Python Workshop for Beginners

Steven Lakin

Table of Contents

The Zen of Python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Introduction

With the advent of big data and the need for automation of repetitive tasks, basic programming skills are becoming a necessary skill for working in many fields. However, much of the instructional material on programming is intended to build a very solid foundation in programming basics. For our purposes, we can skip the details of rudimentary programming and take a more hands-on approach to basic scripting, since this is much of what we as non-computer scientists will be doing. We will be using the Python language, since it is an intuitive and powerful programming language.

Python, named after the Monty Python skits, was built with the intention of being easy to use, quick to learn, and syntactically fast; this is sometimes referred to in the documentation as being "Pythonic," based on the ideals of the language. Because of this, Python is what is called a "high-level" language; much of the clunky syntax of other languages (Java, C, R) is removed in python. There are no semi-colons at ends of lines, no brackets around portions of code, or the need to pre-define variables before use. You will find that this makes Python a fast language to program in, since we don't have to worry as much about typing and checking syntax. Much of the baseline "work" of lower-level languages has been built into Python, which allows you to access Python's intuitive structures and do your work as easily and fast as possible.

In this workshop, we will be focusing on applying Python to biological problems. Much of this "patchwork" programming for solving small problems is well-suited to short segments of code called scripts. A script is simply a file of code that does something. Scripts can be combined to make programs, packages, modules, and generally "software," all of which are generally the same thing with slightly different semantics in different languages. We will mostly be scripting in this class, though we will work toward making packages and learning the basics of building more advanced code structures.

The workshop will be divided into two segments: the first will be a lecture on a programming topic, and the second will be applying those concepts to miniature problems on Rosalind, named after Rosalind Franklin, whose work in X-ray crystallography contributed significantly to the discovery of DNA structure. These problems are bioinformatics related, which is a field that combines biology, computer science, mathematics, and statistics to solve biological problems involving large data sets. You will need to make an account on Rosalind to track your progress.

Installing Python
There are two primary versions of Python: Python2.7 and Python 3.4. For the purposes of learning Python and having the most backward compatibility, I recommend that you install Python2.7. That being said, if you are using 3.4, we can work with that!

Windows and Mac
On Windows and Mac, you can download executable files for installation from https://www.python.org/downloads/

Simply follow the prompts for installing Python, and you may find it useful to check the box "Add Python to PATH" for reasons we will discuss later.

Linux
On Ubuntu and Debian flavors of Linux, you can obtain Python by apt-get:

`http://askubuntu.com/questions/101591/how-do-i-install-python-2-7-2-on-ubuntu`

However, it is recommended to build it from source and use a virtual environment.

Text editors and IDEs

At the very least, you will need to have a basic raw text editor (not Word). You can use Notepad on windows, the basic text editor on Mac, or you can download a simple editor like Editra.

Alternatively, if you want more functionality (and you will eventually), you can look into one of the Python IDE's. IDE stands for Interactive Development Environment, and it will allow you to write scripts, run the scripts in the terminal, and access help documentation all in the same environment. A good example of an IDE is RStudio for the R language.

There are many IDEs for Python, and no one is the "best," so you will have to find your preference. An IDE with a lot of functionality is PyCharm, but it is a larger program and can be confusing when you're starting out. For now, if you find these to be too confusing, then stick with downloading a program like Editra and working in that.

Session 1: Variables, Data Structures/Types, Operations, and I/O

Section 1.1 - Variables and Basic Operations

The Python terminal is an interface to Python's functionality. The terminal "prompt" is denoted by three right wedges:

```
>>>
```

When you see this prompt, the terminal is telling you that you can enter commands. When this prompt is not visible, then Python is "working" on a command you have entered previously. The terminal is what is known as an "interactive" terminal. You can use it like you might use a calculator for basic operations:

```
>>> 2+2
4
>>> 2*3
6
```

While this is useful to us, we are more interested in working with information stored in the Python environment. How do we store data? By assigning it to a variable:

```
>>> a = 2
>>> b = 3
>>> a*b
6
>>> c = a + b
>>> c
5
```

Let's try division:

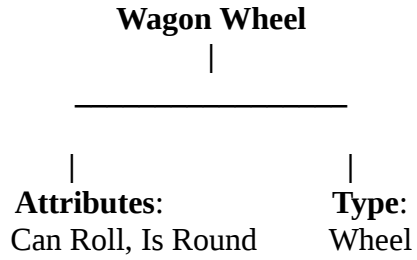
```
>>> a/b
0
```

Wait a tick, that's not right... This is because we are working with integers and to get fractions, we need to tell Python to give us a value of the right **type**:

```
>>> float(a)/b
0.6666666666666666
```

Section 1.2 - Objects: Types, Attributes, and Methods

Let's talk about types. It is best (and it will help us later) to think of data that we put into Python as **objects**. Objects, philosophically speaking, have certain attributes, and we can envision that there are objects of various types. Consider a wagon wheel: it has **attributes**, such as that it is round (by definition) or that it can roll, and it is of a certain **type**, which is that it is a specific kind of wheel.



Types describe a general class that your object falls into, and **attributes** describe what you can do to that specific object. A third term, **methods**, describe what that object can do for you. Let's use a concrete Python example that we have already encountered: the **Int** type, short for Integer.

The type() command will tell us what type our object is:

```
>>> type(c)
<type 'int'>
```

The dir() command will tell us what is in our object, which are it's **attributes** and **methods**:

```
>>> dir(c)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__', '__div__',
'__divmod__', '__doc__', '__float__', '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
'__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mod__',
'__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

That's a lot of attributes and methods! What do they mean? They tell us what kinds of operations we can or can't do to the object. The methods also tell us what the object is capable of doing, such as telling us how big it is in computer-language-size (bits) with the method in bold:

```
>>> c.bit_length()
3
```

There are many types, but the common ones that we will be working with are **string**, **int**, **float**, and the

data structure types **list** and **dictionary**. These are common because they are **mutable**, which means we can manipulate them in different ways. Let's see a few examples of these basic types and how we can work with them:

Section 1.3 – Strings as a Data Structure:

Strings are words made up of letters. Strings are defined with quotes, either single or double quotes:

```
>>> dna = 'AGCT'
>>> type(dna)
<type 'str'>
>>> dna
'AGCT'
```

is the same as doing it with double quotes:

```
>>> dna = "AGCT"
>>> type(dna)
<type 'str'>
>>> dna
'AGCT'
```

But let's say we are interested in specific nucleotides in the DNA string. We can access each individual letter by its position in the string, starting with 0 and going to 3:

```
>>> dna[0]
'A'
>>> dna[1]
'G'
>>> dna[2]
'C'
>>> dna[3]
'T'
```

Or certain **slices** of the DNA string using colons:

```
>>> dna[0:2]
'AG'
>>> dna[2:4]
'CT'
```

Python uses **zero indexing**, which means the indices start at zero. Visualize it like this for our DNA string:

$$_0A_1G_2C_3T_4$$

So when we slice from 0 to 2, we get out 'AG'. When we slice from 2 to 4, we get out 'CT'. If we want

a single element, we use the index on the left side of it, so the index of A is 0. We're not limited to consecutive slices either, let's say we want every other element:

```
>>> dna[::2]
'AC'
```

So the notation looks like this in general:

StringObject[start:stop:by]

Where “by” is equivalent to “every x letters”. We can even reverse the direction of the string:

```
>>> dna[::-1]
'TCGA'
```

We can also use negative indices to refer to the end of the string, where 0 is the very beginning or end of the string, -1 is the last letter in the string, -2 is the second to last letter, and so on:

```
>>> dna[-1]
'T'
>>> dna[-2]
'C'
```

Note that slicing with negative indices doesn't work as well. We'll talk about alternative strategies for slicing the end of strings later.

Let's say we have new information about our DNA and want to add it on. Remember that strings are mutable, so we can do this with simple addition, or **concatenation**:

```
>>> moreDna = "TT"
>>> longDna = dna + moreDna
>>> longDna
'AGCTTT'
```

If we want to remove specific elements of our DNA string, we can use the replace() **method**:

```
>>> longDna.replace("GC","")
'ATTT'
```

Note that we replaced GC with nothing (empty quotes), which is the equivalent of removing it. Methods, in general, are used like this:

Object.method(arguments)

Where the arguments are comma separated and depend on which method you're using. Methods are defined by the type of object you're working with, so for instance, you could use the replace() method on any string, because the type of object we are currently discussing is the string type. If you ever have a question about what methods there are or what arguments are valid, Google has all the answers. It is often faster to Google it than to try to look it up in the Python documentation.

Let's say we have an int (integer) object and want to turn it into a string object. This is called **coercing** one type into another. Not all types can be coerced into other types, but string is a broad category, so many other types can be coerced into strings:

```
>>> c
5
>>> type(c) >>> counts[0]
1
>>> counts[1]
2
>>> counts[2]
3
>>> counts[0:3]
[1, 2, 3]
>>> counts[-1]
5
>>>
<type 'int'>
>>> c_string = str(c)
>>> c_string
'5'
>>> type(c_string)
<type 'str'>
```

Now our integer is a string. This will be useful for some applications later on. This is about all I want to discuss about strings at this point, but I will leave here some useful operations with strings for your reference.

Useful String Manipulations:

```
>>> exampleString = "AGCTTTTCA"
```

Length of a string:

```
>>> len(exampleString)
9
```

Count of a non-overlapping pattern in a string:

```
>>> exampleString.count('A')
2
```

Find the first occurrence of a pattern in a string:

```
>>> exampleString.find('T')
3
```

Split a string at a defined pattern, remove the pattern, return a list:

```
>>> exampleString.split("GCT")
['A', 'TTTCA']
```

Join multiple strings with a separator (note that this works on lists of strings):

```
>>> splitString = exampleString.split("GCT")
>>> splitString
['A', 'TTTCA']
>>> "_".join(splitString)
'A_TTTCA'
```

Remove leading or trailing white space (including line endings):

```
>>> whiteSpace = "I am a sentence with a line ending\n"
>>> whiteSpace
'I am a sentence with a line ending\n'
>>> whiteSpace.strip()
'I am a sentence with a line ending'
```

Section 1.4 – Lists as a Data Structure:

So strings are pretty nifty, but what if we want to store discrete elements in a structure? Lists are both famous and notorious for this. They are great for storing small data, but they can be inefficient computationally if you're changing them frequently with large data sets (by large, I mean anywhere above half a million manipulations or so will be noticeable).

For those interested, here is a quick comparison for generating lists versus arrays (which we will discuss in a later session). Lists are quite slower:

```
Arrays:
$ python -m timeit "x=(1,2,3,4,5,6,7,8)"
10000000 loops, best of 3: 0.0388 usec per loop

Lists:
$ python -m timeit "x=[1,2,3,4,5,6,7,8]"
1000000 loops, best of 3: 0.363 usec per loop
```

However, for our introduction to Python, lists will be a key piece of our code because they are easy to use. So let's take a closer look at them.

Lists are defined by square brackets, and its elements can be other objects, such as strings, ints, other lists, etc. Here, we will use integers:

```
>>> counts = [1,2,3,4,5]
>>> counts
[1, 2, 3, 4, 5]
```

We can access lists the same way we can strings:

```
>>> counts[0]
1
>>> counts[1]
2
>>> counts[2]
3
>>> counts[0:3]
[1, 2, 3]
>>> counts[-1]
5
```

However, if we want to add elements to a list, we need to use the append() method:

```
>>> counts
[1, 2, 3, 4, 5]
>>> counts.append(6)
>>> counts
[1, 2, 3, 4, 5, 6]
```

We can get the length of the list (or of any dimensional object) with len():

```
>>> len(counts)
6
```

Most of what was mentioned in the string section applies here as well, so I will just give examples of list operations here that might be useful for reference.

Useful List Manipulations:

Insert a value into a list at a given position (first argument is the index, second is the value):

```
>>> counts
[1, 2, 3, 4, 5, 6]
>>> counts.insert(7,2)
>>> counts
[1, 2, 3, 4, 5, 6, 2]
>>> counts.insert(3,10)
>>> counts
[1, 2, 3, 10, 4, 5, 6, 2]
```

Remove the first occurrence of an element from a list:

```
>>> counts
[1, 2, 3, 10, 4, 5, 6, 2]
>>> counts.remove(10)
>>> counts
[1, 2, 3, 4, 5, 6, 2]
```

Reverse a list:

```
>>> counts
[1, 2, 3, 4, 5, 6, 2]
>>> counts.reverse()
>>> counts
[2, 6, 5, 4, 3, 2, 1]
```

Sort a list:

```
>>> counts.sort()
>>> counts
[1, 2, 2, 3, 4, 5, 6]
```

Count the occurrence of elements in a list:

```
>>> counts.count(2)
2
>>> counts.count(4)
1
```

Section 1.5 – Dictionaries as a Data Structure:

Dictionaries are a mapping of one value to another, such that they are linked. We refer to the index as a **key**, that has an associated **value**. Keys are unique within the dictionary; you cannot have repeated keys. However, you can have repeated values. This makes dictionaries great for storing associations for things like counting, translating, and storing associations in data sets. Let's take a look at how they work:

Dictionaries are defined by braces (curly brackets) where the key is mapped to its value with a colon:

```
>>> rnaTranslate = {"UUU":"F", "UUC":"F", "UUA":"L"}
>>> rnaTranslate
{'UUU': 'F', 'UUA': 'L', 'UUC': 'F'}
```

Here, we have made a dictionary that translates some of the RNA codons into their respective amino acids. The **keys** are the codons of RNA nucleotides, and the **values** are the single amino acid letter. Notice that we have a degenerate genetic code, which means we have more than one codon that maps to the same amino acid. Therefore, we must use the codons as the **keys**, because **keys must be unique**. We couldn't reverse the keys and values in this dictionary because the letter F repeats. **Values do not have to be unique**. So the amino acids are acceptable as the **values**.

Dictionaries have useful features beyond what a list or string can provide. For instance, we can get a list of the keys like so:

```
>>> rnaTranslate.keys()
['UUU', 'UUA', 'UUC']
```

And the values:

```
>>> rnaTranslate.values()
['F', 'L', 'F']
```

You can get the values out by using the keys as indices:

```
>>> rnaTranslate['UUU']
'F'
```

Now let's say I have a list of RNA codons that need to be translated:

```
>>> codons = ['UUU', 'UUU', 'UUU', 'UUA', 'UUU', 'UUC']
```

Let's translate them into amino acids:

```
>>> [rnaTranslate[x] for x in codons]
```

```
['F', 'F', 'F', 'L', 'F', 'F']
```

Pretty cool right? Try it on your terminal. This is called **list comprehension** and is part of what makes Python intuitive as a language. We will get into loops and flow in a later session, but this is one very Pythonic way to avoid using loops, which we will see can get confusing. I don't expect everyone to understand the above syntax yet, but I thought I'd demonstrate how useful associative structures like dictionaries can be. (Plus it's one of our Rosalind problems in a later session).

Here are some other fun things you can do with dictionaries.

Useful Dictionary Features

```
>>> exampleDict = {'A': 20, 'G': 120, 'C': 8, 'T': 11}
```

Sort a dictionary by its keys:

```
>>> sorted(exampleDict)
['A', 'C', 'G', 'T']
```

Get the key-value pairs out in a nested list:

```
>>> [[key,value] for key,value in exampleDict.items()]
[['A', 20], ['C', 8], ['T', 11], ['G', 120]]
```

Put them back into the dictionary:

```
>>> listData = [[key,value] for key,value in exampleDict.items()]
>>> listData
[['A', 20], ['C', 8], ['T', 11], ['G', 120]]
>>> { key:value for key,value in listData }
{'A': 20, 'C': 8, 'T': 11, 'G': 120}
```

Delete an entry in a dictionary:

```
>>> del exampleDict['A']
>>> exampleDict
{'C': 8, 'T': 11, 'G': 120}
```

Section 1.6 – Reading from Files (Input/Output, AKA I/O)

Much of what we will be doing will be working with data stored in files. Python can read data from files and store it in the environment so that you can manipulate it and then possibly write it to a new file. You can do this for many thousands of files with many thousands of line of data, which is part of what makes programming so useful. Here, I am going to introduce reading files only, since we won't need writing for a few sessions.

In Python, we explicitly open a file and assign that open file to a variable. I am going to use an example of a file on my computer, but you would substitute the location of my file with the location of yours. You can get the location of your file by opening your file browser window and clicking in the bar where it shows what folder you are in. That is called the **filepath**.

```
>>> openFile = open(r'/home/lakinsm/Documents/HelloWorld.txt', 'r')
>>> openFile
<open file '/home/lakinsm/Documents/HelloWorld.txt', mode 'r' at 0x7efd8cbc34b0>
```

You'll notice there is a lowercase “r” (in boldface) at the beginning of the filepath string. This is a special kind of string, called **raw string** or **string literal**. I'm not going to go into what this means for now, but I put it in there for the benefit of Windows people, because you need to put the “r” there for your filepaths to work. Windows filepaths have backslashes instead of forward slashes, which makes them a little harder to work with in Python. We can solve this problem by using the raw string. So if you are on Windows, your filepath might look liks this:

```
>>>openFile = open(r'C:\Documents\HelloWorld.txt', 'r')
```

So what do we have now? The variable openFile is telling Python where our file is on the computer and what we want to do with it. Mode “r” stands for **read**. We could have use “rw” instead of “r” and it would mean **read and write**. Though this is fine for now, we try to explicitly state what we want to do to the file so no accidents happen where we write over the file's contents by mistake.

So now how do we get the data in? We use the read() method:

```
>>> data = openFile.read()
>>> data
'Hello World!\n\n'
```

Great! But what are these \n characters? These are **newline** characters. If you're working in Windows, you might have \r\n instead. These are what tell your text editor programs to begin a new line. However, in Python, we don't care about them per se, since we want to work with the data in a more useful form. So, we can easily get rid of them with the method we learned in the string section:

```
>>> cleanData = data.strip()
>>> cleanData
```

```
'Hello World!'
```

And there you have it: you've just performed your first input and data cleaning operation in Python.

However, be careful with opening files like this. Unless you explicitly close the file, it will remain open. We don't want that, because it consumes your computer's RAM when it doesn't need to. So let's close the file:

```
>>> openFile.close()
```

Now we have the data in Python and the file is closed. You can now manipulate the data to your heart's desire and not worry about the file being open. Yet, sometimes we can forget to close files, so from this moment on, I'm going to use a more Pythonic way to open files, get the data, and close them:

```
>>> with open(r'/home/lakinsm/Documents/HelloWorld.txt', 'r') as openFile:
        cleanData = openFile.read().strip()
        Do Something With The Data
```

Note the indentation here; in Python, this is vital to get correct. I will discuss why this is in the next session, but for now remember that indentation is Python's way of knowing what to do in what order. Let's consider what we've done though. We have opened the file as openFile, imported the data into cleanData, and we (hypothetically) did something with it. The “with” statement make it so that when we are done doing whatever it is we want to do, the file will automatically be closed. This is a much cleaner and more Pythonic way of handling file reading.

Section 1.7 – Rosalind Problem 1: Counting DNA Nucleotides

Location: http://rosalind.info/problems/dna/

Problem:
A string is simply an ordered collection of symbols selected from some alphabet and formed into a word; the length of a string is the number of symbols that it contains.
An example of a length 21 DNA string (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset:
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTG
ATAGCAGC

Sample Output:
20 12 17 21

This Rosalind problem is about counting occurrences of a pattern in a string. Use the methods we discussed in the string section to output the variables, particular the string.count() method.

You can get the values out of the variables all at once with print():

```
>>> A = 20
>>> C = 12
>>> G = 17
>>> T = 21
>>> print(A,C,G,T)
(20, 12, 17, 21)
```

Session 2: Iteration, Comprehension, Logicals and Functions

Iteration is the workhorse of programming. When we repeat the same actions or calculations many times, we call it iterating. For basic scripting in Python, we will need two components for iteration: an **iterable** object to produce an **iterator**, and a temporary **variable** that refers to each element in the iteration.

In this section, we going to go over iteration at a high level, which is all that you'll need to know to use it effectively. Behind the scenes, there are interesting things going on that are at the heart of Object Oriented Programming, so we'll return to iteration again when we get to OOP.

Section 2.1 – Iteration with For Loops

When we apply iteration to an object, we call it "**iterating over**" that object. For instance, consider the following list:

```
>>> iterList = [1,2,3,4,5]
>>> iterList
[1, 2, 3, 4, 5]
```

Let's say we want to print each element of the list. We could do this manually, as we did in the previous section, but that is tedious and time consuming. Instead, let's tell the computer to do it for us by using a **for loop**[1]**:**

```
iterList = [1, 2, 3, 4, 5]
for number in iterList:
    print(number)
```

```
1
2
3
4
5
```

Here, the for statement tells Python that we are beginning iteration. For statements always require a **variable** and an **iterable object**. In general form, we told Python this:

```
for variable in iterable:
        print(variable)
```

---

1 Note: At this point, I'm now working with scripts and not directly in the terminal. The color-formatted text will always be the actual code I am working with in my text editor. The blue text is how the Python terminal looks (usually the output). If you would like to copy/paste this code, then use the **color-formatted** text to retain the correct indentation.

Here are a few examples of how iteration works with different **types** of objects:

```
iterString = "ACGT"
for x in iterString:
    print(x)
```

```
A
C
G
T
```

```
for i in iterList:
    print(3+i)
```

```
4
5
6
7
8
```

Perhaps we are interested in a more applied example (and one that we will use often). Consider this basic FASTA file, stored on my computer as example.fasta:

```
>Rosalind_7823
CAATAGCCCTCAACCCTCCCATCGTCGCTGTGACAATCAGACTCCTGTATGGCATTGCAC
CGTAGCGTCTCTTCCGTTGATAAAAAAAAAATGTGTGTTCGCCTTTCATGTCCCTTGTAAG
TCGCTCATGTAGCACGCTTTAATTAGTCATTTGCGGAGCTCGTTCGACTACTGTTGGCTA
```

FASTA files consist of two things: a **header** denoted by a “>” as its first character, and a **sequence**, usually DNA, but it can be RNA, protein, etc. The sequence is defined as all of the characters that fall between two headers (or the end of the file), so FASTA files can have sequences that are one-line or many lines. This particular file format has a many-line sequence format. We will see later how to read in a FASTA file in *any* of these formats.

Let's read in this file line by line and print the lines in Python:

```
with open("/home/lakinsm/Documents/python-workshop/example.fasta", "r") as openFile:
    for fastaLine in openFile:
        print(fastaLine.strip())
```

```
>Rosalind_7823
CAATAGCCCTCAACCCTCCCATCGTCGCTGTGACAATCAGACTCCTGTATGGCATTGCAC
CGTAGCGTCTCTTCCGTTGATAAAAAAAAAATGTGTGTTCGCCTTTCATGTCCCTTGTAAG
TCGCTCATGTAGCACGCTTTAATTAGTCATTTGCGGAGCTCGTTCGACTACTGTTGGCTA
```

So what we have done here is: 1.) while the file is open as openFile, 2.) for every line in the file, 3.) remove the whitespace with strip() and print the line.

Python will assume that when you are iterating over a file object, that you are iteration over its lines. This is immensely useful for simple cases of scripting, since we don't even have to specify to read the file, the for loop construction does it for us. Most of the time, we are interested in the lines of a file as a data type. Sometimes we will be interested in its columns, and we will go over strategies for handling column-data later on.

Sometimes we will need to work with indices as the iterable object in for loops. Notice that in the previous examples, we were iterating over the elements of the iterable object. Suppose, however, that we want to iterate over their indices instead. In this case, we don't know ahead of time how many elements there are in the object, so we need to generate that information. Let's do this using the range() function.

The range() function takes three arguments in this form:

range(start=0, stop, by=1)

By default, the start is 0 and the by is 1, but you must specify a stop. So here are a few examples:

```
>>> range(6)
[0, 1, 2, 3, 4, 5]

>>> range(5, 10)
[5, 6, 7, 8, 9]

>>> range(0,10,2)
[0, 2, 4, 6, 8]
```

Most of the time we will simply be using the default case, because we will be chaining range() with len():

```
>>> holyHandGrenade = ['One', 'Two', 'Five!', 'Three sir!', 'Three!']
>>> len(holyHandGrenade)
5
>>> range(len(holyHandGrenade))
[0, 1, 2, 3, 4]
```

Now we have the indices and we can loop over the object by its indices:

```
for index in range(len(holyHandGrenade)):
    print(holyHandGrenade[index])
```

```
One
Two
Five!
Three sir!
Three!
```

You might ask why we would ever want to do this when the other form is so much simpler. This form is useful when we need to refer to elements relative to other elements based on their position in the object. For example, we need this form when working with **recurrence algorithms.**

Recurrence algorithms are simply algorithms where each step refers to a previous step. Think about a for loop where we need to refer to information in the previous loop, or if we need to generate a new object whose elements are combinations of some previous elements. These are all **recurrence** relations, and we can do this with our index format. Consider generating the famous **Fibonacci sequence**:

$$f(x_n) = x_{n-1} + x_{n-2}$$

```
start = [1, 1]
for n in range(2, 10):
    answer = start[n-1] + start[n-2]
    start.append(answer)
print(start)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

But Python has anticipated that our index looping might be needed, so they have built a function called enumerate() that will give us simultaneously the index and the element of an iterable object:

```
>>> for i,v in enumerate(holyHandGrenade): print i,v
...
0 One
1 Two
2 Five!
3 Three sir!
4 Three!
```

So now we have access to the index without having to call len(range()) on the object. That will sometimes save us time computationally.

Iteration is pretty cool. For loops are very common and quite useful, but there are more ways to do iteration, so let's move on to other examples of iteration statements.

Section 2.2 – Iteration with Comprehension

While loops are very common and we do need them for certain applications, we usually want to avoid using loops in Python when it is possible. This is because Python has built-in ways of doing certain loop-related applications that tend to be much faster computationally than using the loop. One of these applications is for generating lists, arrays, and dictionaries. We could loop over the list and assign values to it, but this would be much slower than using a built-in python feature called **list comprehension**. List comprehensions take the following form:

[x for x in something]

Where the "something" is an iterable object containing some values. So for example we could do the following:

```
>>> a = [x for x in range(10)]
>>> print(a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that this is very similar to the for loop syntax, only instead of the "doing something" being inside of the loop block, we are just pulling it outside and putting it first. So for instance,

```
x = []
for x in range(10):
        x.append(3+x)
```

is the exact same as:

```
[3+x for x in range(10)]
```

Note that we can **do anything** to the first x and it will work because it is a stored value. We **can't do things** to the second x, because that is just the variable assignment (what we want to call our variable). And remember that the final element must be an iterable object.

Dictionary comprehension is very similar, only we need to specify two variables now, one for the **key** and one for the **value**:

```
>>> b = {key:value for key,value in (('A',1), ('B',2), ('C',3))}
>>> b
{'A': 1, 'C': 3, 'B': 2}
```

Notice that with two variables, our iterable object must contain elements that also have two values per element, so this could be a nested list or a nested array.

What is the tangible benefit for using this comprehension construction? Well, let's take a look for the

example of a million iterations (which actually is a fairly small number of operations in bioinformatics):

```
python -m timeit "x=[x for x in range(10)]"
1000000 loops, best of 3: 0.495 usec per loop

python -m timeit $'x=[]\nfor i in range(10): x.append(i)'
1000000 loops, best of 3: 0.89 usec per loop
```

So the list comprehension is about *twice as fast* as the loop format. While this doesn't really matter for applications that take seconds to complete, if we had a program that took 20 hours to run and relied heavily on list generation, then we could saving hours of time by using comprehension in place of loops.

Section 2.3 – Iteration with While Statements

There are times when we don't know how many iterations we want to perform, but we do know what criteria needs to be met before we want to stop. These cases are perfect for **while statements**, because they will continue to iterate until a condition is satisfied or they are told to stop.

A while loop construction is very simple:

while **condition**:
    do something

Where the condition is a **logical statement**, such as:

```
x = 15
while x > 10:
    x = x - 1
    print(x)
```

```
14
13
12
11
10
```

At the beginning of every iteration, the statement is evaluated, and if it is true, then we continue, and if it is false, then we stop. This makes while loops useful for mathematical operations where we need a certain threshold to be met to stop. However, we can also use while loops in a less intuitive way:

Pseudocode:
while True:
    do something indefinitely
    if a condition is met:
        stop

We can, within the while loop, manually specify when we want to stop. We will cover these control statements and logicals later, but for now, just keep this construction in the back of your head, because it is common. We first need to get through the sections on logical statements and control of flow before we can revisit this construction.

Section 2.4 – Logical Statements

A commonly encountered problem in programming is determining whether or not some condition is true for an object. For example, perhaps we would like to know whether two variables are equal to one another, whether a line in a file begins with ">" for FASTA, or whether we have reached a threshold.

These problems can be solved using logical statements, which can be used to construct decision trees. "If this, t hen do that, otherwise, do this, but if this other thing, then do something else." The structure of logical statements is as follows:

Pseudocode:
if condition:
    do something
elif condition2:
    do something else
elif condition3:
    do something else
….
elif conditionN:
    do something else
else:
    do the last thing

Where **if** means if, **elif** means else if, and **else** means in all other cases

You can have as many or as few of these as you want, and you can use **if** by itself. However, in order to use **elif** or **else**, there needs to be an **if** present before them. The **conditions** also have a certain structure:

Meaning	**Code**
Is equal to	is, ==
Is not equal to	is not, !=
Is greater than	>
Is less than	<
Greater than or equal to	>=
Less than or equal to	<=
And	&, and
Or	|, or
Is empty/None	not <*variable*>
Starts with	<*string*>.startswith("*pattern*")

Each of these **conditionals** will **evaluate** to a True or a False, which the if/elif/else statements will then use to make their decision on what to do.

Let's take a look at a quick decision tree with a for loop:

```
x = [1, 5, 10, 15]
for number in x:
    if number < 10:
        print "%d is less than 10" % number
    elif number > 10:
        print "%d is greater than 10" % number
    elif number is 10:
        print "10 is 10"
    else:
        print("Error")
```

```
1 is less than 10
5 is less than 10
10 is 10
15 is greater than 10
```

We can see how these conditionals evaluate by trying them in the terminal:

```
>>> 15 > 10
True
>>> 10 > 15
False
>>> 10 is 10
True
```

Many different functions accept logical values as input, and in number form they are evaluated as binary 0 or 1:

```
>>> max([True, False])
True
>>> sum([True, True, False, True])
3
```

They are actually their own type (Boolean operator), however, so do not put them in quotes when using them:

```
>>> type(True)
<type 'bool'>
```

We can also use other conditionals to evaluate complex logical expressions:

```
>>> (10 > 15) or (15 > 10)
True
>>> (10 < 15) and (15 > 10)
True
```

Finally, the **not** conditional evaluates to True when there is a missing value, which is useful for things like detecting the end of a file:

```
>>> test = None
>>> not test
True
>>> not not test
False
```

Section 2.5 – Basics of Functions

Functions are objects that do something. Think of them as machines in an assembly line or businesses that take something in, do something to it, and then mail the finished product back to you. Functions give us a way to logically group our code for reuse. If we only wanted to do something one time, then we wouldn't choose to put it into a function. However, let's say that we want to do the same thing in multiple different places in our code structure. Then instead of rewriting the code twice or many times, we simply put it into a function and call the function when we need it.

Functions have four important components: the **name**, the **arguments**, the **body**, and the **return values**. Here is the general form of a function in Python:

```
def functionName(argument1=default, argument2=default, …, argumentN=default):
        body
        body
        body
        return(value1, value2, …, valueN)
```

So, we name the function something, we tell Python how many arguments it accepts and add an optional default if the user does not specify something. If we do not list a default, then the argument is required to use the function. We then do something to the arguments (when the user inputs things as arguments, they are assigned to the argument variable), and we return to the user one or more values.

Take the following as an example:

```
def printCounts(countList):
    for c in countList:
        print(c)
```

When we run that code, Python now has an object printCounts that can be called as any other function would:

```
>>> printCounts
<function printCounts at 0x7fd6bebfb578>
>>> myList = [1,2,3,4]
>>> printCounts(myList)
1
2
3
4
```

So that is an example of a function that didn't return anything, it just does something. But we can also have it manipulate objects and return values to us in any form:

```
def fibonacci(iterations):
    start = [1, 1]
    for n in range(2, iterations):
        answer = start[n-1] + start[n-2]
        start.append(answer)
    return start
```

```
>>> fibonacci(20)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

We can also assign its return values to a new variable:

```
>>> answer = fibonacci(20)
>>> answer
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

We will go over more examples of how variable assignment works with functions later on. For now, we have all the basic building blocks we need to start scripting. Before we move to more advanced concepts, let's apply what we've learned in this section to a Rosalind problem.

Section 2.6 – Rosalind Problem: Rabbits and Recurrence Relations

http://rosalind.info/problems/fib

A sequence is an ordered collection of objects (usually numbers), which are allowed to repeat. Sequences can be finite or infinite. Two examples are the finite sequence (π,−2√,0,π) and the infinite sequence of odd numbers (1,3,5,7,9,…). We use the notation an to represent the n-th term of a sequence.

A recurrence relation is a way of defining the terms of a sequence with respect to the values of previous terms. In the case of Fibonacci's rabbits from the introduction, any given month will contain the rabbits that were alive the previous month, plus any new offspring. A key observation is that the number of offspring in any month is equal to the number of rabbits that were alive two months prior.

As a result, if Fn represents the number of rabbit pairs alive after the n-th month, then we obtain the Fibonacci sequence having terms Fn that are defined by the recurrence relation Fn=Fn−1+Fn−2 (with F1=F2=1 to initiate the sequence). Although the sequence bears Fibonacci's name, it was known to Indian mathematicians over two millennia ago.

When finding the n-th term of a sequence defined by a recurrence relation, we can simply use the recurrence relation to generate terms for progressively larger values of n. This problem introduces us to the computational technique of dynamic programming, which successively builds up solutions by using the answers to smaller cases.

Given: Positive integers n≤40 and k≤5.

Return: The total number of rabbit pairs that will be present after n months if we begin with 1 pair and in each generation, every pair of reproduction-age rabbits produces a litter of k rabbit pairs (instead of only 1 pair).

Sample Dataset

```
5 3
```

Sample Output

```
19
```

We have already solved the fibonacci problem earlier in this section, but can you now apply it to a case where there is a reproductive rate? Use the same general form, but figure out where to put *k*, and how to program a function such that it accepts the two numbers in the sample dataset to produce the sample output.

Section 2.7 – Rosalind Problem: Counting Point Mutations

http://rosalind.info/problems/hamm/

Evolution as a Sequence of Mistakes

A mutation is simply a mistake that occurs during the creation or copying of a nucleic acid, in particular DNA. Because nucleic acids are vital to cellular functions, mutations tend to cause a ripple effect throughout the cell. Although mutations are technically mistakes, a very rare mutation may equip the cell with a beneficial attribute. In fact, the macro effects of evolution are attributable by the accumulated result of beneficial microscopic mutations over many generations.

The simplest and most common type of nucleic acid mutation is a point mutation, which replaces one base with another at a single nucleotide. In the case of DNA, a point mutation must change the complementary base accordingly; see Figure 1.

Two DNA strands taken from different organism or species genomes are homologous if they share a recent ancestor; thus, counting the number of bases at which homologous strands differ provides us with the minimum number of point mutations that could have occurred on the evolutionary path between the two strands.

We are interested in minimizing the number of (point) mutations separating two species because of the biological principle of parsimony, which demands that evolutionary histories should be as simply explained as possible.

Problem

Figure 2. The Hamming distance between these two strings is 7. Mismatched symbols are colored red.

Given two strings s and t of equal length, the Hamming distance between s and t, denoted *d*H(s,t), is the number of corresponding symbols that differ in s and t. See Figure 2.

Given: Two DNA strings s and t of equal length (not exceeding 1 kbp).

Return: The Hamming distance dH(s,t).

Sample Dataset

```
GAGCCTACTAACGGGAT
CATCGTAATGACGGCCT
```

Sample Output

```
7
```

An approach to this problem would be to store the DNA strings in two separate variables, then compare each nucleotide value at each index in the string. Since they are equal length, you can get the len() of one of the strings then pass it to range() to generate a list of indices.

Session 3 – Handling Errors and Control of Flow

In this session, we will learn how to control iteration by handling errors, breaking loops, and returning values. This is commonly referred to as "control of flow," since iteration can be thought of as a flowing current of water/electricity, and when certain events occur, we want to redirect that flow in various directions or stop it all together. Control of flow relies on **statements**, such as **break**, **continue**, **yield**, **return**, **assert**, and **raise**.

Section 3.1 – Control of Flow: the break, continue, and return statements

In the last section, we discussed methods for iteration, including the **for** and **while** loop constructions. There will be times during iteration that you wish to break out of iteration, skip a step, or stop all loops altogether. To do this, we can use statements designed for controlling flow.

Let's start with examining the **break** statement. This statement, when placed inside a for or while loop, will break out of that loop. Perhaps we want to stop a loop if we hit a certain value:

```
for i in range(100):
    if i is 10:
        break
    else:
        print i
```

```
0
1
2
3
4
5
6
7
8
9
```

In this example, we have iterated on a range from 0 to 100, but since we specified to break the loop when the value is 10, the loop stopped before printing the value 10. Likewise, we can construct an infinite loop with **while** and break it when we wish:

```
i = 0
while True:
    if i is 10:
        break
    print i
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

We can use this for matching cases as well; let's say we want to print every line in a list until we reach the end. When working with files, the final line will be a None type or empty string. The following list simulates the lines in a file, where the last entry in the list is an empty line:

```
pretendFile = ["Line one", "Line two", "Line three", ""]
i = 0
while True:
    if not pretendFile[i]:
        break
    else:
        print pretendFile[i]
        i += 1
```

```
Line one
Line two
Line three
```

Remember that the **not** conditional matches the None type or an empty value, such as the empty string in the list above. In the above code, we set the initial index at 0, read each index in the list until we reach the end (empty line) and then break. However, **break** only breaks out of the loop that it is currently in, which means that it only breaks out of the most nested of a nested loop. Let's consider a loop inside a loop with a break statement:

```
j = 0
for letter in ['a', 'b', 'c']:
    print letter
    for j in range(10):
        if j is 2:
            break
        else:
            print j
            j += 1
```

```
a
0
1
b
0
1
c
0
1
```

Notice that we begin the first loop, the letter is printed, then we enter the second loop and print values until we reach 2, in which case we break out of the inner loop and print the next letter, re-enter the inner loop, and so on. If we want to break out of all loops, we will need to use the return statement (assuming the loops fall within a function). We will cover the **return** statement in a moment.

First, let's consider the case where we want to skip a certain value in the loop. Let's say we want to print the values 0 through 9 but we want to skip values 5 through 7 (inclusive). For cases where we want to skip "doing something" to a certain value, we will use the **continue** statement:

```
for i in range(10):
    if 5 <= i <= 7:
        continue
    else:
        print i
```

```
0
1
2
3
4
8
9
```

What **continue** does is to go to the next iteration of the loop. When **continue** is called, the loop immediately proceeds to the next iteration, therefore skipping all commands that follow the continue

statement. As with the break statement, the continue statement only works on the loop where it is placed.

If the goal is to break out of all loops, we can use the **return** statement, as long as we are within a function. The **return** statement breaks all loops and returns a value to the user, as we saw in the last section. This value can be assigned to a new variable, or it will be displayed to the terminal. For instance, let's say we want to return the position of the first nucleotide that matches "G" in the following DNA code:

```
def findNucleotide():
    dna = 'ACCACACACCCACATTTACAG'
    index = 0
    for nucleotide in dna:
        if nucleotide is "G":
            return index
        else:
            index += 1
```

```
>>> findNucleotide()
20
```

We have already seen that this value can be assigned to a variable (as in the previous section) and then the variable can be subsequently worked with. However, we're going to spend some time with this variable assignment from **return**, since Python has very intuitive ways to parse multiple return values with variable assignment. Let's consider the Fibonacci function we developed previously:

```
def fibonacci(n, k):
    start = [1, 1]
    for times in range(2, n):
        start.append(start[times-1] + k*start[times-2])
    return start
```

```
>>> sequence = fibonacci(10, 3)
>>> print(sequence)
[1, 1, 4, 7, 19, 40, 97, 217, 508, 1159]
```

Here, we have assigned the whole list of Fibonacci numbers to a single variable, so therefore the whole list will now be stored in that single variable "sequence." However, what if we want to store different values of this return list to different variables? Let's say we are interested in the first element, the last element, and all elements in-between. We can assign these three portions of the returned list to three separate variables all in the same line.

```
>>> first, middle, last = sequence[0], sequence[1:len(sequence)-1], sequence[-1]
>>> print first, middle, last
1 [1, 4, 7, 19, 40, 97, 217, 508] 1159
```

In general, Python 3.x.x version has much better options for dynamically assigning variables, but we'll leave it at that for Python 2.x.x, since that is what we are primarily using for this workshop. For our purposes, it is enough to remember that as long as you have enough values to "unpack," you can assign them each to a variable:

```
>>> a, b, c, d = [1,2,3,4]
>>> print a, b, c, d
1 2 3 4
```

Section 3.2 – Generators and the Yield Statement

We have talked about **iterables** as a way for python to loop over each element in an object. Python **iterates** over the elements in **iterable** objects, such as lists:

```
>>> for x in range(5):
...     print(x)
...
0
1
2
3
4
```

This is true if we assign a new object to be an iterable (such as a list):

```
>>> my_iterable = range(5)
>>> my_iterable
[0, 1, 2, 3, 4]
>>> for x in my_iterable:
...     print x
...
0
1
2
3
4
```

With iterables, we will get the exact same answer *if we run the code again*:

```
>>> for x in my_iterable:
...     print x
...
0
1
2
3
4
```

This is because Python has stored the values contained within *my_iterable* in memory. That means the computer can access these values any time, and they will be there until you delete the variable or close your Python terminal. That also means that these values are consuming resources on your computer. For such trivial objects as *my_iterable*, this doesn't matter. But consider if we begin working with FASTA files that can be many gigabytes in size. Typical computers will have 8-16 GB these days, but

some of our FASTA files for deep sequencing can be as large as 10-20 GB in some specialized cases. In this case, our file won't fit into memory, and we will need to take another approach. This is where **generators** come in.

Generators are like iterables, except they are not stored in memory. When I create a generator as an object, python doesn't store any values from the generator; it only stores the code itself. It then uses this stored code as instructions for what to do during each cycle of the loop. Let's look at an example. First, we need to construct a generator.

Generators aren't explicitly created in Python. Python will automatically create a generator out of any loop or function that has a **yield** statement in it. What does the **yield** statement mean? Well, it's very similar to the **return** statement, except *it does not stop all loops*. Instead, it just remembers which loop it stopped on and continues for the next loop, **yielding** a value back to the user for each loop completed. Think of it like a bookmark. Your code will execute until it reaches the first **yield**, then it will return that value, but it will remember where it is. Next time you need the *next value*, it will continue from where it left off until it reaches another **yield**. Here are two examples. Let's create a very simple generator function:

```
def shes_a_witch():
    yield "What also floats in water?"
    yield "Bread!"
    yield "Apples!"
    yield "Uh, very small rocks!"
    yield "Cider!"
    yield "Great gravy."
    yield "Cherries.  Mud. Churches.  Lead -- A duck!"
    yield "Exactly."
```

So, we have a function that "holds" the values of a bunch of **yield** statements in its body. Let's see what happens when we use this as a **generator**. This **generator** is the "code template" that I mentioned earlier. First, we need to assign it to a value/object:

```
>>> burn_her = shes_a_witch()
>>> print burn_her
<generator object shes_a_witch at 0x7f80df566730>
>>> for line in burn_her:
...     print line
...
What also floats in water?
Bread!
Apples!
Uh, very small rocks!
Cider!
Great gravy
Cherries.  Mud. Churches.  Lead -- A duck!
Exactly.
```

We can see here that our “for” loop used the **generator** as a list of elements, looping over each element until it hit a **yield** statement, printing the value, then going back to the generator to pick up where it left off. If we use the “for x in object” construction, this will continue until there are no more values to get. But remember, with **iterables**, we can call the same function twice. What about with generators?

```
>>> for line in burn_her:
...     print line
...
```

Nothing happens. Why? Because Python remembers where it was in the generator, and our **generator** has been “**consumed**.” There are no more values in our generator, since they have all been used. Python knows that our generator is at the end of its values, so it doesn't do anything else with it.

These examples might illustrate this better:

```
>>> burn_her = shes_a_witch()  ## Regenerate our generator

>>> for x in range(3):
...     print burn_her.next()
...
What also floats in water?
Bread!
Apples!

>>> for x in range(3):
...     print burn_her.next()
...
Uh, very small rocks!
Cider!
Great gravy

>>> for x in range(3):
...     print burn_her.next()
...
Cherries.  Mud. Churches.  Lead -- A duck!
Exactly.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
StopIteration
```

So, what did we do exactly? We first regenerated the generator by recalling the **shes_a_witch** function. This reset our generator. Now we can get some of the values from the generator by using its internal **next()** method. This is a method present in all **iterables** and **generators**, so that they can do their job. In the “for x in object” construction, it automatically calls the next() method until it reaches the

StopIteration error, as we saw happened with our generator as well (the final lines). The key thing to note about **generators** is that they remember where they are in iteration. This is *not the case* with **iterables**. So why is this useful? Well, either when we want to keep track of where we are in an object, or if we want to return values without breaking out of all loops. Consider as our second example the following nested **while** loop generator:

```
def readFile(infile):
    while True:
        current = infile.next()
        if current is not "":
            break
    while True:
        while True:
            if current[0] is ">":
                break
            elif not current:
                return
            else:
                current = infile.next()
        yield current
        current = infile.next()
```

Now we need a "file" for it to read, so let's make an iterable out of a list and simulate a file with some important lines and some unimportant lines. Let's say the "important" lines are those that begin with ">".

```
>>> example_file = ["", "", "", "", ">What is your favorite color?", "You killed my father, prepare to die!", ">Blue!"]

>>> infile = iter(example_file)
>>> print infile
<listiterator object at 0x7f80df56e950>
```

So now we have an iterable "file" for our function to work on. Let's see what happens when we read it with our generator:

```
>>> for x in readFile(infile):
...     print x
...
>What is your favorite color?
>Blue!
```

So we print only those lines that are important and skip all others. As the end to this section, let's break down exactly what we programmed our generator to do, since it is related to the exercise at the end of

this session.

```
while True:
    current = infile.next()
    if current is not "":
        break
```

This first while loop assigns the first element of the infile to *current*, and tests to see if it is emtpy. If it is an empty line, then it doesn't do anything and the next loop begins with the next element of infile, but if it is *not* empty, we break out of this loop, since we have reached the first non-empty line, which may contain interesting information.

```
while True:
    while True:
        if current[0] is ">":
            break
        elif not current:
            return
        else:
            current = infile.next()
    yield current
    current = infile.next()
```

This second, or "outer" while loop is what contains our **yield** statement. Anything reaching the end of this loop will be yielded. However, it first must get past the "inner" while loop. The inner while loop is the workhorse of this function. It first tests to see if the line begins with ">", which we denoted as the marker for important information. If it doesn't begin with ">", then the loop tests to see if it is an empty value (in which case we should stop the function, since we could have reached the end of the file). Lastly, if it is neither important nor empty, we proceed to the next element.

Note that if the line *does* begin with ">", then we break out of our inner loop and the line is **yielded**, we then proceed to the next element and the inner loop starts again. We can see that this will generate the two lines of output that we received above: ">What is your favorite color?", and ">Blue!".

In closing to this section, make sure you understand what is going on here, since this is at the heart of programming, and it won't be going away any time soon.

Section 3.3 – Raising and Catching Errors

In this section, we will round out our basic vocabulary of function creation with how to report that an error has occurred. This is a vital part of programming, both for yourself and for others, though it is most important when programming for others, since you only want them to use your tools in a very specific way. If they try to use it in a way that will break its functionality, then we want to prevent that from happening *and* provide a meaningful message back to them so they know what they did wrong.

In programming, we call this **raising** or **throwing** errors (it's always raising in Python). There are three fundamental ways to do this. The first is the most intuitive:

```
if something_bad:
        raise ErrorType(“Message”)
```

For example, let's say that we don't want to see any false statements and wish to raise an **AssertionError** when we do see them:

```
>>> if (5 < 3) is False:
...     raise AssertionError("Check yo math")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AssertionError: Check yo math
```

That is the error message that the user would see. Note that **raise** is similar to return, only for errors, so this **raise** statement would break out of all processes and return the error to the user.

Another (shorthand) way of doing this is to use **assert**.

```
>>> assert 5 < 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError

>>> assert 5 > 3
>>>
```

Note that when the assertion is True, then we continue as normal, but if it is False, an **AssertionError** is raised.

The final, and possibly less intuitive way, to raise errors is to **try** something and if that process **throws** an **exception**, then catch it and optionally **raise** it to the user:

```
try:
    'string' + 1
except TypeError as err:
    ## We could do something here if we want
    print "The error message is: %s" % err
    raise
```

When we run this in Python, we get the following back:

```
The error message is: cannot concatenate 'str' and 'int' objects
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Note that the **raise** statement does most of the error reporting for us, but if we wanted a custom message in the **try** construction, we could specify it with a print, or we could **raise** it manually:

```
try:
    'string' + 1
except TypeError as err:
    ## We could do something here if we want
    raise TypeError("String and Numbers don't concatenate!")
```

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
TypeError: String and Numbers don't concatenate!
```

As a last case, if we would like to ignore exceptions (HUGE RED FLAG HERE, DON'T DO THIS REGULARLY, you have been warned), we can use the following two constructions:

```
try:
    'string' + 1
except Exception:
    pass
```

This will catch all Python related exceptions, but it will still throw errors for keyboard interrupts from the user on the command line. If we want to ignore ALL exceptions (I shudder just thinking about it), then this would work:

```
try:
    'string' + 1
except:
    pass
```

To finish this section, I refer you to the Python documentation for a list of built-in exception types that you can use: https://docs.python.org/2/library/exceptions.html

Section 3.4 – Building an Error-Handling FASTA Parser

This section by now should be relatively self-explanatory: we are going to build a file reader (parser) that reads in entries of a FASTA file and stores them in header:value tuples. I'll explain as we go:

FASTA files are the primary way DNA/RNA and Amino Acid information is stored in bioinformatics. The structure of the FASTA file is as follows:

>Header | Some more informaion | Etc.
Sequence
Sequence
>Header | etc.

Example:

>Rosalind_7823
CAATAGCCCTCAACCCTCCCATCGTCGCTGTGACAATCAGACTCCTGTATGGCATTGCAC
CGTAGCGTCTCTTCCGTTGATAAAAAAAAATGTGTGTTCGCCTTTCATGTCCCTTGTAAG
>Rosalind_0317
GCAAGCAATTCCACGTATCATATCGCGAGCGTCGAAGACGACGTCACTCCAAAGTGGTCT
CGCGAACTAGCTCTAACTAATATGGGGCTTCCCGCCGTGTTAAAACATTGTGACGCGACC

So, we have headers that are denoted by “>” and between these headers we have (hopefully) non-blank lines with sequence data on it. These lines can be a fixed width (typically 80 characters) or they can be the whole sequence on one line. We need to build a file reader that can handle both cases.

First, we will want to check and see if there is any whitespace in our file. One thing to keep in mind with whitespace is that “returns,” “tabs,” and “spaces” all are considered white space. A *truly* empty line will only happen at the end of the file, because each of these white space characters has a special character that defines it.

\n = newline on mac/linux
\r\n = newline on Windows
\t = tab

The only time you'll ever have to know what encodes a space is if you're working with encoding, which is outside the scope of this workshop, so for all practical reasons, a space is simply encoded as a space in Python. However, we do have to know what encodes the newline and tab, since these are commonly used. So let's write our first part of our generator function to parse a FASTA file, and make it so this first part skips all whitespace and breaks when we hit something important, i.e. “>”.

```
def fastaParse(infile):
    with open(infile, 'r') as fastaFile:
        # Skip whitespace
        while True:
            line = fastaFile.readline()
            if line is "":
                return # Empty file or premature end of file?
            if line[0] is ">":
                break
```

By now, you should be able to see why this skips all empty lines/white space and ends when we encounter the first important piece of information, denoted by “>”. If this is not immediately clear, either email me with your confusion or stare at it until you can comprehend how it is doing that (or until your head explodes, whichever comes first).

Now we need to add on an error check to see if our first line truly begins with a “>”, just in case.

```
def fastaParse(infile):
    with open(infile, 'r') as fastaFile:
        # Skip whitespace
        while True:
            line = fastaFile.readline()
            if line is "":
                return # Empty file or premature end of file?
            if line[0] is ">":
                break
        while True:
            if line[0] is not ">":
                raise ValueError("Records in FASTA should begin with '>'")
```

Note how we are specifically raising a **ValueError**, because the value of the first character is not what we expected or want. Now let's store the header information in a variable, and begin to parse the sequence data with a second **while** loop:

```
def fastaParse(infile):
    with open(infile, 'r') as fastaFile:
        # Skip whitespace
        while True:
            line = fastaFile.readline()
            if line is "":
                return  # Empty file or premature end of file?
            if line[0] is ">":
                break
        while True:
            if line[0] is not ">":
                raise ValueError("Records in FASTA should begin with '>'")
            header = line[1:].rstrip()
            allLines = []
            line = fastaFile.readline()
            while True:
                if not line:
                    break
                if line[0] is ">":
                    break
                allLines.append(line.rstrip())
                line = fastaFile.readline()
```

Ok. The important thing to keep track of here is where we are calling **fastaFile.readline()**. Those are the spots where we are proceeding to the next line in the file. So we have already checked to make sure we actually have a header, then we store that header (minus the >) in “header” variable, initialize an empty list to hold our lines, and then proceed to the next line (which should be sequence!).

We then need to solve another problem: sometimes in FASTA format, the sequence is on multiple lines, but it represents one continuous sequence. We need to find a way to add all the strings together, or concatenate them. To do this, we will iterate over the sequence lines and add them to our **allLines** list until we reach another header (which means we have no more sequence to concatenate).

The inner loop does this for us. We first detect if we are at the end of the file (if not Line), and if we are, we break. Then we check if we are at a header yet (if line[0] is “>”) and if we are, we break. If we don't detect either of these things, then we have sequence data and we append it (with a strip of whitespace) to the list allLines. Then we proceed to the next line and do it again.

Now, when we *do* encounter the next header, we need to return the header and its associated sequence. Let's add that in.

```
def fastaParse(infile):
    with open(infile, 'r') as fastaFile:
        # Skip whitespace
        while True:
            line = fastaFile.readline()
            if line is "":
                return  # Empty file or premature end of file?
            if line[0] is ">":
                break
        while True:
            if line[0] is not ">":
                raise ValueError("Records in FASTA should begin with '>'")
            header = line[1:].rstrip()
            allLines = []
            line = fastaFile.readline()
            while True:
                if not line:
                    break
                if line[0] is ">":
                    break
                allLines.append(line.rstrip())
                line = fastaFile.readline()
            yield header, "".join(allLines).replace(" ", "").replace("\r", "")
```

All we have added here is a yield statement, but it has a few potentially confusing things in it. First, we are yielding a **tuple**. That will be the following form for every yield statement:

(header, sequence)

Second, within that tuple, the header is simple, but we are yield the **joined** version of the list. Remember we talked about concatenating elements of a list into a string like so:

```
>>> my_list = ["ACTGTGTG", "ACGTG", "GTG"]
>>> "".join(my_list)
'ACTGTGTGACGTGGTG'
```

Where the character within the quotes before the first period is the **spacer:**

```
>>> "-".join(my_list)
'ACTGTGTG-ACGTG-GTG'
```

Then, we also replace any white space that was in the middle of the sequence for some reason, and we also get rid of any \r characters, because sometimes Windows line endings can get “mangled” into that, and we don't want them in our sequence data.

Whew. Only one thing left to do: tell Python when to stop iterating, and make sure we do:

```
def fastaParse(infile):
    with open(infile, 'r') as fastaFile:
        # Skip whitespace
        while True:
            line = fastaFile.readline()
            if line is "":
                return  # Empty file or premature end of file?
            if line[0] is ">":
                break
        while True:
            if line[0] is not ">":
                raise ValueError("Records in FASTA should begin with '>'")
            header = line[1:].rstrip()
            allLines = []
            line = fastaFile.readline()
            while True:
                if not line:
                    break
                if line[0] is ">":
                    break
                allLines.append(line.rstrip())
                line = fastaFile.readline()
            yield header, "".join(allLines).replace(" ", "").replace("\r", "")
            if not line:
                return  # Stop Iteration
        assert False, "Should not reach this line"
```

To tell Python where to stop, we detect if an empty line has been given back to us, and return to end the iteration. Just to absolutely make sure the script stops (potentially overkill, but it's good practice), we raise an error with assert False, and return the message “Should not reach this line”.

Note the indentation and make sure you have it right.

Awesome. That's it. Let's see if it works:

I will use a simple file example for this of the following format:

```
>Rosalind_7823
CAATAGCCCTCAACCCTCCCATCGTCGCTGTGACAATCAGACTCCTGTATGGCATTGCAC
CGTAGCGTCTCTTCCGTTGATAAAAAAAAAATGTGTGTTCGCCTTTCATGTCCCTTGTAAG
TCGCTCATGTAGCACGCTTTAATTAGTCATTTGCGGAGCTCGTTCGACTACTGTTGGCTA
>Rosalind_0317
GCAAGCAATTCCACGTATCATATCGCGAGCGTCGAAGACGACGTCACTCCAAAGTGGTCT
CGCGAACTAGCTCTAACTAATATGGGGCTTCCCGCCGTGTTAAAACATTGTGACGCGACC
```

Now let's put that file path in a variable, then call it with our generator:

```
>>> infile = '/home/lakinsm/Documents/python-workshop/example.fasta'
>>> for pair in fastaParse(infile):
...     print pair
...
('Rosalind_7823',
'CAATAGCCCTCAACCCTCCCATCGTCGCTGTGACAATCAGACTCCTGTATGGCATTGCACCG
TAGCGTCTCTTCCGTTGATAAAAAAAAAATGTGTGTTCGCCTTTCATGTCCCTTGTAAGTCGC
TCATGTAGCACGCTTTAATTAGTCATTTGCGGAGCTCGTTCGACTACTGTTGGCTA')
('Rosalind_0317',
'GCAAGCAATTCCACGTATCATATCGCGAGCGTCGAAGACGACGTCACTCCAAAGTGGTCTC
GCGAACTAGCTCTAACTAATATGGGGCTTCCCGCCGTGTTAAAACATTGTGACGCGACC')
```

There we go! We parsed the FASTA file and returned them as (header, sequence) tuples using our generator. Keep this piece of code handy, because we will use it all the time, and we will return to a more versatile construction of it when we get to Object Oriented Programming.

Section 3.5 – Rosalind Problem: Computing GC Content

http://rosalind.info/problems/gc/

Problem

The GC-content of a DNA string is given by the percentage of symbols in the string that are 'C' or 'G'. For example, the GC-content of "AGCTATAG" is 37.5%. Note that the reverse complement of any DNA string has the same GC-content.

DNA strings must be labeled when they are consolidated into a database. A commonly used method of string labeling is called FASTA format. In this format, the string is introduced by a line that begins with '>', followed by some labeling information. Subsequent lines contain the string itself; the first line to begin with '>' indicates the label of the next string.

In Rosalind's implementation, a string in FASTA format will be labeled by the ID "Rosalind_xxxx", where "xxxx" denotes a four-digit code between 0000 and 9999.

Given: At most 10 DNA strings in FASTA format (of length at most 1 kbp each).

Return: The ID of the string having the highest GC-content, followed by the GC-content of that string. Rosalind allows for a default error of 0.001 in all decimal answers unless otherwise stated; please see the note on absolute error below.

Sample Dataset

```
>Rosalind_6404
CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTTCCC
TCCCACTAATAATTCTGAGG
>Rosalind_5959
CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGGTCT
ATATCCATTTGTCAGCAGACACGC
>Rosalind_0808
CCACCCTCGTGGTATGGCTAGGCATTCAGGAACCGGAGAACGCTTCAGACCAGCCCGGAC
TGGGAACCTGCGGGCAGTAGGTGGAAT
```

Sample Output

```
Rosalind_0808
60.919540
```

We have developed all of the tools we need for this problem. It is up to you to figure out how to get the GC content for each sequence and relate it to its header. You can find the max GC content by using the **max()** function. Also, note that Rosalind wants the %, so remember to multiply by 100.

Session 4 – Namespaces, Pointers, and Vectorized Functions

In this section, we're going to learn how Python searches for variables with **namespaces** through its **scoping** rules and what **names** correspond to on your computer. We will also cover some convenience functions that will allow us to perform operations on vectors of data without using loops; these types of functions are commonly referred to as **vectorized functions**. Finally, we will end with applying some of these vectorized functions to a Rosalind problem, calculating a consensus DNA sequence from multiple sequence alignment data.

Section 4.1 – Namespaces, Scoping, and Variable Assignment

A **namespace** is Python's term for environment. As usual, the Python developers sought a more intuitive name for environments in Python, calling them namespaces instead, which refer to a **space of names**. These names are **objects**. As we learned previously, everything in Python is an object, and every object has a name. This name-to-object mapping is usually a **one-to-one** relationship: each name uniquely refers to an object. However, these names are isolated from one another by their **namespaces**; objects in different **namespaces** *do not know the other one exists.* They are separate insofar as their namespaces are separate. Let's take a look at what this means:

```
>>> knight1 = "Sir Lancelot"
>>> knight1
'Sir Lancelot'
```

We have put the **name** knight1 into our **global namespace**. That means we can access it as long as we keep the Python terminal session open. Additionally, we can reference it inside other namespaces, such as the **local namespace** that we create when we use a function:

```
def localExample():
    for i in knight1:
        print i

>>> localExample()
S
i
r

L
a
n
c
e
l
o
t
```

Here, the function didn't find any variable named knight1 inside its **local namespace** (because we didn't define it inside the function), so it knew to look in the **global namespace** for the **name** and its respective **object**. What happens when we try to assign a **local name** and then retrieve it in the **global namespace**?

```
def localExample():
    local_knight = ""
    for i in knight1:
        local_knight += i
    print local_knight
```

```
>>> localExample()
Sir Lancelot

>>> local_knight
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'local_knight' is not defined
```

Here, the **name** local_knight is not in the **global namespace**, because it only exists within the **local namespace** of the function. For transient objects like functions (where they are "run", they do their job, then they exit), we cannot access **local names** after the fact because they are discarded when the function finishes. However, the function can reference **global names**, because those exist in a place where the function knows to look: the **global namespace**.

Namespaces have a hierarchy so that Python always searches for names in this order:

local namespace → enclosed namespaces → global namespace → built-in namespace

This is also known as the **LEGB** rule.

Another way to phrase this is that Python looks in the *most nested* namespace first, then it proceeds to look at the *next outer* namespace, the next outer namespace after that, then finally the global namespace and Python's built-in names. An example of a built-in name would be any function built into python (which we can override if we make another function of the same name, since the global namespace is searched before the built-in namespace, and we primarily operate in the global namespace when we use the terminal interactively).

This pattern of searching is called **scoping**, and Python has some interesting (and controversial) scoping rules that you should know about. Consider the example above, where we were able to **reference** the knight1 variable even though it wasn't explicitly defined in the **local namespace**. This is a perfectly legal use of **scoping** in Python. However, what happens when we try to change the value of knight1 without defining it in the **local namespace**?

```
def localExample():
    knight1 += ' Du Lac'
    print knight1
```

```
>>> localExample()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in localExample
UnboundLocalError: local variable 'knight1' referenced before assignment
```

This action is not a legal use of Python's **scoping**, for explicit reasons you can read about here. So what does that mean for us?

It means: **we can reference variables that are "out of scope," but we cannot rebind them.** This term, **rebind**, means to change the value of a variable. Another way to think about this is to **change the object the name points to**, which is called **binding**. Let's take a closer look at exactly what we are doing when we assign a value to a variable.

This is the typical mapping of a name to a variable (for all integers, strings, logicals, and other low-level types):

Name1 → **object1**
Name2 → **object2**
Name3 → **object3**

Each name points to its corresponding object, and it is a **one-to-one** mapping. When we rebind Name1 to the value of Name2, what happens?

```
>>> name1 = 1
>>> name2 = 2
>>> name1 = name2
>>> name1
2
>>> name2 = 3
>>> name1
2
```

Take a minute to see what exactly we did here. We assigned the object2 Name2 to Name1, but what we actually did was create a **new instance** of object2 and **bind** it to Name1:

Name1 → **object2$_2$**
Name2 → **object2$_1$**
Name3 → **object3**

I proved that to you by changing the value of name2, and the *value of name1 did not change*. When we assign a name to another name (for low-level types), it creates a new object of the same value as before, but this *does not in any way* **point** to the other name. Name1 does not **point** to Name2, it only points to a new instance of object2. Which means, we are free to change the object of Name2 and it won't affect the object2 assigned to Name1.

This is probably very confusing, but consider that I'm using two different words now, **bind** and **point**. These have very distinct meanings in programming, because one refers to the actual assignment/creation of a new object, which is **binding**, whereas the other simply points a name at an already existing object:

Name1 → Object1 ***this is binding Name1 to Object1***
Name2 → Name1 → Object1 ***this is pointing Name2 at Name1's Object, which is Object1***

So intuitively, what will happen when we change the value of Object1? Well, it will change the bound variable, but *it also changes the variable pointing at it.* This was distinctly not the case earlier, when we made a new binding, and that is the standard way Python will handle variable assignment.

Python typically creates new bindings, and does not work with pointers.

However, there are exceptions for **lists** and **dictionaries**. These objects can have **pointers**:

```
>>> bind = [1,2,3,4]
>>> point = bind
>>> bind[0] = 0
>>> point
[0, 2, 3, 4]
```

Now, when we change the value of the bound variable, we also change the value of the pointer. **Lists and dictionaries are the only examples of this in Python**. So if you run into odd errors due to this, now you know why. Though this section is a lower level discussion of what goes on in the background of Python, it is important to understand, because you will encounter this or need to use it at some point.

Section 4.2 – Importing Packages

One of the most useful features of modern high-level languages like Python is the ability to utilize other's code in an intuitive way. Python offers developers the ability to create their own code for distribution through the use of **packages**. Packages are simply a collection of code in one or more files that fit together to form its own kind of object (which we call a package). Python has built-in packages that come with its distribution; these are referred to as **base packages**. However, you can also install other packages using Python's installation features **pip** or **easy_install**, which you can read more about here: http://dubroy.com/blog/so-you-want-to-install-a-python-package/

For this section, we will use a base package called **re**, which is a commonly used package for working with string searching. **Re** stands for **regular expression**, which is a syntax for finding patterns in text. My intent in this section is not to thoroughly explain regular expressions (AKA regex), so I will keep the examples fairly basic and few.

To use the **re package**, we have to first **import** it. This is done in a couple of ways, and we will discuss their pros and cons.

Method 1: import <package_name>

```
>>> import re
```

This method is considered the gold standard for working with packages. This is because the package retains all of its functions in its own namespace. We can reference the functions within the package using what is called a **decorated variable**:

```
>>> search_string = "The cat says, 'meow'"
>>> pattern = "meow"
>>> for match in re.findall(pattern, search_string):
...     print 'Found "%s"' % match
...
Found "meow"
```

Most of this is basic Python that we have covered already, but the **re.findall** function is specific to the **re package**, and so we have to specify where it is located. We do this with a familiar method call, **re.findall** where **re** is the package and **findall** is a variable (in this case a function) stored in that package. Whenever we import a package with the **import <package_name>** syntax, the package's functions will always be accessed by **package.function**.

Method 2: import <package_name> as <variable>

```
>>> import re as regex
```

```
>>> for match in regex.findall(pattern, search_string):
...     print "Found '%s'" % match
...
Found 'meow'
```

So now instead of the default package name, we have bound the package to a variable called **regex**, which we can use in the same way as we used the default package name. Now, our **decorated variables** will be "decorated" by regex instead of re. This method is still acceptable but less than optimal; people reading your code will have to look at the import statements to see what package you're using, since you have changed the default name.

Method 3: from <package_name> import <method>

```
>>> from re import findall

>>> for match in findall(pattern, search_string):
...     print "Found '%s'" % match
...
Found 'meow'
```

In this case, we have imported a **name** from the package into our **global namespace**. This means we can now use this method without having to decorate it with the package name. This method is useful for personal scripts but not recommended when writing code that others will need to view. In this case, it is not distinguishable whether the **findall** method came from a package or code that you developed in the script body. The reader will have to view your entire code file to determine which is the case, or know to look at the import statements first.

Method 4: from <package_name> import *

```
>>> from re import *
>>> for match in findall(pattern, search_string):
...     print "Found '%s'" % match
...
Found 'meow'
```

The asterisk is a "wild-card" symbol that will import all names in the package into the global namespace. You will now have access to all of the functions in the package without having to decorate them. This method is the least robust method for importing packages, since if you have a function named the same as a function in the package, you will be using whichever one was defined **last**, which is very confusing to both the coder and the reader of the code. This overriding of functions of the same name is called **masking**, and is something we should strive to avoid in our code. When in doubt, use the standard import, "import <package_name>" and work with decorated variables.

Section 4.2 – Tuples

This section will be short and straightforward, since tuples and lists are very similar. Both tuples and lists fall into the category of arrays, but tuples are specifically defined as **an ordered, one-dimensional sequence**, or a **1-D array**. At the basic level, a "flat" or 1-D list is the exact same as a tuple, except the **list is mutable**, while the **tuple is immutable**. Tuples are defined by parentheses in Python and can contain any number of elements. We will work with a 2-element tuple for example:

```
>>> my_tuple = ("spam", "ham")
>>> my_tuple
('spam', 'ham')
```

We can slice tuples as we do any other array:

```
>>> my_tuple[0]
'spam'
>>> my_tuple[:]
('spam', 'ham')
>>> my_tuple[-1]
'ham'
>>> my_tuple[::-1]
('ham', 'spam')
```

However, since tuples are **immutable**, we cannot modify the elements of the tuple:

```
>>> my_tuple[0] = "new_value"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

So why would we want to work with an immutable array when mutable arrays, like lists, are so much more convenient? Tuples are (usually) computationally faster and more memory efficient than mutable arrays like lists. The computer, behind the scenes, can work with tuples faster than lists because it knows the size of the tuple ahead of time, so it doesn't need to find its size every time it does an operation on it.

The other reason for knowing about tuples is that many built-in functions in Python return tuples or accept tuples, so you'll need to know what they are.

Section 4.3 – Zip, Map, and Lambda

Built into Python are several convenience functions for performing specific tasks. From time to time you may find yourself facing a problem where these functions can simplify your code and make your solution more elegant than using many loops.

The first of these convenience functions is the **zip** function. Zip divides objects into a series of tuples depending on the dimensionality of the objects. This kind of operation is referred to as **aggregating** in programming. We can use **zip** in several ways. The first is to group elements by indices from different lists:

```
>>> x = [1,2,3,4,5]
>>> y = ["a", "b", "c"]
>>> zip(x,y)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

So we have grouped the elements by their index; each first element is in a tuple, each second element is in the next tuple, and so on. Zip returns the tuples in a list. You can think of zip's primary action as taking the lists that you input, putting them into the **rows** of a matrix, and then grouping them by columns. This is also know as **transposing** the matrix. We can also **unzip** or **transpose** them again by using the following notation:

```
>>> columns = zip(x,y)
>>> columns
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> zip(*columns)
[(1, 2, 3), ('a', 'b', 'c')]
```

The asterisk before the list denotes that we want to undo the zip. Note that in both of these examples, only the length of the shorter list was used, so if you have two lists of unequal size, only the first x elements will be kept, where x is the length of the shorter list. There are ways to use the length of the longer list, but they involve using the **itertools** package, which you can google, but we won't cover here.

The second convenience function is called **lambda**. **Lambda** functions are simply one-liner functions that we can use as a shorthand when we want to do a simple operation but we don't want to go through the trouble of defining a function. **Lambda functions** can be used in two ways: we can use them as a shorthand way to define a function and assign it to a **name** for reuse, or we can use it in the context of another function, such as the **map** function that we are about to cover. Let's take a look at how this works:

```
>>> add2 = lambda x: x+2
>>> z = 2
>>> add2(z)
4
```

So the general form of a lambda function is as follows:

lambda <var>: return something using <var>

Example:

lambda x,y: x + y

This example would take two arguments (x and y), add them together, and return the value. As previously mentioned, lambda functions are quick to write but are only useful for short functions like this. When combined with **map**, these can be very useful.

The **map** function performs what are known as **vectorized operations**, so it will apply a function to each element in a vector. A vector is simply a 1-D array, such as a flat list or a tuple. Let's see how this works. Suppose we want to add 2 (as in the previous example) but we want to do it **to each element of a vector**. Map was designed for this, and we can do it in conjunction with our previous lambda statement:

```
>>> map(lambda element: element+2, input_list)
[3, 4, 5, 6, 7]
```

So we provide **map** with a function to use, and then pass it a vector on which to apply that function. In general, the **map** syntax is as follows:

map(function, vector)

where the function is applied to each element of the vector. This might not seem particularly useful, but it is extraordinarily useful if you start thinking of your data in terms of matrices. With map, lambda, and zip, we can perform row or column (using transposition with zip) operations with ease. Let's consider a bioinformatics case where this might be useful in the Rosalind problem “Consensus and Profile.”

Section 4.4 – Rosalind Problem: Consensus and Profile
http://rosalind.info/problems/cons/

Problem

A matrix is a rectangular table of values divided into rows and columns. An m×n matrix has m rows and n columns. Given a matrix A, we write Ai,j to indicate the value found at the intersection of row i and column j.

Say that we have a collection of DNA strings, all having the same length n. Their profile matrix is a 4×n matrix P in which P1,j represents the number of times that 'A' occurs in the jth position of one of the strings, P2,j represents the number of times that C occurs in the jth position, and so on (see below).

A consensus string c is a string of length n formed from our collection by taking the most common symbol at each position; the jth symbol of c therefore corresponds to the symbol having the maximum value in the j-th column of the profile matrix. Of course, there may be more than one most common symbol, leading to multiple possible consensus strings.

DNA Strings	A T C C A G C T G G G C A A C T A T G G A T C T A A G C A A C C T T G G A A C T A T G C C A T T A T G G C A C T
Profile	**A** 5 1 0 0 5 5 0 0 **C** 0 0 1 4 2 0 6 1 **G** 1 1 6 3 0 1 0 0 **T** 1 5 0 0 0 1 1 6
Consensus	A T G C A A C T

Given: A collection of at most 10 DNA strings of equal length (at most 1 kbp) in FASTA format.

Return: A consensus string and profile matrix for the collection. (If several possible consensus strings exist, then you may return any one of them.)

Sample Dataset

```
>Rosalind_1
ATCCAGCT
>Rosalind_2
GGGCAACT
>Rosalind_3
ATGGATCT
>Rosalind_4
AAGCAACC
>Rosalind_5
TTGGAACT
>Rosalind_6
ATGCCATT
>Rosalind_7
ATGGCACT
```

Sample Output

```
ATGCAACT
A: 5 1 0 0 5 5 0 0
C: 0 0 1 4 2 0 6 1
G: 1 1 6 3 0 1 0 0
T: 1 5 0 0 0 1 1 6
```

The general strategy for this problem is as follows:

Pseudocode:
Read in the sequences, keeping only the sequences
For each column:
    Count the nucleotide occurrence in each column
    Store the counts in a vector for that nucleotide
    Return the nucleotide with max occurrence in each column as a consensus string

This problem is particularly well-suited to a combination of the **map** function and a dictionary or list object.

Section 4.5 – Rosalind Problem: Mortal Fibonacci Rabbits
http://rosalind.info/problems/fibd/

Problem

Recall the definition of the Fibonacci numbers from "Rabbits and Recurrence Relations", which followed the recurrence relation $F_n = F_{n-1} + F_{n-2}$ and assumed that each pair of rabbits reaches maturity in one month and produces a single pair of offspring (one male, one female) each subsequent month.

Our aim is to somehow modify this recurrence relation to achieve a dynamic programming solution in the case that all rabbits die out after a fixed number of months. See Figure 4 for a depiction of a rabbit tree in which rabbits live for three months (meaning that they reproduce only twice before dying).

Given: Positive integers $n \leq 100$ and $m \leq 20$.

Return: The total number of pairs of rabbits that will remain after the n-th month if all rabbits live for m months.

Sample Dataset

```
6 3
```

Sample Output

```
4
```

This problem is an extension of the previous Fibonacci problem from the earlier session. However, in this case, Rabbits are not immortal, so we need to account for them dying. Consider how you might modify your previous code to account for this.