use of `sorted`.)

3.8 Exercises with dictionaries

▷ **Exercise 6 (Create a dictionary):**

Create a dictionary `myaddress` using your address. Choose relevant keys (they will probably be strings), and separate your address into street address, city, state, and postal code portions, all of which are strings (for your ZIP Code, don't enter it in as a number).

Solution and discussion: For my work address:

```
myaddress = {'street':'3225 West Foster Avenue',
             'city':'Chicago', 'state':'IL',
             'zip':'60625'}
```

As with lists and tuples, I don't need to specify the line continuation character if I break the line in-between the specifications for each element.

▷ **Exercise 7 (Using elements from a dictionary):**
Create a variable `full_address` that is the concatenation of all the elements of the `myaddress` variable from Exercise 6; in your concatenation, include commas and blank spaces as needed. Hint: Remember that commas and blanks can be made into strings.

Solution and discussion: Here's my solution for my `myaddress` from Exercise 6:

```
full_address = myaddress['street'] + ', ' \
             + myaddress['city'] + ', ' \
             + myaddress['state'] + '  ' \
             + myaddress['zip']
```

Notice how when I choose keys that have a clear meaning, in this case labels like "street" and "city," my references to the values in the dictionary associated with those keys read sensibly: `myaddress['street']` makes more sense than `myaddress[0]`. This is one benefit of dictionaries over lists and tuples.

Dictionaries allow you to choose meaningful keys.

3.9 Functions

Functions in Python, in theory, work *both* like functions and subroutines in Fortran, in that (1) input comes via arguments and (2) output occurs through: a return variable (like Fortran functions) and/or arguments (like Fortran subroutines). In practice, functions in Python are written to act like Fortran functions, with a single output returned. (The return value is specified by the `return` statement.) If you want multiple returns, it's easier to put them into a list or use objects.

Functions work like Fortran functions and subroutines.

Function definitions begin with a `def` statement, followed by the name of the function and the argument list in parenthesis. The contents of the function after this `def` line are indented in "x" spaces (where "x" is a constant). Usually, people indent 4 spaces. (In practice, if you use a development environment like IDLE, or the Python mode in vi or Emacs, you don't have to add the indentation yourself; the environment does it for you.) Example 14 below shows how the indentation works to indicate what lines are inside the function.

All lines in a statement block are usually indented in 4 spaces.

Important side note: All block structures in Python use indentation to show when they begin and end. This convention is in lieu of "end" lines like `end do` and `end if` in Fortran. For those of us who have had experience using the fixed-form format of Fortran 77, this will seem like a bad idea. For now, just trust me that this indentation convention actually makes your code clearer, more readable, and more concise.

Example 14 (A function):

Type the following in a file (remember to use four spaces for the indentation instead of a tab or another number of spaces):

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

What happens if you run the file? Why did nothing happen? Now type the following in the interpreter or Python Shell:

```
a = area(3)
print a
```

What happens?

Functions are defined by `def` and used by calling.

Solution and discussion: In the first case, nothing happened because you only defined the function; the function was not called. In the second case, you call the function, set the return value to the variable `a`, and print out `a` to the screen. Note that in such a simple function, you could have skipped creating the local variable `area` and just typed:

```
return 3.14 * (radius**2)
```

which would have evaluated the expression prior to the return.

Note how the indentation shows that the two lines of code after the `def` line are the lines inside the function (i.e., they are the code that does the function's tasks).

Positional and keyword arguments.

As we said earlier, inputs to a function, in general, come in via the argument list while the output is the return value of the function. Python accepts both positional and keyword arguments in the argument list of a function: Positional arguments are usually for *required* input while keyword arguments

are usually for *optional* input. Typically, keyword arguments are set to some default value. If you do not want to have a default value set for the keyword, a safe practice is to just set the keyword to `None`.

Example 15 (A function with both positional and keyword arguments):
Type the following in a file:

```
def area(radius, pi=None):
    area = pi * (radius**2)
    return area
a = area(3)
```

What happens if you run the file?

Solution and discussion: You should have received an error like this (note I handwrapped the last line to make it fit on the page):

```
Traceback (most recent call last):
  File "example.py", line 4, in <module>
    a = area(3)
  File "example.py", line 2, in area
    area = pi * (radius**2)
TypeError: unsupported operand
           type(s) for *: 'NoneType' and 'int'
```

Because in your `a = area(3)` call you did not define the keyword argument `pi`, when the function was called, it used the default value of `None` for `pi`. When the function tried to execute an arithmetic operation using `pi`, an error was raised and execution was transferred to the main program level, where execution finally stopped.

If you type in the following in the interpreter or Python Shell, after first executing the code in *yourfilename.py* (where *yourfilename.py* is the name of the file in which you defined the `area` function):[3]

```
a = area(3, pi=3.14)
print a
```

[3]Recall you execute a file either by typing in `python -i yourfilename.py` at the command-line or by running the module in IDLE.

you will get a print-to-screen of the answer, 28.26. Upon the call of `area`, the value of 3.14 was set to `pi` in the function.

Traditionally, Fortran and similar procedural language programmers have had to deal with the problem of lengthy and unwieldy argument lists: If you want to pass in 30 variables, your argument list has 30 variables in it. (It is not surprising to see such subroutine calls in a climate model.) A list of such a length is an undetected error waiting to happen; one typing slip and you could be passing in surface roughness instead of humidity!

A simpler, compact way of passing in lists of arguments.

Python has a nifty way of passing in lists of positional and keyword arguments in one fell swoop by considering a list of positional arguments as a list/tuple and a collection of keyword arguments as a dictionary. You can then use all of Python's built-in list and dictionary methods to manage your function's arguments, with the function's **calling** line only having two variables. This example illustrates how to do this:

Example 16 (Passing in lists of positional and keyword arguments):

Try typing in the following in the same file where you defined the version of `area` with both positional and keyword arguments (i.e., the version in Example 15):

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

then run your file from the Unix command line by:

```
python -i yourfilename.py
```

(or using the shell in IDLE). Remember to put these lines *after* your definition of `area`; otherwise, you will not have an `area` function to refer to ☺.

Solution and discussion: This code should work exactly the same as Example 15, that is:

```
a = area(*args, **kwds)
```

works the same as:

```
a = area(3, pi=3.14)
```

where `args` and `kwds` are given as above. You will get a print-to-screen of the answer, 28.26.

Example 16 illustrates the following rules for passing in function arguments by lists and dictionaries:

- In the function call, put an asterisk (*) before the list that contains the positional arguments and put two asterisks before the dictionary that contains the keyword arguments.
- The list of positional arguments is a list where each element in the list is a positional argument to be passed in, and the list is ordered in the same order as the positional arguments.
- The dictionary of keyword arguments uses string keys corresponding to the name of the keyword and the value of the key:value pairs as the value set to the keyword parameter.

3.10 Logical constructs

`if` statements.

The syntax for if-statements is

`if` <*condition*>`:`

followed by the block of code to execute if <*condition*> is true. Because indentation delimits the contents of the `if` block, there is no need for an "endif" line.

Example 17 (A compound `if` statement):

The compound `elif` statement.

Type the following in a file:

```
a = 3
if a == 3:
    print 'I am a ', a
elif a == 2:
    print 'I am a 2'
else:
    print 'I am not a 3 or 2'
```

First guess what you think will happen if you run the file (what do you think `elif` does? `else`?) then run the file. What did you get? What would you need to change to get `I am a 2` or `I am not a 3 or 2` to be output to the screen?

Solution and discussion: Because `a = 3`, the first `if` test will test true and `I am a  3` will be printed to the screen.

The `elif` statement is used after the first test and means "else if", or "if the previous `if` was not true, consider this `if`". You can have any number of `elif`s after the initial `if` and the compound `if` statement will go through each one, testing the line's condition and executing what is in the block under the `elif` line if true (then exiting the compound `if`) and going on to the next test if false.

The `else` executes if none of the other `if` and `elif` statements tested true and is ignored otherwise.

Remember the colon after `if`, etc.!

Don't forget the colon at the end of `if`, `elif`, and `else` statements! It's easy to forget them ☺ (same with the `def` statement for defining functions).

3.11 Looping

3.11.1 Looping a definite number of times

The standard loop in Python begins with `for` and has the syntax:

`for` <*index*> `in` <*list*>`:`

The `for` loop goes through a sequence of items.

followed by the contents of the loop. (Don't forget the colon at the end of the `for` line.) The `for` loop is kind of different compared to the Fortran `do` loops you might be familiar with. In Fortran, IDL, etc. you specify a beginning value and an ending value (often 1 and an integer n) for an index, and the loop runs through all integers from that beginning value to that ending value, setting the index to that value. In Python, the loop index runs through a list of items, and the index is assigned to each item in that list, one after the other, until the list of items is exhausted.

Example 18 (A `for` loop):

Type the following in a file (remember to indent 4 spaces in the second line):

```
for i in [2, -3.3, 'hello', 1, -12]:
    print i
```

Run the file. What did you get?

Solution and discussion: The code prints out the elements of the list to screen, one element per line:

```
2
-3.3
hello
1
-12
```

This means `i` changes type as the loop executes. It starts as an integer, becomes floating point, then becomes a string, returns to being an integer, and ends as an integer.

Recall that elements in a Python list can be of *any* type, and that list elements do not all have to be of the same type. Also remember that Python is dynamically typed, so that a variable will change its type to reflect whatever it is assigned to at any given time. Thus, in a loop, the loop index could, potentially, be changing in type as the loop runs through all the elements in a list, which was the case in Example 18 above. Since the loop index does not have to be an integer, it doesn't really make sense to call it an "index;" in Python, it's called an iterator. Note too that since the iterator is not just a number, but an object, you have access to all of the **attributes** and methods of its class (again, more on this in Ch. 7).

Iterators are different than Fortran looping indices.

A lot of the time you will loop through lists. Technically, however, Python loops can loop through any data structure that is **iterable,** i.e., a structure where after you've looked at one element of it, it will move you onto the next element. Arrays (which we'll cover in Ch. 4) are another iterable structure.

You can loop through any iterable.

In Fortran, we often loop through arrays by the addresses of the elements. So too in Python, often, you will want to loop through a list by list element addresses. To make this easier to do there is a built-in function called `range` which produces such a list: `range(n)` returns the list $[0, 1, 2, \ldots, n-1]$.

The `range` function makes a list of indices.

Example 19 (A `for` loop using the `range` function):

Type the following in a file:

```
a = [2, -3, 'hello', 1, -12]
for i in range(5):
    print a[i]
```

Run the file. What did you get?

Solution and discussion: This code will give the exact same results as in Example 18. The function call `range(5)` produces the list:

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

which the iterator `i` runs through, and which is used as the index for elements in list `a`. Thus, `i` is an integer for every step of the loop.

3.11.2 Looping an indefinite number of times

The `while` loop.

Python also has a `while` loop. It's like any other while loop and begins with the syntax:

`while` <*condition*>`:`

The code block (indented) that follows the `while` line is executed while <*condition*> evaluates as `True`. Here's a simple example:

Example 20 (A `while` loop):

Type in the following into a file (or the interpreter):

```
a = 1
while a < 10:
    print a
    a = a + 1
```

What did you get?

Solution and discussion: This will print out the integers one through ten, with each integer on its own line. Prior to executing the code block underneath the `while` statement, the interpreter checks whether the condition (`a < 10`) is true or false. If the condition evaluates as `True`, the code block executes; if the condition evaluates as `False`, the code block is not executed. Thus:

```
a = 10
while a < 10:
    print a
    a = a + 1
```

will do nothing. Likewise:

```
a = 10
while False:
    print a
    a = a + 1
```

will also do nothing. In this last code snippet, the value of the variable `a` is immaterial; as the condition is always set to `False`, the `while` loop will never execute. (Conversely, a `while True:` statement will never terminate. It is a bad idea to write such a statement ☺.)

Please see your favorite Python reference if you'd like more information about `while` loops (my reference suggestions are given in Ch. 10). Because they are not as common as their `for` cousins (at least in AOS applications), I won't spend exercise time on them.

3.12 Exercises on functions, logical constructs, and looping

▷ **Exercise 8 (Looping through a list of street address elements):**
Take the list of the parts of your street address from Exercise 4. Write a loop that goes through that list and prints out each item in that list.

Solution and discussion: My street address list was:

```
myaddress = [3225, 'West', 'Foster', 'Avenue', \
             'Chicago', 'IL', 60625]
```

The following loop will do the job:

```
for i in myaddress:
    print i
```

as will this loop:

```
for i in range(len(myaddress)):
    print myaddress[i]
```

Remember the built-in `len` function returns the length of the list that is its argument; the length is an integer and is the argument passed into the `range` call. Note also that the type of `i` behaves differently in the two loops. Python is dynamically typed!

▷ **Exercise 9 (Looping through a list of temperatures and applying a test):**

Pretend you have the following list of temperatures `T`:

```
T = [273.4, 265.5, 277.7, 285.5]
```

and a list of flags called `Tflags` that is initialized to all `False`. `Tflags` and `T` are each the same size. Thus:

```
Tflags = [False, False, False, False]
```

Write a loop that checks each temperature in `T` and sets the corresponding `Tflags` element to `True` if the temperature is above the freezing point of water.

Solution and discussion: The following loop will do the job:

```
for i in range(len(T)):
    if T[i] > 273.15:
        Tflags[i] = True
```

Remember I'm assuming both `T` and `Tflags` are already defined before I enter the loop.

▷ **Exercise 10 (A function to loop through a list of temperatures and apply a test):**

Turn your answer to Exercise 9 into a function. Assume that `T` is the input argument and `Tflags` is what you will want to return from the function. A hint: You can create a four-element list whose values all equal `False` by typing `[False]*4`. Thus:

```
Tflags = [False]*4
```

does the same thing as:

```
Tflags = [False, False, False, False]
```

Also, you may want to use the `range` and `len` functions at some point in your code.

Solution and discussion: The following function will do the job:

```
def temptest(T):
    Tflags = [False]*len(T)
    for i in range(len(T)):
        if T[i] > 273.15:
            Tflags[i] = True
    return Tflags
```

3.13 Modules

Python calls libraries "**modules**" and "**packages,**" where a package is a collection of modules. (Usually, people say "module" to refer to both modules and packages, since they're used very similarly. I'll do that for most of this book.) Unlike compiled languages like Fortran, however, these modules are not collections of object files but rather regular Python source code files. A module is a single source code file and a package is a directory containing source code files (and possibly subdirectories of source code files).

Modules and packages.

To **import** a module, Python provides a command called `import`, and its syntax is:

Importing a module.

`import` <*module name*>

Let's look at an example to help our discussion of how to import and use Python modules:

Example 21 (Importing a module):

To import a module called NumPy (this is Python's array package, which we'll talk about in-depth in Ch. 4), type:

```
import numpy
```

Type this in the Python interpreter. It should execute without any message being printed to the screen.

Referring to functions, etc. in a module.

Once a module is imported, you can use functions, variables, etc. defined in the module by referring to the imported name (here `numpy`), putting a period ("`.`"), and the name of the function, variable, etc. that was defined in the module. Thus, to use the `sin` function in the package, refer to `numpy.sin`.

Try typing:

```
a = numpy.sin(4)
```

This will return the sine of 4 and set it to the variable `a`. Print out `a` to check this worked correctly; it should equal −0.756, approximately (I truncated most of the digits that the `print` statement provides).

Importing and namespaces.

What `import` essentially does is to run the code file that has the filename of *<module name>.py*. When `import` runs that code file, it runs the file in its own little "interpreter." This "interpreter," however, is not a separate Python session but occurs in the current session within a variable named after the module's name. That is to say, the `import` executes the module's code in its own **namespace;** that namespace is a variable with the same name as the module's name.

For `import numpy`, the filename that would be run is *numpy.py* and the contents of that file would be run in the namespace `numpy`. (This isn't quite what happens for NumPy, because NumPy is technically a package of many files, not a module of a single file, but the principle is the same.) If all the module file does is define functions, variables, etc., then nothing will be output. But you have access to everything that is defined by typing the module name, a period, then the name of the module function, variable, etc. you want (hence, `numpy.sin`, etc.). Just as in a regular Python session you have access to all the variables, functions, etc. you define in that regular session, with an imported module, all the variables, functions, etc. that the module created and used are also sitting inside the module's namespace, ready for you to access, using the syntax just mentioned.

Referring to submodules.

Submodules (which are subdirectories inside the package directory) are also specified with the periods. For instance, NumPy has a submodule called `ma`, which in turn has special functions defined in it. The submodule then is referred to as `numpy.ma` and the `array` function in the submodule as `numpy.ma.array`.

(As an aside, sometimes the name of a module as written or spoken is different from name that goes in the `import` command: NumPy is the module name, but the namespace is `numpy`, the Scientific Python package has

a namespace `Scientific`, and so on. This is confusing, but unfortunately some modules are this way.)

How namespaces prevent collisions.

The idea of a namespace for Python modules helps protect against collisions. In Fortran, you have to be careful you do not duplicate function and subroutine names when you compile against multiple libraries, because if there is a function of the same name in two libraries, one of those will be overwritten. With Python modules, this kind of collision cannot occur, because functions are attached to modules by name through the imported module's namespace. It is, however, possible to defeat this feature and cause collisions if you really want to (e.g., by having duplicate module names in *PYTHONPATH* directories or improper use of `from ... import`), which is why I am teaching you the safe way of importing rather than the risky way ☺.

Sometimes, if you use a module a lot, you will want refer to it by a shorter name. To do this, use the `import` <*module*> `as` <*alias*> construct, for instance:

```
import numpy as N
```

Then, `N.sin` is the same as `numpy.sin`.

Modules can contain data variables in addition to functions.

Finally, remember, modules can contain data in addition to functions. The syntax to refer to those module data variables is exactly the same as for functions. Thus, `numpy.pi` gives the value of the mathematical constant π.

3.14 A brief introduction to object syntax

While we introduced some elements of objects with Section 3.5 on lists and tuples and Section 3.7 on dictionaries, and while we're saving a rigorous introduction to objects for Ch. 7), at this time, having talked about the syntax for modules, we should briefly introduce a little of what objects are and how in Python to refer to objects and their parts.

The key idea of objects is that variables shouldn't be thought of as having only values (and type), but rather they should be thought of entities that can have any number of other things "attached" to them. If the attached thing is a piece of data, it's called an attribute of the object variable. If the attached thing is a function, it's called a method.

Referring to object attributes and methods.

From a syntax viewpoint, if you have an object variable that has many things attached to it, the question is how to refer to those attached things. In Python, the key syntax idea is borrowed from module syntax: Just as you describe functions attached to modules by giving the module name, a period,

then the function name, you describe things attached to a Python object by giving the variable name, a period, then the attribute or method name.

The syntax of operations with or by attributes and methods should also seem familiar to you: If you want to call an object's methods, you use the same syntax as for functions (i.e., with a calling list given by parenthesis); attributes of objects are in turn read and set attributes just like they were regular variables (i.e., with an equal sign).

Thus, for instance, if I have a list object `mylist`, and I want to use one of the methods attached to that object (e.g., `reverse`), I would type in `mylist.reverse()`. This method reverses all the elements in the object, in place, and so it does not require the passing in of any arguments: The data to reverse is in `mylist` itself (note the empty argument list between the parenthesis).

Using `dir` to see what is attached to an object.

If you can attach attributes and methods to an object, you will want a way of viewing all the attributes and methods that are attached. A good interactive development environment will give nicely formatted ways to do this, but if all you have is a Python interpreter, type `dir(x)`, where `x` is the object name, to list (approximately) all attributes and methods attached to an object.

Lastly, as a teaser, here's an idea I want you to think about prior to our introduction of objects in Ch. 7: Nearly everything in Python is an object. Everything. Thus, what I've been calling variables (integers, floats, strings, lists, etc.) are not variables in the traditional Fortran, IDL, Matlab, etc. sense but instead objects. Even functions are objects. This feature of Python will end up having profound implications and will enable us to write programs we never could in other languages traditionally used in the atmospheric and oceanic sciences.

3.15 Exercise that includes using a module

▷ **Exercise 11 (Using functions from a module to do calculations on data):**

Pretend you have the list of temperatures `T` you saw earlier:

```
T = [273.4, 265.5, 277.7, 285.5]
```

Write code that will calculate the average of the maximum and minimum of `T`. Hint: The NumPy package has a `max` function and a `min` function that can look through a list of numerical values and return the maximum and

minimum value, respectively. The single argument they take is the list of numerical values.

Solution and discussion: Here's code that will do the trick:

```
import numpy
T = [273.4, 265.5, 277.7, 285.5]
maxT = numpy.max(T)
minT = numpy.min(T)
avg_max_min = 0.5 * (maxT + minT)
```

3.16 Exception handling

Throwing exceptions and `raise`.

In traditional Fortran, one common way of checking for and processing program error states is to write an "if" test for the error state and then execute a `stop` statement to stop program execution and output an informative message. In Python, you can accomplish the same thing with the `raise` statement: If you want the program to stop when you have an error, you throw an **exception** with a `raise` statement. Here's an example:

Example 22 (Using `raise`):

Consider the function `area` we defined in Example 15. How would we put in a test to ensure the user would not pass in a negative radius? One answer: We could put in an `if` test for a negative radius and if true, execute a `raise` statement:

```
def area(radius, pi=None):
    if radius < 0:
        raise ValueError, 'radius negative'
    area = pi * (radius**2)
    return area
```

The syntax for `raise` is the command `raise` followed by an exception class (in this case I used the built-in exception class `ValueError`, which is commonly used to denote errors that have to do with bad variable values), then a comma and a string that will be output by the interpreter when the `raise` is thrown.

Exceptions are not the same as Fortran `stop` statements.

Raising an exception is not exactly the same as a Fortran `stop` statement (though sometimes it will act the same). In the latter, program execution stops and you are returned to the operating system level. In the former, an exception stops execution and sends the interpreter up one level to see if there is some code that will properly handle the error. This means that in using `raise`, you have the opportunity to gracefully handle expected errors without causing the entire program to stop executing.

In Example 22, we saw how to create an exception, but I didn't show you how to handle the exception. That is, I didn't show you how in Python to tell the interpreter what to do if a routine it calls throws an exception. The `try/except` statement is Python's exception handler. You execute the block under the `try`, then execute the `except`s if an exception is raised. Consider this example:

Example 23 (Handling an exception):

Assume we have the function `area` as defined in Example 22 (i.e., with the test for a negative radius). Here is an example of calling the function `area` using `try/except` that will gracefully recognize a negative radius and call `area` again with the absolute value of the radius instead as input:

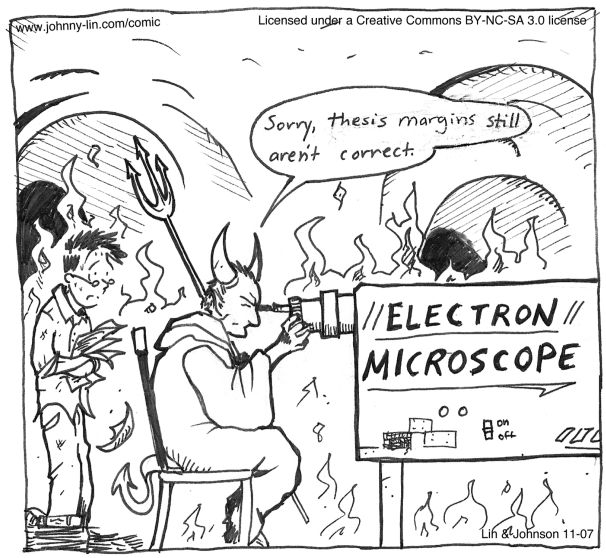
```
rad = -2.5
try:
    a = area(rad, pi=3.14)
except ValueError:
    a = area(abs(rad), pi=3.14)
```

How the interpreter processes a `try/except` block.

When the interpreter enters the `try` block, it executes all the statements in the block one by one. If one of the statements returns an exception (as the first `area` call will because `rad` is negative), the interpreter looks for an `except` statement at the calling level (one level up from the first `area` call, which is the level of calling) that recognizes the exception class (in this case `ValueError`). If the interpreter finds such an `except` statement, the interpreter executes the block under that `except`. In this example, that block repeats the `area` call but with the absolute value of `rad` instead of `rad` itself. If the interpreter does not find such an `except` statement, it looks another level up for a statement that will handle the exception; this occurs all the way up to the main level, and if no handler is found there, execution of the entire program stops.

In the examples in this section, I used the exception class `ValueError`. There are a number of built-in exception classes which you can find listed in a good Python reference (e.g., `TypeError`, `ZeroDivisionError`, etc.) and which you can use to handle the specific type of error you are protecting against.[4] I should note, however, the better and more advanced approach is to define your own exception classes to customize handling, but this topic is beyond the scope of this book.

Exception classes.



Ph.D. Student Hell

3.17 Summary

In many ways, basic Python variable syntax and control structures look a lot like those in traditional compiled languages. However, Python includes a number of additional built-in data types, like dictionaries, which suggest there will be more to the language than meets the eye; in the rest of this book, we'll find those data structures are *very* powerful (just building up the suspense level ☺). Other features of Python that usually differ from traditional compiled languages include: Python variables are dynamically typed, so they can change type as the program executes; indentation whitespace is significant; imported module names organize the namespace of functions and

[4] See http://docs.python.org/library/exceptions.html for a listing of built-in exception classes (accessed August 17, 2012).

module data variables; and exceptions can be handled using the `try/except` statement.

The Python style guide.

Finally, seeing all this Python code may make you wonder whether there is a standard style guide to writing Python. Indeed, there is; it's called PEP 8 (PEP stands for "Python Enhancement Proposal") and is online at http://www.python.org/dev/peps/pep-0008.