Welcome to Day 2 of the Introduction to Python workshop. Today we will go over lists and dictionaries, indexing, writing scripts and running them from the command line, and for-loops.

We'll start be reviewing our function from the end of yesterday.

#%%

Instructions for writing this function. This new function should take a sequence as an input parameter (combinations of A, T, G, C). It will then search that sequence and, if the sequence starts with 'ATC', it should print the T content of the sequence (as we did with the GC content earlier). If the sequence starts with 'AGC', it will print 'Starting with AGC', and the number of times 'G' appears, but only if it does not follow 'A'. If the sequence starts with neither 'ATC' or 'AGC', then it will print 'Starting with neither ATC or AGC'.

```
def startswithATC(seq):
    if seq.startswith("ATC"):
        print(seq.count('T')/len(seq))
    elif seq.startswith("AGC"):
        gcount = seq.count("G") - seq.count("AG")
        print('Starting with AGC', gcount)
    else:
        print("Starting with neither ATC or AGC")
```

#%% ------ INDEXING ------

Indexing is perhaps one of the most important feature of programming (personal opinion). Very often we are interested in extracting individual (or multiple) elements from a string, or a datafile, or a folder. In order to do this, we need to know the "place holders" (or "indeces") of those elements. Let's define a string so we can practice indexing its elements.

```
teststring = 'philadelphia'

#%% To index this string, type the name of the string followed by a set of square brackets, and a numerical value that represents the location of the element along the length of the string.

# Try indexing for the third character in your string.  Which letter do you expect Python to return?

teststring[3]

# Did you get what you expected?

#%% Python starts its counts at 0, not at 1.  You must always keep this in mind when indexing in Python.  And whenever you start working with another language, you should always ask "Is this language 0-indexed?"

# Let's try some more indexing.  Ask Python for the 0th element in your string.

teststring[0]

#%% Ask Python for an index that is larger than the length of the string your defined.

teststring[100]

#%% Negative indexing:  You can also index by negative values.  In Python, indexing by a negative value means you are starting from the end of the string.  The counting is awkward though.  An index of "-1" represents the first element from the end of the string (we can't negate 0).

# Try negative indexing your string by -1

teststring[-1]

#%% Try negative indexing your string by -3
```

```
teststring[-3]

#%% Indexing by Ranges:  You can also index by a continuous range of values.  To do this, we use the color as an operator (":").
# Try indexing your string from the second element to the fourth.  Which letters do you expect to get?

teststring[1:3]

# Which letters did you get back?  What does this mean about the second value in the range?

#%% Indexing by Ranges with Skips:  We can also index by a continuous range of values that skips every other, every 2, every 3, etc. values.  We do this by using a second colon.  So, the index 1:8:2 would represent, "index from the second element to the ninth, but return every second entry" or more specifically "return the second, fourth, sixth and eighth elements".  Try this command out on your string.

teststring[1:8:2]



#%% Now, let's talk about some different ways to run Python code.
# (1)  The interactive environment.  This is basically what we have been doing, where we are executing code using the console.  This can also be done through the Command Line (Windows) or Terminal (Mac), and we will experiment with that approach on Day 3.
# (2)  Scripts.  This is where we have a dedicated Python file (with suffix .py) that contains the code we want to run and may or may not require any input parameters.  Scripts are run using the Console by calling their name, or they may be called upon by other scripts.  In this way, scripts are similar to software.  Note that the file/tab we are currently writing in has a .py suffix.  However, we are not running executing this file as a script, but instead are executing chunks of it as a time through the Console.
```

```
# (3) Modules. These are Python (.py) files that contain one or more functions which we can import into the interactive environment. We can import modules in several ways.
#  (a) We can import the module using only its name, but this requires we precede the call to every function in the module with the name of the module.
#  (b) We can import only one function from the module.
#  (c) We can import every function from the module.
#  Options (b) and (c) mean that we no longer need to precede the function names with the of the module from which they came. However, option (c) could be dangerous if the module is very large. Examples are below.

# We will practice importing from the math module.
# (a)

import math


#%% Now we can use fancy math functions like
# math.sqrt()

math.sqrt(25)

#%% math.exp()

math.exp(2)

#%% math.log()

math.log10(10)


#%% math.pi

math.pi
```

```
#%% Option (b)  Now instead we will import just the square root function.

from math import sqrt

#%% Now we should be able to use the sqrt() function without the 'math.' preceding it.

sqrt(25)

#%% Option (c)  Now instead we will import everying from the math module.

from math import *

#%% Now we should be able to use anything from the math module without the 'math.'
preceding it.

pi



#%% ------ SCRIPTS ------
# We're now going to begin working with scripts, and running them from either your command
line (windows) or terminal (macs).  First, let's open the command line/terminal to quickly
familiarize ourselves.  From the command line/terminal, we can open an interactive session of
Python by typing 'python' (for windows) or 'python3' (for macs).  When doing this you should see
that the blinking cursor is now preceded by three greater than symbols.  This mean that you are
in an interactive python session.  That is similar to what we are doing now, where our code is
executed within the console window of the Spyder program.  Note, all of the online tutorial
material is designed such that you are running python through the command line/terminal.

# Now, we are going to exit the command line/terminal python session by exectuing 'exit()', and
will instead run a python script from the normal command line/terminal environment.

# Next, we are going to write our first script by converting our 'startswithATC' function into a
separate script.  So, please open a new file within Spyder.
```

Once we have written our script we must save it to a convenient location (your desktop) we can easily navigate to using the terminal (macs) or command line (windows). After having saved the file, open the terminal or command line as instructed, and run your script by typing: python startswithATC.py AGCAGGAGGAG

Now, for more practice scripting, we are going to wrie a script that calculates the tip and tax on top of a restaurant bill. The two inputs are going to be the total cost of dinner and the amount by which you want to tip.

#%% ------ True/False Conditionals ------

Often times when programming we need to use what are called 'conditional statements', or 'conditional operations'. These are logical comparisons of equivalence, or truth statements. The operators we use for this are the 'in' and 'not in' statements from before, as well as: double equals == for "are these two things the exact same?", not equals != for "are these two things not the exact same?", less than <, greater than >, and less than or equal to <=, and greater than or equal to >=. Python will return True or False depending on whether the condition (statement) is True or False. Let's practice.

Ask Python if 2 is exactly equal to 3.

```
2 == 3
```

#%% Ask Python if 2 is not equal to 3.

```
2 != 3
```

#%% and: Sometimes, we will want to compare multiple conditional statements. For example, is 'A' in the string 'ATGTAGC' as well is the length of the string 'ATGTAGC' equal to 7? We can

do this with the use of the 'and' command typed sequentially between both conditional statements. Try asking Python to check both of these conditions.

```
'A' in 'ATGTAGC' and len('ATGTAGC') == 7
```

#%% What if one of the statements were False, but the other were True? Try changing the length condition to 10.

```
'A' in 'ATGTAGC' and len('ATGTAGC') == 10
```

#%% What if both of the statements are False?

```
'Q' in 'ATGTAGC' and len('ATGTAGC') == 10
```

#%% or: Sometimes you may not want the strictness of the 'and' statement, but willing to except cases where either of the conditions is True. In these cases, we use the 'or' statement. Try again, but now askig if 'A' is in the the string or if the length is 10.

```
'A' in 'ATGTAGC' or len('ATGTAGC') == 10
```

#%% What about if 'Q' is in the statement or the length is 7?

```
'Q' in 'ATGTAGC' and len('ATGTAGC') == 7
```

#%% What if both are False?

```
'Q' in 'ATGTAGC' or len('ATGTAGC') == 10
```

We can organize these features of 'and' and 'or' in what are called 'Truth' tables.

#%% ------ LOOPS ------

```
# Sometimes we need to perform the same task on every item in a list, string, or dictionary.  In these instances, we use loops.  We will focus on learning "for" loops.

# The general structure of a for loop is the following
# for index_variable in range(start_integer, stop_integer):
#       code that is executed with loop

# Above we used the range() function.  When used as 'range(start_integer, stop_integer) The range() function defines a list of integers from start_integer to stop_integer - 1.  Note that python does not normally display the whole list, to do so you must run range() inside of list().  Thus, the index_variable takes on the first value from range(), and executes the code within the loop where the index_variable can be used to access different elements of lists within the loop.  Lets try running range(0,20) and list(range(0,20))


#%% Loops on a list
# Let's define a list of numbers, and loop through that list and multiply each value by 2.

numbersList = [1,2,3,4,5,6,7,8,9,10]

for i in range(0,len(numbersList)):
    print(i, numbersList[i], 2*numbersList[i])


#%% Now, if we use a third entry in the range() function, then the returned list skips integers by that value.

for i in range(0, len(numbersList), 2):
    print(i, numbersList[i], 2*numbersList[i])

#%% We can also let the index take on the actual value of the elements within a list instead of taking on indexing values.  This use can be beneficial if you only need to index a single list, dictionary, or string.  We'll demonstrate its use with our experiments dictionary from before.
```

```
for i in numbersList:
    print(i, 2*i)

#%% As well as with a string.

string = "cheese"
for i in string:
    print(10*i)


#%% Finally, you may want to save operations done during a loop to another object.  To do this, it is best that we "initiliaze" an empty object where we can save our work.  For example, suppose we want to loop through the letters in philadelphia and save only the  "p's" to a new string.  To do so, we will make an empty string first, then make our loop with an if statement to check if the letter is a 'p'.

new_string = ''

for i in teststring:
    if i == 'p':
        new_string = new_string + i.upper()

print(new_string)


#%% Next we will practice using loops inside of a script to determine the reverse complement of a gene sequence.  So, let's switch to a new script.py file titled 'revcomp_student.py'.
```