

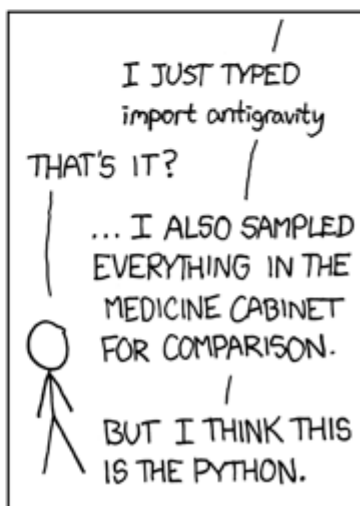
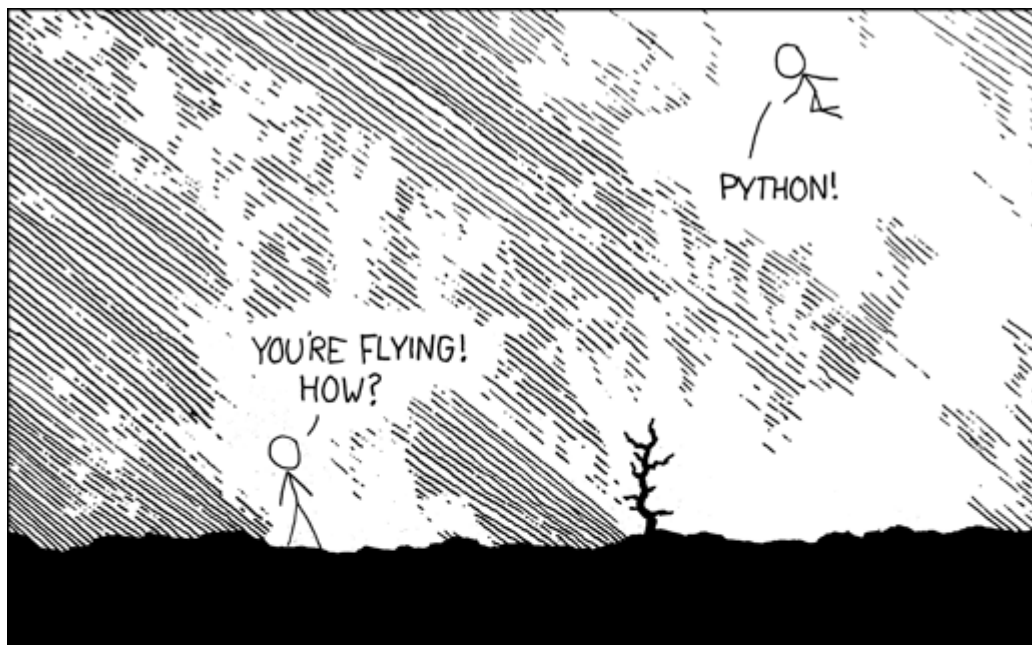
Python Basics (A Reference)

By the end of this lab, you will:

- Demonstrate your new knowledge by submitting a Jupyter notebook which counts the word frequencies in Romeo and Juliet (as described in the Separate Assignment File)
- If this seems daunting we've included this lab from last year for you to help get your bearings.

In this lab you will learn:

- to read data from text files
- the difference between lists, strings, and dictionaries



0. Before we get started...

A big part of becoming a good programmer is knowing when and how to look up documentation and help on commands and expressions you may not have used before - so when you're unsure about something, Google to see if you can find some help online or better yet, just try it! For example:

- What is the difference between an **int** and a **float**? What will you get if you do **13/2**? What do you need to do to ensure that the output will be **6.5**?
- What is the difference between **=** and **==** (that's one equals sign and a double-equals sign)?
- How would you go about reversing a string?

Many questions you can think of can best be answered with: "Try it and find out!" That is not to say you shouldn't ask questions - by all means, do! - but if it's something you might be able to learn the answer to by trying it, give it a go!

- If you don't have a Python book, look up some websites with command references and bookmark them. You can refer to these when you're unsure of the syntax of a command or you forgot exactly how to write a **for** loop or something. Here are a few:
 - [Official Python Documentation](#)
 - [Official Python Documentation reference section](#)
 - [Codecademy](#) - Codecademy now has an interactive Python track. While there are only four courses, this is a great way to get a handle on some basics, as referenced previously.
 - [Python Info Wiki - Beginners Guide for Non-Programmers](#) - this site has a whole list of tutorial links worth looking through. Be aware of what version of Python they are written for.

1. Some quick basics

Before we get started writing a full program, we'll go over a few quick basics to make sure we're on the same page. We'll cover these technical topics at a pretty bare-bones level, and you'll need to seek out additional resources and practice to make sure you have a good handle on them.

I strongly recommend going through the first three tracks of the interactive [Codecademy](#) course on Python (Python Syntax, Strings & Console Output, and Conditionals & Control Flow), especially if you have little prior programming experience. You will learn many basic concepts of programming, how they are implemented in Python, and you will get to do it as you learn it.

(A) USING THE SHELL (you can also use these commands within the Jupyter notebook)

You could open a Python shell by simply typing

\$ python

or log into a notebook server as we've been doing.

Let's get a couple of basic examples out of the way quickly to get used to the shell (or the notebook server) . Here are sample commands and the outputs you should get:

```
>>> 4+4
8
```

```
>>> print("Hello, Ferdinand!")
Hello, Ferdinand!
```

```
>>> 7**2
49
```

[In Python, a double asterisk is the command to raise a number to an exponent.]


```
>>> total = 3
>>> total
3
```

```
>>> total = total + 4
>>> total
7
```

```
>>> total += 2
>>> total
9
```

```
>>> total -= 5
>>> total
4
```

(B) STRINGS AND METHODS

You'll be working a lot with **strings**. As discussed in class, strings are a class of object that are processed in a particular way. Very roughly, strings are sequences of text characters. 

Classes can also contain methods, or functions associated with objects of that class, and strings contain many methods that will be very useful. Here's one example, and you should look for references on all of the wonderful methods available for strings. We'll look at the **upper** method, which converts a

string to upper case. Remember, though - as Prof. Homes noted, strings are immutable - so none of these methods will actually change the stored string. To change the stored string, you have to reassign it, as below.

```
>>> name = "Herman"
```

```
>>> name
```

```
Herman
```

```
>>> name.upper()
```

```
HERMAN
```

```
>>> name 
```

```
Herman
```

```
>>> name = name.upper()
```

```
>>> name
```

```
HERMAN
```

"Slicing"

We discussed in class the concept of "slicing" strings. The basic syntax is as follows: Square brackets following a string variable take a slice depending on what's in the square bracket. A single number gives you a single character, a range using a colon gives you that range (remember - the first number is INCLUDED and the last one is EXCLUDED), and a negative number counts from the back. For more details, look back at the lecture notes from Monday - some of these could be useful!

(C) LISTS

Another useful type of object is the **list**. We will discuss lists in greater depth later, but this is a brief introduction/review.

A **list** is an ordered set of objects that are stored together. Lists can contain anything - strings, integers, whatever. Two examples of lists:

```
["Groucho", "Harpo", "Chico", "Gummo", "Zeppo"]  
[1, 1, 2, 3, 5, 8, 11]
```

Note that:

- Lists exist in square brackets
- Entries are separated by commas
- You can have repeated items
- A question you might wonder - does it matter if you have a space after the commas in the list? Sort of trivial, but interesting to know. What do you think the answer is? Of course, it's our favorite answer - try it and find out!

Lists can be indexed into by position. Python indices beginning with 0, not 1. Thus, if you saved the second list above as a variable called **fib**:

```
>>> fib = [1,1,2,3,5,8,11]
>>> fib[0]
1
>>> fib[1]
1
>>> fib[4]
5
```

(D) CONTROL STRUCTURES

This will be an overview of the primary types of control structures. These are for your own review - so if you feel comfortable with these from class and/or other tutorials you've gone through, feel completely free to skim quickly past them!

Conditional Loops

"If-else" loops are written as in the following example:

```
if temp > 80:
    print("Boy, it's hot!")
elif temp < 50:
    print("Brrr...it's cold!")
else:
    print("Nice and temperate!")
```

If you set a variable called **temp** equal to some number before running this code, you'll see how it works!

Definite Loops

Also known as "for" loops, these take one of the two following forms. To loop through a defined range:

```
for i in range(3):
    print("Counted number",i)
```

And the output would be:

```
Counted number 0
Counted number 1
Counted number 2
```

Here's one with indexing using numbers:

```
drink = "water"
for i in range(3):
    print("Letter", i+1, "is", drink[i])
```

Output:

```
Letter 1 is w  
Letter 2 is a  
Letter 3 is t
```

(Note the **i+1!**)

The second form allows you to iterate through anything indexable directly:

```
cowTypes = ["brown", "white", "mooing", "corn on the cob"]  
for cow in cowTypes:  
    print "I see a", cow, "cow!"
```

Output:

```
I see a brown cow!  
I see a white cow!  
I see a mooing cow!  
I see a corn on the cob cow!
```

The characters in a string are also indexable. Thus, you could also do this with a string:

```
professor = "Ian Holmes"  
for letter in professor:  
    print("Letter: ", letter)
```

Output:

```
Letter: I  
Letter: a  
Letter: n  
Letter:  
Letter: H  
Letter: o  
Letter: l  
Letter: m  
Letter: e  
Letter: s
```

Note that the indexing variable (e.g. above: **i**, **cow**, **letter**, etc.) can be whatever you want. So you could also technically have written **for i in professor** or **for turnip in professor** in the final example and used **i** or **turnip** as your variable instead if you so felt like it. Stylistically, it is good practice to use **i**, **j**, **k**, etc. when indexing through numbers or positions (i.e., a range of some sort) and an obvious name when indexing by item rather than by index position.

INTERLUDE: TABS

In many other languages (Perl, Java, etc.), sub-levels of code are primarily delineated with brackets of some kind or other (like `{ }`). Your indentation, though it may be stylistically nice, does not matter at all in a technical sense.

As you see in the example above, this is not true in Python. Python does care about tabs - sub-levels of code are signified by the presence of a colon at the end of the preceding line (as in **for cow in cowTypes:**) and by indentation of the next line. An indentation may be signified either with a tab or with four spaces, but you should be consistent about it. In some editors and IDEs, hitting the "Return" key after a line ending with a colon will automatically indent the next line for you.

Indefinite Loops

Also known as "while" loops.

```
bonks = 0
while bonks < 3:
    print("Bonk times", bonks, "!")
    bonks += 1
```

Output:

```
Bonk times 0!
Bonk times 1!
Bonk times 2!
```

This might be useful if you don't know when something might end. Beware of infinite loops, however, and realize what can be duplicated with other types of control structures.

(E) MODULES

While Python has decent native functionality, much of Python's power comes from external "modules." Modules are simply Python scripts containing functions; these can be collectively imported into your Python program so that you can use those functions in your program. Python comes with many modules built-in and natively available for import; other modules can be downloaded separately and loaded in.

A basic example of a built-in module is the **math** module. Python's built-in math capabilities are fairly basic; should you want to, for instance, take the logarithm of a number or the sine of a number, you will need the **math** module. Fortunately, this is rather simple (as noted by the floating guy in the comic above); all you need to do is import the module and begin using the function that you want from that module. Let's try a quick example.

Suppose you wanted to take the base-10 logarithm of a number. If you try typing `>>> log10(100)`, you will get an error as Python does not have this function. However - let's import the **math** module! Import it simply by typing:

```
>>> import math
```

You will get no feedback, but it has been imported. I will tell you a secret: the function **log10()** is a function included in the **math** module. So now try taking the base-10 logarithm the same way:

```
>>> log10(100)
```

Oops! Another error! How come? When you use a function that comes from a module, you must call it as a function of that module, as follows:

```
>>> math.log10(100)
```

If you try this, you should get **2.0**. We will be importing modules frequently, including at least one more in this lab.

[How can you know what functions a module contains? Modules have documentation. First import a module, then enter the command **>>> help(modulename)** - for instance, **>>> help(math)**. Full documentation will appear.]

Whew! That's a lot to take in and a lot to get comfortable with.

Opening an external file

One of the first things we need to figure out how to do is to read from a file. To do this, we use the **open()** function. This function can be called with just one argument - the file you want to open. (Technically, the function takes two arguments: first, the filename we want to open; and second, the "mode" we want to open it in. The main mode options are "r" to read from a file and "w" to write to a file. If you don't include a second argument, it defaults to read mode.)

In the spirit of starting simple, let's just start with a script that reads lines from a file and print them on the screen. In your notebook use the magic command `%ls`. Then choose one of those filenames and use it to fill in the "filename" below.

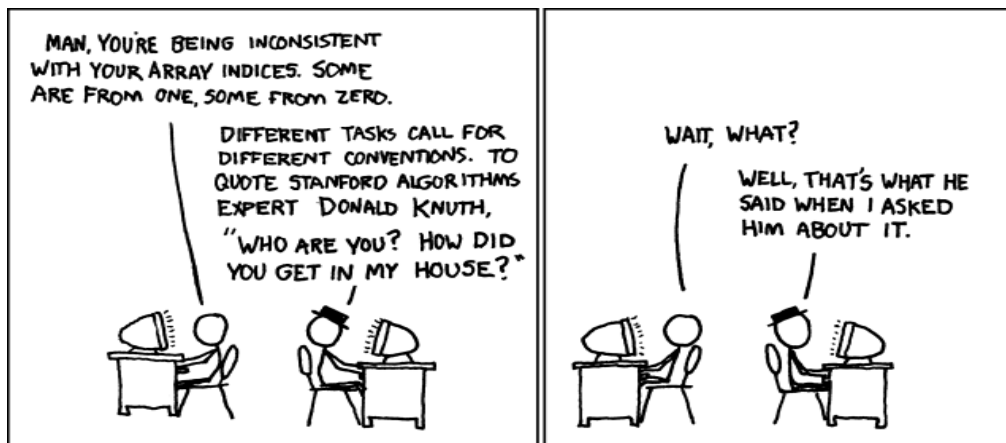
```
import sys
infile = open("filename") # Creates a variable to store the opened file to
                           # reference later
for line in infile:
    print(line)
```

Professor Holmes discussed a couple of different ways to read the lines in a file. Simply indexing through the opened file object will index through the lines, conveniently enough! This code will print out the lines in order. (If you want to explicitly create a list of those lines in order, you can call the file method **readlines** - i.e., **infile.readlines()** would be a list of the lines.) If you call this after you called the code above, what happens? Why?

Lists and Dictionaries in Python

By the end of this section of the lab, you should know:

- how to use lists and dictionaries in Python
- one way to write the reverse complement exercise
- some things to keep in mind when doing error-checking



- The topics of this section - Lists, Dictionaries, and Functions are all covered in various other tutorials, including our buddy [Codecademy](#). Course #5 in their Python track is Lists & Dictionaries, and course #4 is Functions. Feel free to go through their tutorials, the below information, or some of both (below is good for practice using them yourself in our editor and shell).

2. Lists

As you begin to write more complex programs, data structures like lists and dictionaries will become increasingly useful. These list structures are great for storing a set of items and providing easy access to them. At a very simplistic level, the major difference between lists and dictionaries is just the way you access the elements within them - in a list, you access elements using integer indices (i.e., item #0, item #1, etc.), and in a dictionary, you access elements using "keys."

The word "key" is ambiguous on purpose, because nearly anything can be a key. In a real-life dictionary of word definitions, for example, you don't look up the 28th word - you look in the dictionary for the word "consanguinity" (your key) and find its definition. With the dictionary structure in Python, most things can be keys - strings, numbers, etc.

[Technical note: What happens on a very low level when you ask for the element with the key X in a dictionary is to convert X into some numerical value using some built-in hash function and then use that numerical value to access an internal array. Why then, you might ask, wouldn't I write my own array and my own hash function? Well, you could and in some programming languages, you have to if you want to use such a structure. But, the purpose of dictionaries (aka, hash tables) is to provide very quick access to its elements and this all comes down to a good hash function. In computer science classes, you can spend weeks talking about what's a good hash function, so for most of us, using built-in hashes means not having to worry about any of this.]

So with that in mind, let's start with lists. Lists are useful for storing a list of similar items, e.g. a list of numbers, a list of strings, and even a list of lists.

Creating/Using Lists - Let's briefly review some examples of lists and how to go about making them. All of the following are valid ways to create a list:

```
>>> mylist1 = [1,2,3,4,5]
>>> mylist2 = ['a','c','g','t']
>>> mylist3 = range(1,6)
```

Remember, use square brackets to access the elements in a list:

```
>>> print(mylist1[0])
1
```

[Remember - Python list indices start at **0**!]

```
>>> i = 2
>>> print(mylist2[i])
g
```

The index you use to access the list element can even be stored in another variable! This is useful for looping.

Hopefully, this all sounds vaguely familiar. OK, let's try making our own list! We'll work again with our favorite file format (FASTA). Let's try to read in all the sequence names from a FASTA file and store them in a list

```
import sys      # correct import
fastafile = open("filename")    # please put in your filename
names = []      # any variable we are going to *modify* must be created
                  ahead of time.
for line in fastafile:
    names.append(line)
print(names)
fastafile.close()
```

Test this script out with the sample fasta file uploaded to bCourses (next to this Lab webpage). You'll notice that it'll print out a pretty big mess of sequence names, one right after the other separated by only a space. You'll notice that it saves the **>** character into the array as well, which is not that nice. How do we get rid of that first character? There are a couple of ways to do it - one way involves using regular expressions, but we won't be talking too much about regular expressions until next week's lab. Let's do it the most straightforward way using string slicing:

```
import sys      # correct import
fastafile = open(":filename")
names = []
for line in fastafile:
    if line[0] == '>':    # corrected line
        names.append(line[1:])    # <--- Note the modification here -
string slicing!
print(names)
fastafile.close()
```

Try this out. The first character of the string is at offset 0, so slicing with start index of **1** will return everything after the **>** character. (Remember - by leaving out a final index, it will go to the end of the

string.) But there's still a small problem - see how at the end of every line there's a `\n`? That's the "newline" character - it starts a new line when it sees one of those. We've discussed how to get rid of these - we'll use the `.rstrip()` method. Modify that line once more:

```
names.append(line[1:].rstrip())
```

- Iterating through lists - Once you've stored data into a list, you probably want some way to get the data back out. We already saw how to print out the whole array, because the `print` statement is smart and does this for us automatically. But there will be many other situations when you will need to go through each element of the list one-by-one (i.e., iterate through a list). Let's say in our FASTA example, we want to check if we have a sequence for 'protease' in the file. We already read in the file and stored the names in an array, so we just need to check each element in the array and see if any one of them is 'protease'. We know how to iterate already, so let's do it.

```
found = False
for name in names:
    if name == "protease":
        found = True
if found == True: # You can also just write "if found" - "if"
    statements check for truth/existence by default
    print("Found protease in file!")
```

As we might remember, there's another, quicker way to check whether something exists in a list without iterating through the list explicitly. Simply use `in`:

```
found = False
if "protease" in names:
    found = True
    print("Found protease in file!")
```

Try a slightly more useful example of iteration. Modify `seqNamesArray.py` to use a for loop to print out the name of each sequence on a separate line.

3. Dictionaries

Now that we've played around with lists, let's move on to dictionaries. As mentioned before, dictionaries are pretty similar to lists, except that instead of using integers to access them, you use "keys." Every entry in a dictionary consists of a "key" and a "value" - think of a real-life dictionary with the "keys" being words and the "values" being their definitions.

- Creating/Using Dictionaries - There are a couple of ways you can directly assign values into a dictionary. Specifically, in lecture, we talked about

```
comp = {'Cyp12a5': 'Mitochondrion',
        'MRG15': 'Nucleus',
        'Cop': 'Golgi',
        'bor': 'Cytoplasm',
        'Bx42': 'Nucleus'};
```

Note the use of curly braces to define a dictionary, as contrasted with lists, which use square brackets. To access this, we can name a key of the dictionary and access its value:

```
>>> comp["bor"]  
Cytoplasm
```

Working with the same FASTA file we were using for **seqNamesArray.py**, let's now read the data into a dictionary. This time, we'll read both the names and the sequences and store them into the dictionary accordingly, with the names used as keys. Remember that elements in a dictionary are accessed the same way as a list - using the square brackets [].

```
sequences = {} # Creates an empty dictionary - note the uses of curly  
braces  
infile = open("filename")  
name = None  
seq = ""  
for line in infile:  
    newline = line.rstrip()  
    if newline.startswith(">"):  
        if name != None:  
            sequences[name] = seq  
            seq = ""  
        name = newline[1:]  
    else:  
        seq += newline  
sequences[name] = seq  
infile.close()
```

If you try to put a dictionary into a print statement, it will print out the results like a list, only the order may be arbitrary. There's an internal reason for the ultimate ordering of the items, but it is not relevant. One way to check whether you put in all the data is to use the built-in methods **.keys()** and **.values()** for dictionaries, which return actual lists that you can just print out nicely. Simply try adding the line:

```
print(sequences.keys())
```

- Iterating through dictionaries - Like lists, sometimes you will need to go through each element in your dictionary one-by-one. The easiest way to do this is via the **keys** and **values** methods of dictionaries, which return a list from that dictionary that you can iterate through using a for loop. Say we want to see if there's a sequence for 'protease' in the file again. A crude way to do it would be:

```
for element in sequences.keys():  
    if element == "protease":  
        print("there is a protease in file")
```

- But why check keys lists when you can... - So now we come to one of the nice things about dictionaries. In the above example, we wanted to find out if there is a protease in the sequence file. We did this by reading the file into a dictionary and then accessing the list of keys, then checking that list for the existence of a key "protease". But why don't we use the same **in** that we used to check lists? When you use an **in** check with a dictionary, it automatically checks the list of keys to determine **True** or **False**:

```
if "protease" in sequences:  
    print("there is a protease in file")  
if "kinase" not in sequences:
```

```
print("there is no kinase in file")
```

Nice, huh? The convenience of accessing items by keys rather than indices will be appreciated when you start dealing with larger amounts of data! Not only is the code you write shorter, but the time it took to find out whether something exists in the dictionary is much much shorter than it took to look through an entire list of keys. This doesn't matter so much for this short example but when you have massive amounts of data, you'll appreciate this performance difference.

4. Functions

Some of you have figured out functions already. Good use of functions will be expected in future code you write - but don't worry, you'll want to use them because they're so useful!

Functions are important to have a handle on. Fortunately, they're fundamentally not anything very new. A function is just a sort of sub-program, a set of commands that can be invoked by calling that function. Methods are an example of functions - each one has some underlying functionality that you access by calling that name. Functions are used to reduce code duplication and increase the modularity of your program. A function can take inputs (though need not) and can be repeatedly invoked. Functions can be designed to **return** something upon being invoked. For instance, a function can manipulate numbers and return the result of the manipulation, or it can run a comparison and return **True** or **False**. Output can in fact happen without "returning," though - a **print** statement will still print to the screen. Let's look at an example.

```
def square(num):  
    ==return num*num==
```

```
>>> square(3)  
9
```

Note that you could also write the program like this:

```
def square(num):  
    print(num*num)
```

and seemingly get the exact same output. What's the difference? The difference is what you can do with the function. **return** is tied to the meaning of the function, while printing will simply display the result. So if you tried this with the second function:

```
>>> score = square(3)
```

It would immediately output **9**, but if you then typed in **score** you would get nothing. However, if you tried this with the function that **returns** the output, typing in **score** would display **3**.

Whereas variable naming convention, for instance, is to use camel-case (**myVariable**), functions in Python are conventionally named using underscores (**my_function**).

Docstrings

All functions you write should be notated with a docstring. This is a multi-line string placed first thing after beginning to define the function with an explanation of what the function is, what the inputs are, and what is returned. Docstrings begin with three quotation marks and end with three quotation marks. You can access the docstring for a function by pushing shift+tab after you open the parenthesis to call it. Here's an example docstring from the [StyleGuidelines](#) page:

```
def motion(v,t):
    Multiplies a velocity and time to get the position: x=vt
    v: velocity (number)
    t: time (number)
    Returns: v*t ( = x) (number)

    # That thing above is the docstring!
    return v*t
```

5. Writing to files

We already know how to read into files - now, let's quickly go over to writing to files. This is actually quite simple!

When we wanted to open a file to read from it, we used a command as such:

```
>>> infile = open("myfile.txt")
```

And then we could use the **.readline()** method or similar ones to read the lines in the file. The secret here is - the **open()** function actually takes two arguments. The first is the file name, which we always input. The second is the mode we want to open the file in. The main two modes are reading and writing. If no second argument is input, it defaults to read mode. To choose a mode explicitly, we add a second argument, which is either the string **"r"** or **"w"** (you do have to put the quotation marks!).

Let's write a file with a quick note to . We'll begin by opening a file. Handily, you don't actually have to pre-create a file to open it for writing - "opening" a new file name will both create the file and prepare it to be written to. Let's open a new file for writing:

```
>>> outfile = open("ilovehw.txt", "w")
```

Now that it is open, let's write a couple of lines to it. This is accomplished by calling the **.write()** method of the file variable. One thing to remember is that a write command does not insert a newline automatically - you must make sure you insert a newline wherever you want one or the next write command will simply go at the end of the last one.

```
>>> outfile.write("Dear GSI,\n")
>>> outfile.write("I love homework. Please give me more.\nOn second
thought, please don't.\n\nSigned,\nme.")
```

Note how you can concatenate the next lines directly next to a newline character (`"\n"`) - you don't have to issue a new "write" command if you don't want to. Whether you do will depend on the nature of your program.

The last thing you need to do is close the file:

```
>>> outfile.close()
```

If you don't, you probably won't be able to see what you have written to the file. If you now try opening this file on your computer, you should be able to see your note!