Ten Things Python Programmers Should Know

I used to program in Java, before I [transitioned](http://danieltakeshi.github.io/2012/08/05/java-to-python-transition/) to Python. And now that I’ve become a huge Python fan, I thought I’d provide ten basic but important facts or concepts about Python. All of these were useful to me within the past year as I made Python my primary programming language. If you’re just getting to grips with Python or am interested in seeing what this language has to offer, you might find this overview helpful. This entry does assume that you are comfortable with elementary programming terminology and Python syntax, such as understanding the role of [Python’s whitespace](http://www.secnetix.de/olli/Python/block_indentation.hawk).

I also make substantive use of code throughout this post, so just be aware that one-line comments in Python are preceded by a hashtag #. To comment a *block* of text in Python, we can wrap the text around with three quotation marks """ at the start and end. This method makes it easier to comment multiple lines, since we don’t have to put slashes at the start of each line.

Finally, before we get to the meat of the post, if one plans to seriously use Python, then it is essential to familiarize himself or herself with the following websites:

1. [Official Python Website](http://www.python.org/)
2. [Python 2 Documentation](http://docs.python.org/2/) (Ignore this if using only Python 3)
3. [Python 3 Documentation](http://docs.python.org/3/)
4. [Stack Overflow](http://stackoverflow.com/)

I’m including Stack Overflow because ever since it opened up, people have been asking an absurd amount of Python programming questions there. If you’ve got a basic syntax error, try to avoid asking the question on Stack Overflow since someone has probably done it already. In fact, in many cases, Stack Overflow has *become*the documentation. (But that’s a story for another day.)

That said, let’s discuss ten of some of the important concepts that Python programmers should know.

***1. Python Version Numbers***

While this is technically not a programming feature, it’s still crucial to know the current versions of Python just so we’re all on the same page. Python versions are numbered as A.B.C., where the three letters represent (in decreasing order) important changes in the language. So, for instance, going from 2.7.3 to 2.7.4 means that the Python build made some minor bug fixes, while going from 2.xx to 3.xx represents a major change. Note the ‘x’ here, which is intentional; if a Python feature can apply to version number 2.7.C for any valid ‘C’ value, then we put in an ‘x’ and refer to Python 2.7.x. We can also omit the ‘x’ entirely and just use 2.7.

As of this writing (July 2013), Python has two stable versions commonly used: 2.7 and 3.3. Less important is the third character, but right now 2.7.5 and 3.3.2 are the current versions, both of which were released on May 15, 2013. The short answer is that, while both 2.7 and 3.3 are perfectly fine to use, 3.3 is the future of the language and someone just starting Python today should probably use Python 3.3 over 2.7. Of course, if one is in the middle of an extensive research project that makes heavy use of 2.7, then it might not make sense to upgrade to 3.3 right away. This is actually quite similar to my current situation, since I’m using a good number of my own Python 2.7 scripts to help me analyze [algorithmic combinatorics on words](http://www.uncg.edu/cmp/reu/index.html). Once August arrives, I’ll fully transition to Python 3.3. In the meantime, though, I’ve done quite a bit of reading on Python 3’s new versions and I have 3.3.2 installed (in addition to 2.7.4) on my laptop, so this post and its code syntax will assume that we’re using Python 3.

One important thing to note, though, is that Python 3 is intentionally *backwards incompatible*. Backward compatibility is often a desired feature of programming languages and software that routinely undergo revisions, since it means that input from older versions (e.g. older Python programs) can still run under the latest builds. In this case, Python 2 code will *not* be guaranteed to run successfully if using Python 3, so some conversion may be necessary to allow code to properly run. Backwards incompatibility was necessary in order to allow Python 3 to be more clear, concise and use additional features.

In the meantime, what’s the difference between Python 2 and 3, anyway That’s beyond the scope of this post, but I’ve added in references at the end of the section based on the official Python documentation. I suppose the main improvement is that there’s better Unicode support, but there’s also been some minor fixes here and there, improving some of the annoying features of 2.7. Still, while there are enough changes to warrant a 2.x to 3.x change, Guido van Rossum says in his [overview](https://danieltakeshi.github.io/2013/07/05/ten-things-python-programmers-should-know/Nevertheless,%20after%20digesting%20the%20changes,%20you%E2%80%99ll%20find%20that%20Python%20really%20hasn%E2%80%99t%20changed%20all%20that%20much%20%E2%80%93%20by%20and%20large,%20we%E2%80%99re%20mostly%20fixing%20well-known%20annoyances%20and%20warts,%20and%20removing%20a%20lot%20of%20old%20cruft.) that:

*Nevertheless, after digesting the changes, you’ll find that Python really hasn’t changed all that much – by and large, we’re mostly fixing well-known annoyances and warts, and removing a lot of old cruft.*

A note: Guido van Rossum is Python’s creator, and still maintains his leadership over the programming language’s development, so if there’s something he says about Python, it can usually be taken as correct.

By the way, if you’re curious to see your version of Python, you can simply paste the following into a program:

import sys  
**print**("My version Number: {}"**.**format(sys**.**version))

Here, the text inside the quotation marks gets printed as it is, except for the brackets { and }, which transform into the sys.version or in other words, the Python version. This is classic *string formatting*.

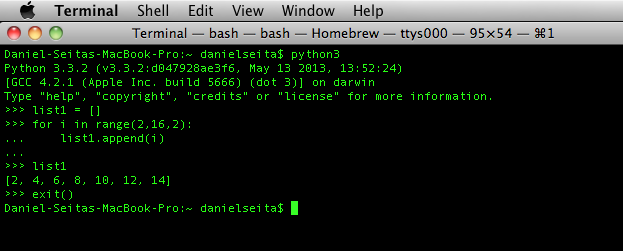
Alternatively, if you’re using a Linux or Mac computer, you can do the same stuff directly in the Python interpreter on the command line interpreter (i.e. Unix Shell) or the Mac OS X Terminal. Windows users will need to install third party software, such as [Cygwin](http://www.cygwin.com/), since there is no built-in command line interface. But in the meantime, this incredibly useful Terminal brings us to our next point …

***2. Using the Python Shell***

Without a doubt, one of the most useful aspects of Python is that it comes auto-installed with its own shell. The Python Shell can be executed by typing in python from the command line (in Windows, there should be an application that you can just double-click). By doing so, you’ll see the default version number, a copyright notice, and three arrows (or “r-angles” in LaTeX-speak) >>> asking for your input. If you have multiple versions of Python installed, you may need to add in the version number python3.3 to get the correct version.

So why is the Python shell so useful? Simply put, it lets you test out simple commands in isolation. In many cases, you’ll be able to detect if there’s going to be a syntax or logical error in some command you want to use *before* it gets tested in some gigantic script that could consume memory or be time-intensive.

Below, I’ve included a screenshot of me performing some commands on my Macbook Pro’s Terminal, using the 3.3 Python shell.



This set of commands sets up an empty list (list1) and adds to it all the *even* integers in the range [2,16)[2,16).

For the sake of showing why this shell might be useful, suppose I had forgotten to put in the last parameter of range so I had typed in range(2,16) instead.  Then when I printed the contents of the list after the for loop, I would have seen all the numbers between 2 and 15 inclusive, rather than just the even numbers. But since I *want* only the even numbers in my “real” program that I’ve been working on, this would remind me to add in that last parameter. It’s a silly example, but it really shows how checking what you do in the shell before you insert it in a real program can be beneficial. I’ll be using the shell in some other code bits in this post, which you can recognize by the three “r-angles” >>>.

Other popular languages such as C++ and Java have their own versions of the Python shell, but I believe you would need to install something. And installing Linux-style programs *can* be nontrivial since often times there is no nice clickable GUI that can do the installation immediately.

***3. Using ‘os’ and ‘sys’***

I find both the [os](http://docs.python.org/3.3/library/os.html) and [sys](http://docs.python.org/3.3/library/sys) modules to be incredibly useful to me for the purposes of *conveniency* and *generality*.

First, let’s go over the sys module. Possibly the biggest advantage that it offers to the programmer is the use of command line inputs to the program. Say you’ve built a large program that will perform some task that depends on inputs from the user. For instance, in my machine learning class last semester, I implemented the [k-means clustering algorithm](http://en.wikipedia.org/wiki/K-means_clustering). This is a learning algorithm that is given data and can *classify* it into groups depending on how many clusters are given as input. It’s clear that this can be useful in many life applications. Someone who has standardized data on medical patients’ records (e.g. blood-sugar levels, height, weight, etc.) may want to classify patients into two “clusters,” which could be (1) healthy or (2) ill. Or perhaps there could be *n* clusters, where patients classified into lower numbered clusters have a better outlook than those with high numbers.

To perform k-means clustering, then, we logically need two inputs: (1) the data itself and (2) the number of clusters. One idea is to put these directly into the program, and then run it. But what happens if we want to keep changing the data file we’re using or the number of clusters? Each time the program has finished executing, we’d have to go back into our text editor and modify it before re-executing it.

A better way would be to use command line arguments. Changing inputs on the command line is usually faster than opening a text editor and retyping the variables. We can do this with [sys.argv](http://docs.python.org/3.3/library/sys#sys.argv), which takes in input from the command line. As an extra protection, one can also make sure that the user inputs the correct number of parameters. I have done this in the following code snippet from my k-means clustering algorithm.

import sys  
  
*# If number of parameters is incorrect, terminate.*   
**if** (len(sys**.**argv) **!=** 3):   
 **print**("USAGE: kmeans\_clustering.py [file] [clusters]")   
 sys**.**exit()   
num\_clusters **=** int(sys**.**argv[2])  
  
**with** open(sys**.**argv[1], 'r') **as** feature\_file:   
 *# Do stuff with the file*

Here, sys.argv represents the list of command line arguments, with the name of the code as the first element. If I’ve put in the correct parameters, then the program should proceed smoothly, with sys.argv[1] and sys.argv[2] seamlessly incorporated.

In addition to speed, another advantage of command line arguments is that they can be used as part of a process to automate the same script over and over again. Suppose I wanted to run my kmeans\_clustering script over and over again with the cluster value ranging from 2 to 100. One way is to tediously call kmeans\_clustering on the command line with ‘2’ as the last parameter, then do the same with ‘3’ after the first run has finished, and then do ‘4’ and so on. In other words, I’d have to call the program 99 times!

A better way is to make another Python script and use [os.system](http://docs.python.org/3.3/library/os.html#os.system) to call kmeans\_clustering as many times as I want. And this is as easy as changing the input to os.system. It takes in a string, so I would set a for loop that would create its unique string which would then act as input to the command line. See the following code for an example, where file1.arff is the made-up file that I’m using as an example.

import os  
  
**for** i **in** range(2,101):   
 input\_string **=** "python kmeans\_clustering file1.arff " **+** str(i)   
 os**.**system(input\_string)

So now this program will call kmeans\_clustering 99 times automatically, each time with a different parameter for the number of clusters. Quite useful, isn’t it? This is one of the biggest benefits of *using a program to call another program*. Just be wary that if you make a change to a program *while* another script is calling it, then those changes *will* be reflected the next time the program gets called.

***4. List Comprehension***

In my opinion, *list comprehension*, or the process of forming lists out of other lists or structures, is something that exemplifies the beauty and simplicity of Python programming. Remember the code I wrote earlier which set up a list that contained all the even integers in [2,16)[2,16)? I could have just written the following one-liner:

**>>>** list1 **=** [i **for** i **in** range(2,16,2)]   
**>>>** list1   
[2, 4, 6, 8, 10, 12, 14]

To understand the syntax, it’s helpful to refer to the (old) Python 2.7.5 documentation, which has a [nice explanation](http://docs.python.org/2/reference/expressions.html#list-displays) (emphasis mine):

*A list display yields a new list object. Its contents are specified by providing either a list of expressions or a list comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order.****When a list comprehension is supplied, it consists of a single expression followed by at least one for clause and zero or more for or if clauses. In this case, the elements of the new list are those that would be produced by considering each of the if or if clauses a block, nesting from left to right, and evaluating the expression to produce a list element each time the innermost block is reached.***

In other words, we’ll be given some expression that becomes an element of the list, and it will be subject to some restriction based on our series of for or if causes. Sometimes, if there are multiple loops and conditionals to evaluate, it can be more easily viewed if split into multiple chunks. I do this in the comments in the below code example. (If it’s absolutely necessary to introduce line breaks to better understand some list comprehension, the code might be a tad too complicated, but I believe it’s perfectly fine to use list comprehension in this example I provide.)

list2 **=** [(x, x**\*\***2, y) **for** x **in** range(5) **for** y **in** range(3) **if** x **!=** 2]  
  
'''  
list2 = [(0, 0, 0), (0, 0, 1), (0, 0, 2), (1, 1, 0), (1, 1, 1), (1, 1, 2), (3, 9, 0), (3, 9, 1), (3, 9, 2), (4, 16, 0), (4, 16, 1), (4, 16, 2)]  
  
This expression can be easily understood as:   
list2 = [(x, x\*\*2, y) for x in range(5):   
 for y in range(5):   
 if x != 2]   
'''

As the documentation clearly states, it’s also possible to create nested lists via list comprehension. This can be useful if one wants to initialize something like a *table* or a *matrix*. When I wrote my first Python program a year ago, I indeed used list comprehension to construct a table of elements that I would update as part of a [dynamic programming](http://en.wikipedia.org/wiki/Dynamic_programming) algorithm.

**>>>** list3 **=** [[0 **for** i **in** range(3)] **for** i **in** range(3)]   
**>>>** list3   
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

***5. Slicing***

Slicing is the process of taking a subset of some data. It is most commonly applied to strings and lists. My backstory for how I first learned about slicing was when I had to repeatedly iterate through a list and apply a function to all but its last element. I kept using an ugly loop that iterated through the indices of the list and performed a check each time to ensure that I wasn’t at that last element.

I eventually realized that this was one of the dumbest things I was doing, so I searched about how to obtain everything but the last element. And that was when I began my slicing journey. For this particular example, we can just use [:-1] to obtain everythingbut the element with index negative one, which will be the last element. (If one makes a list with NN elements, then the element located at index N−1N−1 *also* has an equivalent index of −1−1, and similarly for the indices N−2N−2 and −2−2, and so on.)

*# The bad way*   
**for** index **in** range(len(list4)):   
 **if** (index **!=** len(list4)**-**1):   
 *# Do something*  
  
*# The better way*   
**for** element **in** list4[:**-**1]:   
 *# Do something*

Fortunately, slicing isn’t limited to just getting rid of one element. Letting list1 be an arbitrary list, we can make list2 be a new list taking on a specified subset of list1s values by using the following general syntax:

list2 **=** list1[start:stop:step]

Here, start is the index of the first element we want, stop is the index of the first element we don’t want (remember that in Python, ending indices are often exclusive rather than inclusive), and steprepresents the number kk where we take each kk-th value. It can be negative, too, which would indicate that we’re moving backwards through the list. Not all of these values are needed; if the step is omitted, it defaults to +1. And as the example earlier should make clear, if either start or stop are omitted, they should default to 0 and the length of the list, respectively.

To gain a better intuition of slicing, it also helps to know how the indexing process works for negative numbers. In the official documentation, there’s a nice ASCII-style diagram (with the text “Python” in it) in the [Strings](http://docs.python.org/3/tutorial/introduction.html#strings) section that suggests you think about Python indices as pointing *between* elements of data.

It’s also important to understand the role that the colons play in splicing syntax. In the code above, I used [:-1] to refer to all but the last element in a list. If I had omitted that earlier colon, that would have resulted in just getting the last element of the list! If I had put the colon to the right of the -1, then I would still only obtain the last element, since that starts from the last-indexed element and gets all values beyond that (of which there are none). The following code shows how differences in colon placement and the number of parameters present affect splicing. An easy way to understand where colons should be placed is to just put in all three start, stop, and step values and delete the ones that are set at their default values (0, length of list, and +1, respectively). What’s left is how the colons can be formatted, though if ‘step’ is at its default value, we don’t need the colon preceding it at the end. For instance, list[2::] is the same as list[2:], and list[:4:] is the same as list[:4].

**>>>** list1 **=** [3,4,5,6,7,8]   
**>>>** list1[2:4]   
[5, 6]   
**>>>** list1[2:]   
[5, 6, 7, 8]   
**>>>** list1[:4]   
[3, 4, 5, 6]   
**>>>** list1[::2]   
[3, 5, 7]   
**>>>** list1[::**-**1]   
[8, 7, 6, 5, 4, 3]   
**>>>** list1[:5:2]   
[3, 5, 7]   
**>>>** list1[:4:2]   
[3, 5]

(Yes, it’s interesting that [::-1] reverses a list.)

I advise anyone to play around with semi-complicated slicing before using it in code. This is where the Python shell becomes extremely useful. (See #2 on this post.)

***6. Dictionaries and Sets***

Lists are by far the most common data structure I use when Python programming, but I still make extensive use of dictionaries, sets, and other data structures, since they have their own advantages.

A set is simply a container that holds items, like a list, but only holds distinct elements. That is, if you add in element X to a set already containing X, the set doesn’t change. This can be an advantage of sets over lists, since I often need to ignore duplicates when I’m managing lists, and making a set based on a pre-existing list is as easy as typing in set(list\_name).

But possibly an even bigger advantage with sets is their *super fast* lookup. Testing if an element is in a list takes O(n)O(n) time. With sets, however, membership testing is constant, O(1)O(1)-time. Of course, this requires set elements to be [hashable](http://docs.python.org/2/glossary.html), which means that items need to be associated with some constant number (i.e. “hash value”) so that they can be looked up in a table quickly.

**>>>** example1 **=** [i **for** i **in** range(5)]   
**>>>** example2 **=** [i **for** i **in** range(3,8)]   
**>>>** example3 **=** example1 **+** example2   
**>>>** example1   
[0, 1, 2, 3, 4]   
**>>>** example2   
[3, 4, 5, 6, 7]   
**>>>** example3   
[0, 1, 2, 3, 4, 3, 4, 5, 6, 7]   
**>>>** set\_example1 **=** set(example3)   
**>>>** set\_example1   
{0, 1, 2, 3, 4, 5, 6, 7}

Of course the downside with sets over lists is that they don’t support indexing of elements, so there’s no ordering. This is a pretty big drawback, but regardless, if you don’t care about order and duplicates, and want speedy membership testing, sets are the way to go.

In addition to sets, I find dictionaries to be an incredibly useful data structure. A dictionary is something that associates to each *key* a *value*, so it’s essentially a function that pairs up elements together.

**>>>** dict\_example **=** {'Bob' : 21, 'Chris' : 33, 'Dave' : 40}   
**>>>** dict\_example   
{'Bob': 21, 'Dave': 40, 'Chris': 33}   
**>>>** dict\_example['Adam'] **=** 11   
**>>>** dict\_example   
{'Adam': 11, 'Bob': 21, 'Dave': 40, 'Chris': 33}   
**>>>** dict\_example['Bob']   
21

There are many scenarios where dictionaries are useful. As an added benefit, searching the values by key is efficiently done in constant time, just like in sets. Due to their widespread use, dictionaries are one of the most heavily optimized data structures in basic Python.

***7. Copying Structures (and Basic Memory Management)***

Since it’s so easy to make a list in Python, one might think copying it is also straightforward. When I was first starting out with the language, I would often try to make separate copies of lists using simple assignment operators.

**>>>** list1 **=** [1,2,3,4,5]   
**>>>** list2 **=** list1   
**>>>** list2**.**append(6)   
**>>>** list2   
[1, 2, 3, 4, 5, 6]   
**>>>** list1   
[1, 2, 3, 4, 5, 6]

Notice what happens? I make list1 and try to make list2 be a copy of that list via assignment. But if I modify list2, such as by adding in the number 6, it *also* modifies list1! This is a deceptive but important point. Making lists equal to other lists will essentially create two variable names pointing to the *same* list in memory. This will apply to any ‘container’ item, such as dictionaries.

Since simple assignment does not create distinct copies, Python has a built-in [list](http://docs.python.org/3/library/functions.html#func-list) statement, as well as generic [copy](http://docs.python.org/3/library/copy.html) operations. It’s also possible to perform copies using slicing. Some solutions are shown below.

**>>>** list3 **=** list(list1)   
**>>>** list1   
[1, 2, 3, 4, 5, 6]   
**>>>** list3   
[1, 2, 3, 4, 5, 6]   
**>>>** list3**.**remove(3)   
**>>>** list3   
[1, 2, 4, 5, 6]   
**>>>** list1   
[1, 2, 3, 4, 5, 6]   
**>>>** import copy   
**>>>** list4 **=** copy**.**copy(list1)

There’s also another thing to be worried about — what if you have containers *within* containers? While implementing a machine learning algorithm last semester, I ran into the problem of copying dictionaries that contained dictionaries. I thought I was okay using the straightforward [dict](http://docs.python.org/3/library/stdtypes.html#dict) operation, but I realized that if I changed a dictionary within one of those dictionaries, that change would be reflected in both of the larger dictionaries!

An example of this error is shown below, where I modify dict2s first dictionary by adding in the 'z1' -> 60 mapping. That key-value pair will also be reflected in dict1s first dictionary.

**>>>** dict1 **=** {'a': {'x1' : 20, 'y1' : 40}, 'b': {'x2' : 15, 'y2' : 30}}   
**>>>** dict2 **=** dict(dict1)   
**>>>** dict2   
{'a': {'x1': 20, 'y1': 40}, 'b': {'x2' : 15, 'y2': 30}}   
**>>>** dict2['a']['z1'] **=** 60   
**>>>**   
**>>>** dict2   
{'a': {'x1': 20, 'y1': 40, 'z1': 60}, 'b': {'x2': 15, 'y2': 30}}   
**>>>** dict1   
{'a': {'x1': 20, 'y1': 40, 'z1': 60}, 'b': {'x2': 15, 'y2': 30}}

The solution to this is to use the [deepcopy](http://docs.python.org/3/library/copy.html#copy.deepcopy) method, which will copy *everything*. This is the most memory-intensive operation of the copying solutions I’ve discussed here, so use it only if the other methods won’t work.

***8. Generators***

Remember all the stuff about lists I’ve talked about? If you’ve been thinking about memory management, you might wonder why we need to store gigantic lists in memory if we might not even access their values. This is where *generators* come into play.

Generators in Python provide us with the advantageous concept of [lazy evaluation](http://en.wikipedia.org/wiki/Lazy_evaluation), so when we “construct” generators, we don’t actually evaluate some value within it unless it’s absolutely needed. One of the biggest benefits of lazy evaluation is in memory footprint. If we construct a generator that consists of the numbers 1 through N for large N, and our code ends up only needing the first three numbers, then Python doesn’t need to construct and store the remaining numbers in memory as it would for a list. (This hypothetical scenario could happen because there might be a function with different N values for each call that uses a generator.) If you’re curious, the Wikipedia page has additional information about lazy evaluation.

The first time I was introduced to generators was when I read a Stack Overflow post that said Python 2.x programmers should always use xrange() over range(), because xrange() is a generator, or at least, has the generator-like quality of lazy evaluation. For the record, even though I almost always use xrange() over range() in Python 2.7 code, range() does have its uses, such as if one needs an actual list.

Basically, range() and xrange() perform the same task, but the difference is that range(n) will literally construct a list consisting of the numbers 0 through n-1, while xrange(n) only provides us with those numbers when we need them.

As a testament to the usefulness of generators, Python 3 changed range() so that it now possesses xrange() qualities. The xrange() function has been omitted, as the Python 3 shell indicates:

**>>>** [i **for** i **in** range(10)]   
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   
**>>>** [i **for** i **in** xrange(10)]   
Traceback (most recent call last):   
File "", line 1, **in**   
NameError: name xrange **is** **not** defined

There are other sites that can explain generators better than I can, such as the Python Wiki page on[generators](http://wiki.python.org/moin/Generators). One thing to note is that if we do need a list out of a generator, then we can just call the list() method that I used earlier (in #7, on copying stuff). Generators also have their own version of list comprehension, called *generator comprehension*.

So why do we even use lists anyway? I can illustrate some reasons in the following code.

**>>>** list1 **=** ['adam', 'bob', 'chris', 'dave']  
**>>>** gen1 **=** (x **for** x **in** list1)   
**>>>** gen1   
at 0x1006a68c0**>**   
**>>>** **for** i **in** gen1:   
**...** **print**(i)   
**...**  
adam   
bob   
chris   
dave   
**>>>** **for** i **in** gen1:   
**...** **print**(i)   
**...**  
**>>>**

I made a generator, gen1, out of an list containing four common English names, but if I try to print the generator, I instead get a “generator object” expression. *Values in generators don’t “exist” until they’re needed on demand.*

A second tricky point about generators is that if we iterate through it, we can’t iterate it again like we would during the first time. The second loop fails to print anything.

So generators do have their place in Python, but so do lists and other non-generators. To understand generators better, it might also be useful to incorporate the *yield* keyword in Python. This (fantastic) Stack Overflow [question and answer](http://stackoverflow.com/questions/231767/the-python-yield-keyword-explained) explains it far better than I can, and I learned a lot about generators just by reading that page.

***9. File Management***

With many Python scripts using files as input, such as my kmeans\_clustering code I posted earlier, it’s important to know the correct ways to incorporate files in one’s code. The official documentation explains that the [open](http://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files) keyword is used for this purpose. It is pretty straightforward, and we can loop through the file to analyze it line by line. Alternatively, we can use the readlines() method to create a list consisting of each line in the file, but just be wary if the file is large.

f **=** open('test.txt', 'r')   
**for** line **in** f:   
 *# Read each line*   
f**.**close()

The f.close() is is important, since it’s done to free up memory. From the official documentation:

*When you’re done with a file, call f.close() to close it and free up any system resources taken up by the open file. After calling f.close(), attempts to use the file object will automatically fail.*

I almost never use f.close() though, because I can use the with keyword, which will automatically close the file for me once the code exits its block. In fact, I posted an example of this earlier when I talked about my kmeans\_clustering code. Here’s the relevant parts of it reproduced below.

*# Note: sys.argv[1] is the file, and 'r' means I'm reading it*   
**with** open(sys**.**argv[1], 'r') **as** feature\_file:   
 all\_lines **=** feature\_file**.**readlines()   
 **for** i **in** xrange(len(all\_lines)):   
 *# Do stuff here*

Now, in most cases, I believe that you don’t absolutely *need* to take advantage of f.close(), since if a script that was reading in a basic text file (but doesn’t close it) finishes running, then that text file should automatically be closed anyway. I can imagine that things can get more complicated with multiple files and scripts running together, so I’d get in the habit of using with to read in files.

If you’re interested in writing files, you can change the r parameter to w (or r+ which will enable both reading and writing) and use the file.write() method. This is common in research settings, where you might have to modify text files in accordance to some experiment.

***10. Classes and Functions***

It’s pretty easy to define a function in Python, using def, such as the following trivial example, which counts the number of zeros in the input, which will be a string.

**def** **count\_zeros**(string):   
 total **=** 0   
 **for** c **in** string:   
 **if** c **==** 0:   
 total **+=** 1   
 **return** total  
  
**print**(count\_zeros('00102')) *# Will return 3*

Recursive functions are also straightforward, and behave as in most major object-oriented programming languages.

Compared to Java, I haven’t used too many classes in my Python Programs, so my expertise in this realm is quite limited. Still, classes are an important part of object-oriented languages, and Python is (contrary to some people’s opinions) object-oriented, so it’s worth it to read the [Classes](http://docs.python.org/3/tutorial/classes.html) documentation if you have the time. The documentation page I just linked to, though, states affirmatively that:

*Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.*