Intro to Python

Lesson 1

(Haddock and Dunn Chapter 8, updated for Python 3)

Python is a widely used programming language, which is particularly relevant for biologists as it is commonly used by many scientists. Python is known for being well suited for beginner programmers compared to other languages and having the ability to easily manipulate text data, among other reasons. The need for complex yet easily obtainable data manipulation is becoming a necessity as the sciences become more interdisciplinary and with the explosion of extremely large datasets, notably in genomics.

To download Python on your computer, go to:

Mac: https://python.org/downloads/mac-osx/
Windows: https://python.org/downloads/windows/

*There are some differences for Windows users. Mac OS X, Linux, or most types of Unix are already set up to run Python programs.

Writing a Program

We will use an example DNA sequence to learn how to write a basic program in Python and use common built in functions. We will be using command line text editors for writing and running code `nano` or `vim`. Comments in blue will indicate code written in the text editor and comments in brown will indicate commands used in the terminal (bold indicates commands you need to type in).

Creating new directory

First, we need to create a directory to save scripts. Go to your home directory with the command `cd ~/`. The command `cd` means 'change directory' and the tilde (~) is a shortcut to your home directory. Use the command `mkdir` to make a new directory (this is equivalent to right clicking and selecting 'New Folder' using the GUI) and name the directory `scripts`, or something that informs you what is in the directory.

```
host:~ name$ cd ~/
host:~ name$ mkdir scripts
host:~ name$ cd scripts
host:scripts name$ ls
```

You can see that once you move into the scripts directory, the host name is now `scripts` and not ~ (home). The last command, `ls`, means 'list' and will list all of the items in the directory. In this case, there should be nothing in the `scripts` directory.

Making comments using

When writing a script, it is common to add comments. Comments can help other people to understand sections of your script and also as reminders for yourself when looking back on old scripts. Notes that are commented out using the hash mark or pound sign (#) will be ignored by Python and won't interfere with other code. This can also be used for troubleshooting. If there is a problem with your code, you can use # to comment out sections of code to find the issue area. Each line that you wish to comment out needs to begin with a #. For large blocks of code, however, it may be easier to add three double quotes (″″″) to the beginning and end of the section you would like to comment out. Comments are often used at the beginning of a script to explain what it is for, who wrote it, and other important details. You can add comments anywhere in the script after the first line, which will indicate which program the file is to be sent to; in this case, Python.

During this class we will make a python script with documentation. The first line is the:

Shebang #!

The first line of code in the script needs to start with a shebang: #!. The shebang, when used at the beginning of a text file, will send then file to the program named after. First, we need to add into the script where to find Python. You can do this by using the `which` command. Executing `which` followed by the name of a program will give you the absolute path of that program.

```
host:~ name$ which python
/Users/name/anaconda3/bin/python
```

You could use this in your script, however, in the case that this script is shared with other colleagues, it is unlikely that they have Python in the same place on their computer as you do. A better command is `env`, which can find Python (or whichever program you are looking for) on any computer. This command sends the script to `env` instead of Python and `env` then finds Python, no matter the location. So, lets open up a file called dna_example.py and for the first line of code we will write the shebang line:

```
host:~ name$ nano dna_example.py
```

```
#! /usr/bin/env python3
```

`print` statements

Using `print` allows you to tell Python to print out a specific statement, which can be simple or complex. Here, we will print out a small DNA sequence to start with. We will make the DNA sequence a string variable so it can be easily used in the subsequent code. A string variable is a sequence of text characters. Next, we will tell the code to print out the sequence by calling the variable name. This section of code should be added underneath the line that begins with the

shebang. You can add spaces between lines but make sure there are not single blank spaces because that may cause issues with the script.

```
DNASeq = 'ATGAAC'
print('Sequence:', DNASeq)
```

Note: When identifying the start and end of a string, you can use either single (') or double (") quotes. This can be an issue if the string contains a quote. For example, if you have a string that contains the state name `Hawai'i`, you need to use double quotes in your code in order to remove confusion of where the string begins and ends: `"Hawai'i"`.

Making files executable

To run the script, save the script as `dna_example.py` into your `scripts` directory. Python scripts should be saved with the `.py` ending. To run this file, we need to make sure the file is executable, and the shell has permission to run it. You can test this by trying to execute the file in the terminal and if it returns '`Permission denied`' you know that you need to adjust the permissions. You can also list the files in the directory using `ls` and adding `-l` to view the current state of each file in the directory, which may look like this:

```
host:scripts name$ ls -l
-rw-r--r--@ 1 name  staff    70 Jul 06 08:18 dna_example.py
```

The output will have dashes (meaning permission is not granted) and other letters (r, w, x). r stands for who can read the file, w stands for who can write to it, and x stands for who can execute it. The first three letters indicate the user's (you) permissions. In this case, we want to add the x to the user's permissions so we can execute the file.

To do this, use the command `chmod`, which stands for change mode. After this command, add whose permission we want to change (user or 'u') and what permission we want to add ('x'), and lastly, what file we are talking about. Use `ls -l` to double check that the permissions are set correctly.

```
host:scripts name$ chmod u+x dna_example.py
host:scripts name$ ls -l
-rwxr--r--@ 1 name  staff    70 Jul 06 08:24 dna_example.py
```

You can see that there is now an `x` after the first consecutive `rw`. Now you are able to execute the file in the terminal. You can also remove permissions by replacing the + with a `-`.

Depending on the file system on your computer you might have to run it like this:

```
host:scripts name$ ./dna_example.py
```

Executing a file in the terminal

In the `scripts` directory in the terminal, type in the file name. When you begin typing, you can use the tab key to auto-fill in the rest of the file name. This will make sure the file name is typed correctly. If there are multiple files in the directory that begin with dna, for example, pressing tab once will not auto-fill the file name. Pressing tab twice will list the files that begin with dna if there are multiple. Once the file name is written out in the terminal press enter to execute the command. You should see:

```
host:scripts name$ dna_example.py
Sequence: ATGAAC
```

`len()` function

The built in `len()` function finds the length of an item. This command can be very helpful when working with long DNA sequences. Similar to what we did before, we can create a variable that holds the sequence length to use later in the script.

```
SeqLength = len(DNASeq)
print ('Sequence Length:', SeqLength)
```

See that we used the variable `DNASeq` that we made earlier within the `len()` function. Save the code again and execute it in the terminal.

```
host:scripts name$ dna_example.py
Sequence: ATGAAC
Sequence Length: 6
```

Using multiple types of variables

Some common variable types are strings (a sequence of text characters), integers (whole numbers without fractions), and floats (numbers with decimal points). You can use Python to do mathematical calculations but in some instances you may need to specify what type of variable you are looking for. You can use the basic `print` command to act as a calculator. For example, `print 5 + 7` will give the output `12`. Python took both numbers as integers and added them together. You can also create a string by adding quotes: `print '5' + '7'` with the output `'57'`. If you try to execute the command `print '5' + 7`, Python will give you an error because it does not know if you are looking for an integer or string. To specify, you can use the commands `str(), int(),` and/or `float()`.

Instead of creating an entire new script just to test out a few lines of code, you can use the interactive prompt in Python to save time. To do this, call `python` from your home directory.

You can use the interactive prompt to test the code before adding it into a script and, therefore, saving yourself from some troubleshooting in the future.

```
host:scripts name$ cd ~/
host:~ name$ python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

The output will tell you which version of Python you are working in and the three arrows, >>>, indicate the commands are now executed in Python. To get out of Python in the terminal press 'control d'.

Here, we show some examples of combining multiple types of variables. Adding together a string and an integer will return an error.

```
>>> print ('7' + (3*2))
TypeError: can only concatenate str (not "int") to str
```

You need to tell Python to convert `(3*2)` to a string.

```
>>> print ('7' + str(3*2))
76
```

You can also convert a string or integer into a float variable by using the command `float()`.

```
>>> print (float('7.5') + 3*2)
13.5
```

Additionally, you can use `float()` to print out numbers set in scientific notation.

```
>>> print (float(3.642e-2))
0.03642
```

Note: Older versions of Python (2.7 or earlier) will truncate basic math calculations that have a decimal. For example, `5/2` would give the output `2` instead of `2.5`. Newer versions, however, have changed this so the output should be `2.5`, not `2`.

Add the `float()` function to the `SeqLength` variable. The full script should look as follows:

```
#! /usr/bin/env python

DNASeq = 'ATGAAC'
```

```
print ('Sequence:', DNASeq)

SeqLength = float(len(DNASeq))
print ('Sequence Length:', SeqLength)
```

Make sure to close all parentheses and watch for correct use of lower- and upper-case letters. The output should now look the same as before, but the sequence length will have a decimal place.

```
host:scripts name$ dna_example.py
Sequence: ATGAAC
Sequence Length: 6.0
```

`.count()` function

The function `.count()` is a built in function for string variable types and is known as a method. This function counts the amount of times a substring is found within a string. For DNA sequences, you could use this function to find how many times a small sequence is found within a large sequence. We will use this function to count how many of each base are in the DNA sequence we named `DNASeq` earlier. Add the following code to your script.

```
NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')
```

We created new variables (*e.g.,* 'NumberA') assigned with the number of bases of each base found within the variable `DNASeq`. Adding `.count()` after the variable name of interest will count the number of times the string placed within the parentheses (A, C, G, or T) is found within the variable.

Note: There is a more efficient way to write this code with fewer lines by adding everything to a list and using a loop.

Using these new variables, we can find out the fraction of each base within the sequence.

```
print ("A:", NumberA/SeqLength)
print ("C:", NumberC/SeqLength)
print ("G:", NumberG/SeqLength)
print ("T:", NumberT/SeqLength)
```

The output should look as follows:

```
host:scripts name$ dna_example.py
Sequence: ATGAAC
```

```
Sequence Length: 6.0
A: 0.5
C: 0.16666666666666666
G: 0.16666666666666666
T: 0.16666666666666666
```

% operator

The % operator can be used to create strings of different variable types. In a `print` statement, you need to add the % as a placeholder in the first section (the string you wish to print) denoting what type of variable you would like it to be: `%d` for an integer, `%f` for a float, and `%s` for a string. Placing the % operator after the string, you then need to add which variable you want to be placed within the % placeholder. For example, adding the following to your script:

```
print ("There are %d A bases." % (NumberA))
print ("There are %d C bases." % (NumberC))
print ("There are %d G bases." % (NumberG))
print ("There are %d T bases." % (NumberT))
```

will output:

```
There are 3 A bases.
There are 1 C bases.
There are 1 G bases.
There are 1 T bases.
```

The variable within the parentheses after the % replaces the % placeholder within the string with the requested variable type.

You can add multiple % operators in a print statement. Make sure the variables listed to the right of the % are in the order you want them to appear in the statement.

```
print ("A occurs in %d bases out of %d." % (NumberA, SeqLength))
print ("C occurs in %d bases out of %d." % (NumberC, SeqLength))
print ("G occurs in %d bases out of %d." % (NumberG, SeqLength))
print ("T occurs in %d bases out of %d." % (NumberT, SeqLength))
```

Output:

```
A occurs in 3 bases out of 6.
C occurs in 1 bases out of 6.
G occurs in 1 bases out of 6.
T occurs in 1 bases out of 6.
```

You can also use the % operator to control the number of decimal points. To do this, use `%f` and add the number of decimals you want between the % and `f`. To add a percentage sign to the

number, you need to escape your code with %%. In this case, trying to escape the code with \% will not work. The following code will print out the percentage of each base in the DNA sequence with a single decimal point.

```
print ("A occurs in %.1f%% of %d bases." % (100 *
     NumberA/SeqLength, SeqLength))
print ("C occurs in %.1f%% of %d bases." % (100 *
     NumberC/SeqLength, SeqLength))
print ("T occurs in %.1f%% of %d bases." % (100 *
     NumberT/SeqLength, SeqLength))
print ("G occurs in %.1f%% of %d bases." % (100 *
     NumberG/SeqLength, SeqLength))
```

Make sure to divide the number of bases by the total sequence length and multiply by 100 to get the percentage.
Output:

```
A occurs in 50.0% of 6 bases.
C occurs in 16.7% of 6 bases.
G occurs in 16.7% of 6 bases.
T occurs in 16.7% of 6 bases.
```

`input()` function

The `input()` function (formerly known as `raw_input()`) allows you to paste in a sequence of your choice when running the script. A prompt will appear for you to enter your sequence. This allows you to use a single script easily for multiple DNA sequences, instead of creating multiple scripts. Below the original `DNASeq` variable, override the variable using `input("Enter a DNA sequence: ")` instead of the DNA sequence we used before. Your full script should look something like this, depending on the print statements you decided to keep or delete.

```
#! /usr/bin/env python

DNASeq = 'ATGAAC'
DNASeq = input("Enter a DNA sequence: ")
print ('Sequence:', DNASeq)

SeqLength = float(len(DNASeq))
print ('Sequence Length:', SeqLength)

NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
```

```
NumberT = DNASeq.count('T')

print ("A:", NumberA/SeqLength)
print ("C:", NumberC/SeqLength)
print ("G:", NumberG/SeqLength)
print ("T:", NumberT/SeqLength)

print ("A occurs in %d bases out of %d." % (NumberA, SeqLength))
print ("C occurs in %d bases out of %d." % (NumberC, SeqLength))
print ("G occurs in %d bases out of %d." % (NumberG, SeqLength))
print ("T occurs in %d bases out of %d." % (NumberT, SeqLength))

print ("A occurs in %.1f%% of %d bases." % (100 *
     NumberA/SeqLength, SeqLength))
print ("C occurs in %.1f%% of %d bases." % (100 *
     NumberC/SeqLength, SeqLength))
print ("T occurs in %.1f%% of %d bases." % (100 *
     NumberT/SeqLength, SeqLength))
print ("G occurs in %.1f%% of %d bases." % (100 *
     NumberG/SeqLength, SeqLength))
```

Now when executing the script, you will have to enter a DNA sequence before Python will give you the final output.

```
host:scripts name$ dna_example.py
Enter a DNA sequence: ATGAAC
Sequence: ATGAAC
Sequence Length: 6.0
A: 0.5
C: 0.16666666666666666
G: 0.16666666666666666
T: 0.16666666666666666
A occurs in 3 bases out of 6.
C occurs in 1 bases out of 6.
G occurs in 1 bases out of 6.
T occurs in 1 bases out of 6.
A occurs in 50.0% of 6 bases.
C occurs in 16.7% of 6 bases.
G occurs in 16.7% of 6 bases.
T occurs in 16.7% of 6 bases.
```

Avoiding data problems with `.upper()` and `.replace()`

It can be extremely beneficial to add safety nets when working with large, complex data sets that may have problems that you can't see. For DNA sequences, it is possible some bases were inserted as lower case instead of upper case. To make sure your script produces accurate outputs,

we can use the method `.upper()`. This will convert the string to all uppercase. We can add this to the script by including `DNASeq = DNASeq.upper()` underneath the line that reads `DNASeq = input("Enter a DNA sequence: ")`. If you need to use the original `DNASeq` variable later in the script, you can rename the variable that includes the `.upper()` method something else. In this case, we just wrote over the original variable.

You can use the `.replace()` method to remove unwanted spaces (you can use this method to replace anything, not only spaces). The spaces will be counted as characters and can change the total DNA sequence length. Similar to `.upper()`, we will rename the variable `DNASeq` and add what we want to find and replace with within the parentheses: `DNASeq = DNASeq.replace(" ", "")`. Inside the parentheses, we first put a single blank space in quotes followed by no space in quotes, which will remove blank spaces within the `DNASeq` string.

To have even less code, you can nest `.replace()` within `.upper()`. Depending on what the script is for, it is often preferred to have as few lines of code as possible, without making it incomprehensible or too complex.

You now have a cleaned up program to execute some basic analyses on a DNA sequence. Your full script should look as follows (or similar):

```
 #! /usr/bin/env python

DNASeq = 'ATGAAC'
DNASeq = input("Enter a DNA sequence: ")
DNASeq = DNASeq.upper().replace(" ", "")
print ('Sequence:', DNASeq)

SeqLength = float(len(DNASeq))
print ('Sequence Length:', SeqLength)

NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')

print ("A:", NumberA/SeqLength)
print ("C:", NumberC/SeqLength)
print ("G:", NumberG/SeqLength)
print ("T:", NumberT/SeqLength)

print ("A occurs in %d bases out of %d." % (NumberA, SeqLength))
print ("C occurs in %d bases out of %d." % (NumberC, SeqLength))
print ("G occurs in %d bases out of %d." % (NumberG, SeqLength))
print ("T occurs in %d bases out of %d." % (NumberT, SeqLength))
```

```
print ("A occurs in %.1f%% of %d bases." % (100 *
     NumberA/SeqLength, SeqLength))
print ("C occurs in %.1f%% of %d bases." % (100 *
     NumberC/SeqLength, SeqLength))
print ("T occurs in %.1f%% of %d bases." % (100 *
     NumberT/SeqLength, SeqLength))
print ("G occurs in %.1f%% of %d bases." % (100 *
     NumberG/SeqLength, SeqLength))
```

And your output should look similar to this, depending on the DNA sequence you entered:

```
host:scripts name$ dna_example.py
Enter a DNA sequence: GATTCAAGTCCACT
Sequence: GATTCAAGTCCACT
Sequence Length: 14.0
A: 0.2857142857142857
C: 0.2857142857142857
G: 0.14285714285714285
T: 0.2857142857142857
A occurs in 4 bases out of 14.
C occurs in 4 bases out of 14.
G occurs in 2 bases out of 14.
T occurs in 4 bases out of 14.
A occurs in 28.6% of 14 bases.
C occurs in 28.6% of 14 bases.
G occurs in 14.3% of 14 bases.
T occurs in 28.6% of 14 bases.
```

References

Haddock S, Dunn C (2010) Practical computing for biologists. Sunderland (Massachusetts): Sinauer Associates.