(2) First Steps in Scripting

(2.1) Scripting Hello World

(2.1.1) Create a python script file called hello_world.py which prints Hello World to the screen and run it from the command line.

(2.1.2) Modify the script to include a multiple line comment explaining the purpose of the script. If running in a Unix environment, put a "shbang" comment at as the first line so that the script file can be run as a python script directly from the shell without invoking the "python" command explicitly; make the script executable and run it.

(2.2) Scripted Downloading from NCBI

We're not going to download that actual SRA data (it's too big!), but we'll make a script which prints the commands that would need to be run to the screen. It would then be easy to run them if we wanted to.

(2.2.1) Start a new script file (using the "shbang" method or not whichever you prefer) called wget_sra_files.py. Begin by making a multiline comment at the top explaining what the script will do. This is to remind you or anyone else to might use the script. Also put single line comments throughout the script to explain what each part does where ever you think it is not already obvious.

A reusable script normally takes some information from the command line such as the name of an input file, so that it can be run on different data each time without having to edit the script. For now we will just put the name of the input file inside the script itself, but have it at the top, separate from the code itself. This way the script will actually better document what you have done. Which approach is better will depend on the situation: whether the script is meant to be a reusable tool or to document a pipeline run only once.

Start off by creating a string variable called inp by assigning it the filename of the list of SRA ids:

```
inp = 'SraAccList.txt'
```

Now to change the script to work on a different file only this line would need to be changed.

If you haven't downloaded the file already, you can create a fake list in your text editor, as long as each line starts with 'DRR' followed by 6 digits.

To check the script will run, put a temporary line printing inp to the screen. Save the script and run it. If it doesn't print the filename to the screen, try to track down and fix the bug. Once it works you can remove the print command by commenting it out or deleting the whole line.

(2.2.2) Next open the file and assign the "file handle" to a variable called f. To do this use the builtin function open, and pass the name of the file to the function like this : f = open(inp). Usually where ever you see something in brackets like this it signifies a function is being called, and the variables inside the brackets are being passed to the function. Later we will create our own functions.

For now, to check the script works correctly, print the output of the type command applied to f. If the file opened correctly f should be of type 'file'. If there was an error the script should stop with an error message. Now that we are using a script we must explicitly use a print statement to get something to show in the screen, so to show the type of f you must use: print type(f).

Add this line to end of the script, save and run it. Once it works, remove or comment the print statement.

(2.2.3) What would happen if the file was not found by the script? Temporarily change the filename to something incorrect so that the file cannot be opened. Save and run the script. What does the script do now? Change it back once you are done.

(2.2.4) We have been running the script in such a way that the file is opened but not explicitly closed before the script ends. Python will automatically close any open files when the script ends, but it is good practise to explicitly close the file as soon as you are finished with it. To make sure we don't forget to do this, before we write the code to deal with the file's contents, let's write the line that closes the file immediately after the line that opens it, and then fill in the middle part afterwards. This way we are much less likely to forget to close the file.

You might think close(f) is the correct way to close a file, but python is an object oriented language and like to keep functions associated with files neatly contained within the file objects themselves. To see what I mean, let's examine the file handle f in the interpreter environment that we used last week. Go to a command prompt and launch python. Make sure you are in the same directory as the SraAccList.txt file. Open the file and save the handle as f,just like the script does:

```
>>> f = open('SraAccList.txt')
```

Check that close(f) does not correctly close the file:

```
>>> close(f)
```

Use type(f) to check what datatype f is. To get more information about file objects use the help command:

```
>>> help(f)
```

You will get the same information if you call help on the name of the data type:

```
>>> help(file)
```

You should see something starting with 'class file(object)' followed by help information. 'class file(object)' means that file is a class which is based on a more primitive class called object, but we won't worry about that now.

```
file(name[, mode[, buffering]]) -> file object
```

This tell you how to create a file object. In general to create an object of any type you use the class name, followed by brackets containing any additional options required. In this case we need to give at least the filename, and optionally something called mode and buffering, and a file object will be returned to us. The default mode is reading, which is what we want to do now. If the mode is writing and the file already exists it will be 'truncated', meaning erased!

When we did f = open('SraAccList.txt') we created a file object. If you do:

```
>>> help(open)
```

you will see that open is basically just an alternative way to call the file() object creation function, and it takes exactly the same options (name, mode, buffering). help(f) also shows, further down, how to close an open file. The function is called 'close'. So why didn't close(f) work? The key point to realise is that everything listed as a function by help(f) is a function inside the class itself. To access those functions you need to use the following notation:

```
>>> f.close()
```

This is handy because there is no chance of accidentally calling a function of the same name meant for a different type of object (a network or database connection perhaps). To access the function you must have on object of type file and put a dot after it.

Add a line to close the file after the line where you opened it. Then make some space in between where you will write the rest of the script.

(2.2.5) For loop:

```
for line in f:
```

Next we begin a 'for loop' to apply the same operations to each line in the file. This piece of code means 'return each line in the file one at a time and assign it to the variable 'line'. The colon marks the beginning the of the code that will be repeated for each loop. The lines that follow must be indented using 4 spaces. When the indenting ends that signifies the end of the loop (this should be the line that closes the file.

To quickly test the for loop lets simply print line to the screen. Put a print statement in the for loop to display line to the screen, save and run. You should see all the SRA ids printed to the screen.

(2.2.6) Processing the ids

You should have found that there was a blank line in between the ids. This is because each line ends with a newline marker, and python prints these to the screen as well as starting a new line after each print statement. Therefore each id generates two lines of output, one empty. This is a general problem from data read from a text file. To see this problem, go to the interpreter and create a string called s which contains some text but ends with a newline character \n. Print the string using the print statement and verify that it generates an extra blank line.

To remove unwanted 'whitespace' characters, such as spaces, tabs and newlines from the start and end of strings python has a function called strip(), which belongs to the class str (meaning string). To find the help documentation for the function we would do:

```
>>> help(str)
```

and scroll down to the strip(...) function. There are also lstrip and rstrip to apply the operation only to the left and right ends of a string.

Apply the strip() function to the string s in the interpreter and print the result to the screen to verify that it does not generate an extra blank line.

```
>>> print s.strip()
```

Print s itself after you have applied the strip function. Notice that s remains unchanged. Why is this?

Instead of changing the string itself the function returns a new string with the whitespace removed but keeps the original string unchanged. In python strings are what is called "immutable", meaning they never change once created. To actually change s you would assign the returned string back to s like this:

```
>> s = s.strip()
```

Back in the script file, use the strip function to remove the newline character from the line and assign the new version a name of uid (meaning the unique identifier of the SRA file) or whichever name makes most sense to you. You will have to type this name multiple times to finish the script so don't make it too long! Add a temporary print statement to print the uid to the screen. Run the script to check that the ids now appear without extra spaces between them.

(2.2.7) Lists

The URL of the SRA file contains the uid in four different forms: DRR, DRR+3 digits,DRR+6 digits,DRR+6 digits+file extension. It would be possible to create each of these parts of the URL assigned to a different variable name:

```
part1 = ...
part2 = ...
part3 = ...
part4 = ...
```

Then put them together to create the final URL:

```
cmd = 'wget http://....' + part1 + '/' + part2 + ... etc
```

But this is tedious and easy to make an error, such as putting the wrong part number, and harder to read back and understand later on. Instead we will create a list object to contain all four parts. Python lets us create a list with all the right parts in as a single line of code:

```
id_list = [uid[:3], uid[:6], uid, uid + '.sra']
```

Let's break this line down. The square brackets indicate a list is being created. In the interpreter let's create an empty list to begin with and assign it the name l:

```
>>> l = []
>>> type(l)
>>> help(l)
```

Help will give us help about the class list as before. Help tells us that we could also us the class name followed by brackets: list() to create an empty list. [] is just a short cut to the same list creation function. If we wanted to add items individually to the list we can use the append function. Again this is a function inside the list class, so to use it on a list we use the variable name of the list then a dot then the append function. Python lets us add any type of item to a list, the items don't have to be the same data type at all. Try it: append an int , float and string to l:

```
>>> l.append(123)
>>> l.append(1.2e6)
>>> l.append('mystring')
>>> print l
>>> len(l)
```

len will tell us the length of the list. We can also of course add items we have assigned variable names:

```
>>> s = 'mysecondstring'
>>> l.append(s)
>>> print l
```

Notice that items appear in the list strictly in the order you appended them. This is the defining property of a list, which in other languages you may hear referred to as an "array". To access a particular item in the list you only need to know its position. To see item number 2 in the list, remember that python counts things starting from 0, meaning "offset zero places from the beginning", so 2 would be offset 1. Use the following syntax:

```
>>> l[1]
```

The first item in the list is at offset 0, and the last item is at offset len(l)-1:

```
>>> l[len(l)-1]
```

To avoid writing len(l)-1 all the time there is a handy shortcut: negative number indicate position starting from the back, -1 meaning the last item, -2 the second from last etc:

```
>>> l[-1]
>>> l[-2]
```

To create a list already containing items, simply list them in between the square brackets when you create the list:

```
>>> l = [1,2,3,4,5,6,1.234,'the end']
>>> l[-1]
```

Accessing a range of items from a list creates a new, sub list. To say which items you want to include using use python's "slice" notation. For example to return the first three items use:

```
>>> l[:3]
```

To get the last 4 items use:

```
>>> l[-4:]
```

In general slice notation takes the form [A:B] where A is the offset of the first item to *include* and B is the offset of the first item to *exclude*. If A is omitted it defaults to 0, meaning the sublist starts at the beginning; if B is omitted it means include items up to the last one. Therefore:

```
>>> l2 = l[:]
```

will create a copy of the whole list. To join two lists together we can use the '+' operator:

```
>>> l3 = l + l2
```

Python's slice notation is very useful and is essential to learn if you're going to write good scripts yourself, and also handy if you only want to follow other people's. The great thing is that it also applies to strings. Think of a string as a list of characters, then the same rules apply for generating substrings and for generating sublists. Back to our SRA id example you now know how to generate a substring from the full id. We need strings containing the first 3 characters of the id, the first 6, the full id and one with the full id plus the filename extension. Check you can create these in the interpreter if you wish:

```
>>> s = 'DRR013866'
>>> s[:3] etc
```

To add the filename extension '.sra' we can just use the '+' operator in the same way as we did to join two lists together. You should now be able to create a list containing the required components to construct the full URL. Do not include the forward slash character, this will added as we join the strings together into the final URL.

(2.2.8) Creating the final URL

An example of a full URL is:

```
wget http://ftp-trace.ncbi.nlm.nih.gov/sra/sra-
instant/reads/ByRun/sra/DRR/DRR013/DRR013876/DRR013876.sra
```

The first part is generic, so we simply create a string containing the URL up to "...ByRun/sra/". Next we will use a string function called join, which is a handy way to join a list of strings together. In the interpreter create a list of strings of your choice, eg:

```
>>> l = ['a','b','c']
```

Now create a string containing these joined up using the join function from the string class. help(str.join) will tell you about the function.

```
>>> ''.join(l)
```

Here '' is two single quotes, without anything inside them. This means "join the strings from the list into a new string separated by nothing". If we wanted the strings separated by dots we would use:

```
>> '.'.join(l)
```

We need them joined by forward slashes, so add the appropriate line to your script. Save and run it. Check that the appropriate URLs are printed.

(2.2.9) Running the command
To actually run the command you would need the command "system" from the module called os. The function is contained in a module that is not loaded by default. Use the import command to load it. It cannot be accessed at all before being imported:

```
>>> help(os.system) #should give an error
>>> import os
>>> help(os.system)
```

Simply pass the command to the function: os.system(cmd)