# Programming Bootcamp 2015: Lab 6

Sarah Middleton

Last Updated: June 18, 2015

Name: ANSWER KEY

If you would like to receive points for your work, copy and paste your code into this document, save it, and send it to **sarahmid@mail.med.upenn.edu** with "Bootcamp Lab6" as the subject line **before 9am on 5/23**.

## 1 Guess the output: scope practice (2pt)

```
1  def fancy_calc(a, b, c):
2          x1 = basic_calc(a,b)
3          x2 = basic_calc(b,c)
4          x3 = basic_calc(c,a)
5          z = x1 * x2 * x3
6          return z
7
8  def basic_calc(x, y):
9          result = x + y
10         return result
11
12 x = 1
13 y = 2
14 z = 3
15 result = fancy_calc(x, y, z)
```

(a) **(1 pt)** Number each line of code above based on the order of execution. If a line is executed more than once, number it multiple times. (Alternatively, enter the line numbers in order below.)

12
13
14
15
1
2

(b) **(1 pt)** Based on the above code, guess the output of the following:

| Code | Predicted output | Actual output |
|---|---|---|
| `print x` | | 1 |
| `print z` | | 3 |
| `print x1` | | NameError |
| `print result` | | 60 |

## 2 Data structures as function parameters (1pt)

Use the following code:

```
1 def getMax(someList):
2         someList.sort()
3         x = someList[-1]
4         return x
5
6 scores = [9, 5, 7, 1, 8]
7 maxScore = getMax(scores)
```

| Code | Predicted output | Actual output |
|---|---|---|
| `print maxScore` | | 9 |
| `print someList` | | NameError |
| `print scores` | | [1, 5, 7, 8, 9] |

Why does `scores` get sorted? When you pass a data structure as a parameter to a function, it's not a copy of the data structure that gets passed (as what happens with regular variables). What gets passed is a direct reference to the data structure itself. The reason this is done is because data structures are typically expected to be fairly large, and copying/re-assigning

the whole thing can be both time- and memory-consuming. So doing things this way is more efficient. It can also surprise you, though, if you're not aware it's happening. If you would like to learn more about this, look up "Pass by reference vs pass by value".

Try the following code:

```
list1 = [1, 2, 3, 4]
list2 = list1
list2[0] = "HELLO"

print list2
print list1
```

Yes, that's right–even when you try to make a new copy of a list, it's actually just a reference to the *same* list. This is called aliasing. The same thing will happen with a dictionary.

So what if we want to make a separate copy? Here's a way for lists and dicts:

```
# for lists
list2 = list1[:]

# for dicts
dict2 = dict1.copy()
```

# 3 Starting your function file (8 pt)

Before starting this question, create a file called `my_utils.py`. You will store all the functions you create in this file. For these problems, you can use your previous code as a starting point. If you didn't finish those problems, feel free to use the code from the answer sheet, just make sure you understand how they work!

(a) **(1 pt)** Create a function called `gc` that takes a single sequence as a parameter and returns the GC content of the sequence (as a 2 decimal place float).

```
def gc(seq):
    gcCount = seq.count("C") + seq.count("G")
    gcFrac = float(gcCount) / len(seq)

    return round(gcFrac,2)
```

(b) **(1 pt)** Create a function called `reverse_compl` that takes a single sequence as a parameter and returns the reverse complement.

```
def reverse_compl(seq):
    complements = {'A':'T', 'C':'G', 'G':'C', 'T':'A'}
    compl = ""
    for char in seq:
        compl = complements[char] + compl
```

(c) **(1 pt)** Create a function called `read_fasta` that takes a file name as a parameter (which is assumed to be in fasta format), puts each fasta entry into a dictionary (using the header line as a key and the sequence as a value), and then returns the dictionary.

```
def read_fasta(fileName):
    ins = open(fileName, 'r')
    seqDict = {}
    activeID = ""

    for line in ins:
```

(d) **(2 pt)** Create a function called `rand_seq` that takes an integer length as a parameter, and then returns a random DNA sequence of that length. *Hint: Look for Python functions that will make this easier.*

```
def rand_seq(length):
    import random

    nts = ['A','C','G','T']
    seq = ""
    for i in range(length):
```

(e) **(3 pt)** Create a function called `shuffle_nt` that takes a single sequence as a parameter and returns a string that is a shuffled version of the sequence (i.e. the same nucleotides, but in a random order). *Hint: Look for Python functions that will make this easier.*

```
def shuffle_nt(seq):
    import random

    strList = list(seq)
    random.shuffle(strList)
    shuffSeq = "".join(strList)
```

Test each function using simple examples to ensure that they work. Note, you can test your functions in the interpreter–just use `import my_utils` as you would with any other module. One important note if you do this, though: your functions won't automatically update when you re-save your file, and re-typing `import my_utils` will do nothing. You must either restart the interpreter, or use the `imp` module to manually reload it:

```
1 import imp
2 imp.reload(my_utils)
```

# 4 Command line args practice (7 pt)

(a) **(2 pt)** Write a script that takes 3 integer arguments from the command line. Check that the correct number of parameters is supplied, and if not, print an error message and exit. Otherwise, go on to use those three integers to compute the quadratic formula. (Note: As with most things, all arguments are read in as strings. To use them as integers, you must convert them.)

```
import sys

if len(sys.argv) == 4:
    a = float(sys.argv[1])
    b = float(sys.argv[2])
    c = float(sys.argv[3])
```

(b) **(5 pt)** Write a script that takes two string arguments from the command line: (1) the name of an existing fasta file (to be read in), and (2) an output file name (which you will create and print to). Using the functions you created in problem 3, read in the fasta file into a dictionary. Then loop through the dictionary and compute the length and GC-content of each sequence. Print an output file in the following format:

```
1 SeqID   Len   GC
2 ...     ...   ...
```

That is, print the header shown above, followed by the corresponding info about each sequence on a separate line. The "columns" should be separated by tabs. It will probably be easiest to print as you loop through the dictionary. You can use `horrible.fasta` to test your code.

```
import sys
import my_utils

if len(sys.argv) == 3:
    inFile = sys.argv[1]
    outFile = sys.argv[2]
```

# 5 K-mer generation (10 pt)

Create a function called `get_kmers` that takes a single integer parameter, k, and returns a list of all possible k-mers of A/T/G/C. For example, if the supplied k was 2, you would generate all possible 2-mers, i.e. [AA, AT, AG, AC, TA, TT, TG, TC, GA, GT, GG, GC, CA, CT, CG, CC].

This function must be **generic**, in the sense that it can take *any* integer value of k and produce the corresponding set of k-mers.

To get you started, here's a skeleton of the function you can use (feel free to change it, though):

```
def get_kmers(k):
        kmers = []

        # your code here

        return kmers
```

Write a short bit of code that calls this function so that you can test it. As there are $4^k$ possible k-mers for a given k, stick to smaller values of k for testing.

I have not really taught you any particularly obvious way to solve this problem, so feel free to get creative in your solution! I will be grading this problem for correctness and will give partial credit for effort.

*Note: there are many ways to do this, and plenty of examples online. Since the purpose of this question is to practice problem solving, don't directly look up "k-mer generation"... try to figure it out yourself. You're free to look up more generic things, though.*

```
# Method 1
# Generic kmer generation for any k and any alphabet (default is DNA nt)
# Pretty fast
def get_kmers1(k, letters=['A','C','G','T']):
    kmers = []
    choices = len(letters)
    finalNum = choices ** k

    # initialize to blank strings
    for i in range(finalNum):
        kmers.append("")

    # imagining the kmers lined up vertically, generate one "column" at a time
    for i in range(k):
        consecReps = choices ** (k - (i + 1))   #number of times to consecutively repeat each letter
        patternReps = choices ** i #number of times to repeat pattern of letters

        # create the current column of letters
        index = 0
        for j in range(patternReps):
            for m in range(choices):
                for n in range(consecReps):
                    kmers[index] += letters[m]
                    index += 1
```