Eight better steps to Python proficiency
Anth/Biol 5221, 12 September 2011

You’ve done a few introductory lab exercises and know something about the elements of the Python language. This document briefly reviews some central principles. Reading it may help you consolidate what you’ve learned so far mostly by example, and it may also help you prepare for what’s to come.

1. **Representing stuff with values, variables and expressions.** Computers store information in sequences of binary “bits” (1 for “on” or 0 for “off”). Software then interprets these bit patterns as representations of the things we’re really interested in, which are most fundamentally numbers (like 42) and strings of text characters (like “spam”). The numbers are of two kinds: integers which can be represented exactly (the meaning of the universe is 00000000 00000000 00000000 01010101 in the standard 32-bit notation for integers), and floats which approximate real numbers (using a form of scientific notation that we won’t bother to explain). In the representation of a string, each character is often encoded as single 8-bit “byte” (in this code the first letter of “spam” is 01110011, but if interpreted as a number it would be 115).

Python does the interpretation for us so we don’t have to think about bits and bytes and such. But we do have to let Python know what kind of data “object” a given variable or other expression represents. A variable is just a name that Python associates with the value of a data object of a certain kind. If we type `x = 2`, then Python associates the name `x` with the integer value 2. But if we type `x = '2'` or `x = "2"`, then the value of `x` is the string 2, which is text, not a number of any kind.

Values are represented most simply by “literal” or “constant” values like 2, 4.0e-001, and 'spam', or they may be built up through the use of appropriate operators (as in `2 + 4.0e-001`) or functions (as in `sqrt(2)`), which has the value 1.41421...). If `x = "sp"` and `y = "am"`, then `x+y` is an expression that evaluates to "spam".

2. **Doing things to values with statements.** Values exist but they accomplish nothing until we instruct Python to change or use them in some way. When we say `y=sqrt(x)`, we are using an assignment statement to create the variable `y` and associate the value `sqrt(x)` with it. If we say `print 2.0*y`, we are again expressing a value (2y) and using a statement (print) to make Python show it on the screen. Note: We do not need to know in advance what the value of `x` will be! Programming allows us to write general algorithms for transforming wide ranges of inputs into equally wide ranges of outputs, according to the same fully specified rule.

3. **Organizing data structures with lists.** Simple variables such as `x` allow us to use the same expressions and statements to refer to many possible values, as with our old friend `y=sqrt(x)`. Similarly, compound variables such as `x[1]` allow us to refer to lists of values. Such collections may become very large, but the algorithms needed to operate on them remain simple if each element (or group of elements) in the data structure can be transformed by the same procedure.

Python has several kinds of compound variables, but the most important by far is the list, which you have already met (in section 3 of JEPy). Tuples and dictionaries are also important, and you should learn about them, but you won’t need to master them for this course.

In Python, lists are one-dimensional arrays of variables that may refer to objects of any kind. Most often, the members of a list are numbers or strings. For example,

```
x = [12, 9, 0, 4.2, 3]
words = ['parrot', 'dead', 'spam', 'eggs', 'grail']
```

But they don’t have to be all of the same kind!

```
answer = ['meaning', 42]
```

And most usefully, the members of a list can be other lists!

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

The elements of a list are referred to by indexes that begin at zero. What are the values of `my_matrix[0][0]` and `my_matrix[1][2]`? What is the value of `words[2]`? And what is the value of `words[2][3]`?
4. Iterating lists with **for statements**. We often want to step through the elements of a list. This could be done with a **while** loop that increments an index variable as in coin2.py on page 11 of *JEPy*. But it’s much simpler and more elegant to say

```python
for word in words:
    print word
```

For each of the five elements of `words`, sequentially, the `for` statement associates the new variable name `word` with that element. The `print` statement then causes that element to be printed to the screen.

The function `range(n)` creates the list of `n` consecutive integers beginning with 0. Thus `x = range(8)` assigns the list `[0,1,2,3,4,5,6,7]` to `x`. In the most popular looping construct, `for` iterates on a `range`:

```python
for i in range(5):
    print i, words[i]
```

The **list comprehension** (another feature unique to Python) is an implied `for` loop that creates and initializes a list. Thus `x = [i for i in range(8)]` is the same as `x = range(8), and

```python
my_matrix = [[1 + j + 3*i for j in range(3)] for i in range(3)]
```

creates the list of lists shown in section 5, above. Try this, with modifications, if the syntax isn’t clear.

5. **Addressing the pieces of a string with indexes and slices**. A string is a sequence of characters that can be referenced individually by subscripts, just like the elements of a list.

```python
>>> word = 'spam'
>>> print word[2]
a
```

But you can’t directly change the individual elements of a string, so `word[3] = 't'` won’t work. Instead, you can make a new string by extracting the first three characters with a **slice**, which can then be combined with the part to be changed. The expression for this value can then be assigned to the original variable name, if desired. Thus `word = word[0:3] + 't'` changes the value associated with `word` from 'spam' to 'spat'.

**Slice notation** is unique to Python, and it is extremely useful with lists as well as with strings. Any segment of any sequence-type expression can be referred to as `[<first position>:<last position + 1>]`. The value of `word[0:3]` is 'spa', and the value of `words[2:4]` is the list `['spam', 'eggs']`.

6. **Modularizing the work with functions**. These are self-contained blocks of Python-speak that do something and then deliver the result. Typically they take one or more input “arguments” and transform them into an output (usually a value, as in `y=sqrt(x)`, but sometimes a side effect such as printing something to the screen).

Some functions are built in to the core of Python (e.g., `int(x)`, `float(x)`, `abs(x)`). Some functions must be loaded into memory from modules (e.g., `random()` and `randint(i,j)` from the `random` module, or `sqrt(x)`, `log(x)` and `pow(x,y)` from the `math` module). And you can write your own, using the reserved keyword `def`, which may not be used for any other purpose. There’s an example near the end of *JEPy* (section 4.5, page 23).

```python
def var(list_of_nums):
    sum_x = float(sum(list_of_nums))
    mean_x = sum_x/float(len(list_of_nums))
    sum_dev_squared = 0.0
    for x in list_of_nums:
        sum_dev_squared += (x – mean_x)**2
    variance = sum_dev_squared/float(len(list_of_nums))
    return(variance)
```

If `my_data` is the list `[0, 1, 0, 1]` (or those four numbers in any order), then `var(my_data)` evaluates to 0.25. This is very handy, because you can use `var` at any number of places in your program just by “calling” it:

```python
my_variance = var(my_data)
```
7. Lists and strings do amazing things with their methods. These are much like functions, but they are tied more closely to properties of the object, and they sometimes modify the object instead of “returning” a new value derived from it. The methods available for lists are the most powerful and most frequently used. For example,

```python
>>> words.sort()
>>> words
['dead', 'eggs', 'grail', 'parrot', 'spam']
```

The value of `words` is now changed to a list of the same five strings, but in alphabetical order (the default, but other options exist). Lists of numbers can also be sorted (in numerical order, of course, not alphabetical order).

```python
>>> words.reverse()
>>> words
['spam', 'parrot', 'grail', 'eggs', 'dead']
```

Kind of obvious, I guess. Another useful list method is `append`, which adds its argument to the end of the list.

```python
>>> words.append('Brian')
>>> words
['spam', 'parrot', 'grail', 'eggs', 'dead', 'Brian']
```

Some methods are more function-like, in that they return a value derived from the object without changing it. The `count` method provides a good example:

```python
>>> words.count('eggs')
1
>>> words.count('cheese')
0
```

The `index` method can be extremely useful. It locates the first instance of a given element in the sequence:

```python
>>> j = words.index('eggs')
>>> print j
3
```

Strings also have the `count` and `index` methods (and many others – we’re just scratching the surface here).

```python
>>> inspired = 'We believe these truths to be self-evident'
>>> inspired.count('e')
10
>>> inspired.count('e ')
4
```

Note that in the second case the argument is a two-character substring. The `index` method for strings also works as you might hope:

```python
>>> inspired.index('e ')
1
```

But suppose we wanted to find the location of the third word ending in “e”? Fortunately, the `index` method will accept a second argument that tells it where in the sequence to start searching:

```python
>>> j = 0
>>> for i in range(3):
...     j = inspired.index('e ',j+1)
...     print j
1
9
15
```

This algorithm searches three times, beginning initially at position 1 (the first possible location of a word-ending “e”). Then in the subsequent iterations, it begins one position after the previous word-ending “e”. By this method it finds and reports the locations of the word-ending “e” in “We”, “believe” and “these”. Note that the algorithm would work on any text string with at least three words that end in “e” followed by a space.
8. **Controlling the flow with indenting.** You don’t just want things to happen – you want them to happen in a particular order, and often you want them to happen only if some particular condition is true. In Python, the “flow of execution” is controlled in part by the order in which statements occur within a block of statements (from top to bottom, running down the page, as in all programming languages). But flow is also controlled very importantly by indentation, and Python is almost unique in this respect. You’re already familiar with this feature of the language, but it always gives beginners fits, and it’s a frequent cause of bugs even for experts.

Problems often arise when there are several levels of looping and/or conditional execution, in a hierarchical pattern. For example, suppose we wanted to calculate the mean value of the largest random number in a set of ten random numbers. Let’s collect a million such “extreme values” and average them. This little program has been extensively commented to emphasize the hierarchically nested flow of execution (with comments that are themselves indented, to emphasize the point).

```python
# extremo.py  12 September 2011

# Import the random-number generator *before* it's needed.
from random import random

# Initialize memory variables for the extreme values *before* the case-loop begins.
sum_ex = 0.0
n_ex = 0

# Begin the 1000000 cases.
for case in range(1000000):
    # Initialize memory for the largest random number just before *each* set of 10.
    max_x = 0.0

    # Now draw the 10 random numbers in a set.
    for i in range(10):
        x = random()

        # Test to see if x is the largest value yet (in *this* set of 10).
        if x > max_x:
            # Update max_x only when this is true.
            max_x = x

    # NOW WE RETURN TO x = random(), ABOVE, UNLESS i == 9!
    # We fall out of the i-loop when 10 random numbers have been drawn (i == 9).
    # Now max_x is the largest random() in a set of 10.
    # Use it to update the sum, and increment the sample size.
    sum_ex += max_x
    n_ex += 1

    # NOW WE RETURN TO max_x = 0.0, ABOVE, UNLESS case == 999,999 (i.e., we’ve done 1M cases).

# We fall out of the case-loop when 1,000,000 sets of 10 have been drawn and processed.
# Note that the case-loop is the *outer* loop, which encloses the *inner* i-loop.
# Now we can calculate and print the mean, using the outer-loop memory variables.
mean_ex = sum_ex/float(n_ex)
print "Mean of %d largest random()s in a set of 10 is %7.5f" % (n_ex,mean_ex)
```

The third lab (*JEPy* section 4) is challenging largely because it requires careful construction of loops within loops, as in this program. By studying how this one works, you may get some insight into the general principle.

By the way, the answer I get is 0.90909. By thinking hard about the meaning of expectations, you may be able to explain this. Remember, `random()` returns a *uniformly* distributed number on the interval between 0 and 1, and in each set you collect 10 of them. *JEPr* may provide some help.