# Introduction to Python
# Day Three Exercises

Stephanie Spielman
Email: stephanie.spielman@gmail.com

## 1 Defining Functions

For this set of exercises, you will re-write some of yesterday's exercises as functions. The code doing the actual computation will remain virtually the same, except it will be written in the context of a function and subsequently called. **After you write each function, run it with 2-3 test cases to confirm that it works as expected!!**

1. In Texas, you can be a member of the elite "top 1%" if you make at least $423,000 per year. Alternatively, in Hawaii, you can be a member once you start making at least $279,000 per year! Finally, if you live in New York, you need to earn at least $506,000 a year to make the cut.

   Write a function to determine if a given salary is a "top-1%" salary in Texas, Hawaii, and/or New York. Your function should take a single argument, the salary, and *print* a sentence indicating in which state(s) this salary is and is not a top-1% salary. Your function should not return a value.

2. Write a function that returns a list of the powers (exponents 0-15) for a provided number. This function should take 1 argument: the number to raise to powers 0-15.

3. Write a function to curve a list of grades, silly-professor style. This function should take *four arguments*:

   - A list of grades to curve
   - The cutoff *above which* grades are reduced
   - The cutoff *below which* grades are raised
   - The scaling value

   The function should return a list of curved grades.

4. Write a function to compute the molecular weight of a protein sequence. This function should take a single argument, a protein-sequence string, and it should return a single value, the molecular weight. Your function should account for the potential presence of ambiguous amino acids (again, compute these weights from average of all weights). Once your function is written, run it on the protein sequence "ABVPOXIRBTQQWS." Use this dictionary in your function:

```
amino_weights = {"A":89.09, "R":174.20, "N":132.12, "D":133.10, "C":121.15,
"Q":146.15, "E":147.13, "G":75.07, "H":155.16, "I":131.17, "L":131.17,
"K":146.19, "M":149.21, "F":165.19, "P":115.13, "S":105.09, "T":119.12,
"W":204.23, "Y":181.19, "V":117.15}
```

5. Write a function to determine if a sentence is an alliteration. Your function should take two arguments: the sentence and the alliteration letter. Your function should return the value `True` if the sentence is alliteration and `False` otherwise. Importantly, note that you can us an `if` statement to return different values under different conditions!

## 2 Interacting with files

These questions make use of a series of files included in this zip directory.

1. Open the file "flu_sequences.fasta" in read-mode, and save the contents of a file as a single string using the `.read()` method (be sure to close the file, or use `with` control-flow!). Address the following questions using Python:

   (a) How many *characters* are in this file? Remember that `.read()` saves the entire file as a single string!

   (b) How many *lines* are in this file? (Hint: use the `.strip()` method as part of the solution).

   (c) Use a for-loop to print the first 10 lines of the file.

   (d) Open and read the file again, except this time, use the `.readlines()` method instead of `.read()`. Perform the same three tasks. (Hint: to count the characters, you can either loop over the line list, or use the string method `.join()`. This method is the "opposite" of `.split()` – it joins together a list of strings (argument) using a given separator, e.g. `" ".join(["a", "b", "c", "d"])` returns the string `"a b c d"`.)

2. Write a function to return a list of lines in a given file. This function should take one argument, the file name, and return a list of the lines. Use this function as part of your code to perform these tasks on the included files named file1.txt, file2.txt, file3.txt,..., file20.txt (below, they are referred to as file<1-20.txt> for convenience). (Note that the code which generated these files is included in the python scipt `generate_files.py`.)

   (a) Determine the number of lines in each file. Use a for-loop to loop over files. When writing this for-loop, think about the cleanest way to loop over these file names (hint: it is *not* to loop over a list with contents `["file1.txt", "file2.txt", ...]`). Print, for each file, a sentence that reads "file1.txt contains XXX" lines (with file1.txt and XXX replaced accordingly).

   (b) For each file<1-20>.txt, create new files called "file<1-20>_upper.txt" which contain the full contents of each file, except entirely in upper-case. Note that this is a 2-step process for each file: first read in, and then write the new file.

3. Create a new file called "file_full.txt" which contains all contents from each file<1-20>.txt (in order). Perform this task twice using two different approaches:

   (a) Open the "file_full.txt" a single time for writing and write all contents to this handle.

   (b) Using the "append" mode to open and close the file multiple times for writing.

   Hint: think about whether the `.read()` or `.readlines()` method makes the most sense for this task.

4. For this question, you are going to save some useful information to a file. Specifically, you will create a *CSV* (comma-separated values) file. This is basically what Excel files are - files with columns of data. The top row is the header, and each subsequent row contains data. Columns of data are separated by commas in csv files (note that there are other types of *delimited* files, like tab-delimited files).

   You will use the data from the file "dopamine_sequences.fasta," which contains DNA sequences of vertebrate dopamine receptors published in Spielman et al. [2015]. For convenience, the separate file "dopamine_sequences_dictionary.txt" contains a Python dictionary of these sequences; the keys are sequences IDs, and the values are the sequences. You can copy and paste the contents of "dopamine_sequences_dictionary.txt" into the Python script you'll write for this question (tomorrow we will learn how to actually parse the dopamine_sequences.fasta file itself).

Now, create a CSV file with two columns: the sequence ID and the sequence length. Follow this general strategy:

(a) Collect all information to write to the output file (this has been done for you in the dictionary!)

(b) Open the output file and write a header to the file (remember to add `"\n"` for newlines!)

(c) Write each line to the output file. In this case, you will need to loop over the dictionary and write each key:value pair in CSV format with a newline character, e.g. "sequenceID,length\n"

Once you have written the file, open it in a text editor to be sure that it is correct.

Now, modify this code in the following ways:

(a) Modify your CSV-writing code such that it uses a function to write the CSV. This function should take three arguments: the header string, the dictionary to write, and the output file name. The function does not need to return anything.

(b) Modify your code and function so that you now write a file with three columns: sequence ID, sequence length, and percent of ambiguous nucleotides per DNA sequence. In this sequence, the character "N" is considered ambiguous (unlike A, C, G, T).

# References

SJ Spielman, K Kumar, and CO Wilke. Comprehensive, structurally-informed alignment and phylogeny of vertebrate biogenic amine receptors. *PeerJ*, 3:e773, 2015. doi: 10.7717/peerj.773.