

```
### ----- Day 3/3 -----
```

```
# Welcome to Day 3 of the Introduction to Python workshop. Today we will go over lists, while loops, error handling, modules, and Jupyter.
```

```
### ----- LISTS -----
```

```
# Lists are another object type in Python. You can think of them as one-dimensional vectors, but each element in the vector can be of a different type, even itself another list. To define a list, we use the square brackets.
```

```
myEmptyList = []
```

```
###
```

```
myShoppingList = ['Apples', 'Cheese', 'Chocolate']
```

```
###
```

```
myMixedList = [123, 'a string', 2.75]
```

```
###
```

```
myListofLists = [myShoppingList, 1, 2, [1, 2, 3, 4]]
```

```
### We can access elements of a list by indexing, just as we did with strings. In fact, strings are essentially a list, just with all elements of the same type, and with some Python interpretation happening 'under the hood'. Let's make a new list of nucleotides and practice indexing.
```

```
myNuclist = ['A', 'G', 'C', 'T']
```

```
### Now index list
```

```
myNuclist[0]
```

Note that we can change elements of the list using assignment.

```
myNuclist[2] = 'G'
```

```
print(myNuclist)
```

----- LIST METHODS -----

There are many functions (called methods) that are designed for manipulating lists. To see a list of them, run 'dir()', with the name of a list as the argument.

```
dir(myNuclist)
```

append: The append method can be used to add elements to the end of the list.

```
myNuclist.append('C')
```

index: The index method can be used to return the list index corresponding to the element provided. Again, this method only identifies the first instance of the matching element.

```
myNuclist.index('G')
```

sort: The sort method can be used to sort the elements of the list. Some assumptions are made regarding the correct ordering.

```
myNuclist.sort()
```

```
myNuclist
```

remove: The remove method can be used to remove elements from the list, as identified by their value (not by their index). Again, for repeated elements, this only removes the first instance of the element.

```
myNuclist.remove('G')
```

myNuclist

pop: The pop method can be used to remove elements from the list, as identified by their index.

myNuclist.pop(1)

len: As before, len is not a method (it is not called by listname.len()) but instead is an inherent function in Python called by len(listname). len will tell us the number of elements in the list.

len(myNuclist)

###---- Reverse Complement Version 2----

Now, let's rework our reverse complement script to use lists and list methods instead of string and string methods. Open the script for revCompv2_student.py.

----- DICTIONARIES -----

Dictionaries are another object type in Python. And they happen to be specific to Python (they do not appear in other languages). Dictionaries are similar to lists, but instead of having indices and values, they have 'keys' and values. Keys can be strings or numbers, and serve as unique identifiers for their paired value. Like lists, the values can be anything (e.g. strings, lists, or even another dictionary).

Let's define a short dictionary and practice indexing it by its keys. To define a dictionary, we use curly brackets instead of square brackets or parentheses.

```
experiments = {'control':1000, 'exp1':500, 'exp2':600}
```

```
### Call the name of the dictionary to view its elements
```

```
experiments
```

```
### Index the dictionary by any one of the keys.
```

```
experiments['exp2']
```

```
### Add a new key:value pair to the dictionary
```

```
experiments['exp3'] = 300
```

```
### Call the name of the dictionary again to view its elements.
```

```
experiments
```

```
### We won't use dictionaries much more in the future as they are a rather special use object.  
However, we can demonstrate their utility quickly for our reverse complement scripts.
```

```
###----- Reverse Complement Version 3-----
```

```
def revcomp3(seq):
```

```
    # define empty string to save output
```

```
    rc_seq = "
```

```
    # define complement dictionary
```

```
    compDict = {'A':'T', 'T':'A', 'C':'G', 'G':'C'}
```

```
    # loop through input sequence using compDict to perform the switch and string addition to  
merge
```

```
    for nuc in seq:
```

```
        rc_seq = compDict[nuc] + rc_seq
```

```
return rc_seq
```

```
###
```

```
revcomp3('AAAAGGGG')
```

```
### ----- MODULES -----
```

Running your custom module can be tricky if the file is not located in a commonly searched directory. You can see where Python searches (and modify the locations) by examining the `sys.path` list from the 'sys' module.

```
import sys
```

```
sys.path
```

```
### I want to add my desktop to the list of commonly searched paths.
```

```
sys.path.append('/Users/alexwork/Desktop')
```

Now we can load our functions written as parts of larger modules into our active IPython sessions. For example, we have the stats module. This is a .py script full of many functions that "we wrote" for performing statistical analyses. As written, we can not simply run this script from the command line like we did with our tip calculators or first reverse complements. Instead we must load this module of functions into an active Python session. We do this using the `from` and `import` statements as before.

```
from stats_module import *
```

```
### Now we should be able to call any of the functions from the stats_module.
```

```
### ----- WHILE LOOPS -----
```

In certain scenarios where iteration is required, you may want a loop to run during a duration of time while a particular condition is held. We have just described the utility of a while loop. With these loops, as long as the given condition (or compounded conditions) is True, then the code contained within the body of the while loop will be executed. The general structure of a while loop follows:

```
# while logical_condition_is == True:
```

```
#     all code here is executed
```

```
# Let's look at some simple examples of while loops.
```

```
### In this example we will first define a variable that represents our condition.
```

```
example_condition = True
```

```
### Then our while loop will be contingent on this condition. Inside of our while loop, we will redefine the value of the example_codition variable to be False, and provide a print command stating print("This will only print once")
```

```
while example_condition:
```

```
    example_condition = False
```

```
    print("this will only print once")
```

```
### In this next example, we'll define a counting variable and make the condition be on an inequality related to that value.
```

```
count = 0
while count < 40:
    print(count)
    count += 1
```

```
# How can we get the final number to print?
# What happens if we accidentally generate an infinite loop?
# Can we have while, for, if, else, elif all in one statement/function?
```

```
### ----- Error Handling -----
```

```
# When programming, you will make errors. It is a normal part of writing code. Knowing how to interpret and handle errors is an important part of programming.
```

```
# Broadly speaking, there are two types of errors that you will run into with Python. Errors that occur in the interpretation of the code (such as syntax errors or indentation errors) and errors that occur during the execution of the code (these are also called exceptions, and examples are value errors, name errors, divide by zero errors, etc.).
```

```
# Some examples of interpretation errors are as follows:
```

```
### Syntax error
```

```
statement = True
```

```
while statement
    print('This will print only once')
    statement = False
```

```
# We need to add a colon.
```

```
### Indentation error
```

```
statement = True
```

```
while statement:
```

```
print('This will print only once')
```

```
statement = False
```

```
# We need to indent. Python helpfully tells us the first instance in which the indentation error occurs.
```

```
### Some examples of execution errors are as follows:
```

```
### Name error
```

```
someText += 2
```

```
# We need to define the variable someText.
```

```
### Value error
```

```
int('aaa')
```

```
# We can not coerce something that is inherently text into an integer.
```

```
### Divide by zero error
```

```
1/0
```

```
# We can not divide by zero.
```

```
### Error handling
```

```
# If we expect to receive some errors as we are looping through a set of data, then we can handle these anticipated errors using the try/except functions.
```


Suppose we have a list of numbers that we are going to use to divide into the number 10.
This list of numbers is the following...

```
numlist = [1, 2, 3, 4, 5, 'abc', 6, 7, 0, 9, 10]
```

If we try to loop through this list and divide 10 by the value of each number in this list,
there will be an error. Let's try...

```
for i in range(1,len(numlist)):  
    print(10/numlist[i])
```

We have a TypeError when dividing by 'abc'. Lets try/except for TypeErrors.

```
for i in range(0,len(numlist)):  
    try:  
        print(10/numlist[i])  
    except TypeError:  
        print("%s is not a number" % numlist[i])
```

Our first problem is solved! But not we have a new error, division by zero. That is also
OK, we can simply add another exceptiong.

```
for i in range(0,len(numlist)):  
    try:  
        print(10/numlist[i])  
    except TypeError:  
        print("%s is not a number" % numlist[i])  
    except ZeroDivisionError:  
        print("Can not divide by zero")
```

Sometimes you will get an error message and have no idea what to do. In such instances
we must search the internet. The chances that someone ran into a similar (if not exact) error
while trying to do something similar to you (or exactly the same as) are very high. I generally do

a google search of the exact error output from Python, and then look at results from the StackOverflow website (although sometimes solutions will come from other sites).

File input and output. A very common practice in programming is the need to open and examine the contents of a file and/or write our own output to a new file (or append new output to an existing file). We will go over a few quick examples of doing this for text files (.txt).

Writing to a new file with the open() function and "w" parameter.

```
file = open("filename.txt", "w")
```

The open() function with the "w" parameter tells Python to make a new .txt file with the name "filename.txt".

Let's loop through our previous numlist and print the results of the division to our .txt file.

However, instead of printing to the console, we want to write to the file. Thus, we will replace the print commands with file.write() commands.

```
for i in range(0, len(numlist)):
```

```
    try:
```

```
        file.write(str(10/numlist[i]) + '\n') # here we coerce the output of the operation into a string, then append '\n' to the end of the string so that a line is skipped once the string is written.
```

```
    except TypeError:
```

```
        file.write("%s is not a number" % numlist[i] + '\n')
```

```
    except ZeroDivisionError:
```

```
        file.write("Can not divide by zero" + '\n')
```

```
file.close()
```

Reading a file.

To read a file we use the same open() function, but now with the parameter "r" for read instead of "w" for write.

```
file = open("filename.txt", "r")
```

We can now inspect this file using the read methods associated with the file object. file.readline() will print one line at a time, and sequentially. file.read() will print all remaining/unread lines.

```
print(file.read())
```

```
###
```

```
file = open("filename.txt", "r")
```

```
###
```

```
print(file.readline())
```

If we want to interact with the contents of the file then we can assign them to a list in python using a for loop.

```
file = open("filename.txt", "r")
```

```
filelist = []
```

```
for i in file:
```

```
    filelist.append(i)
```

```
file.close() # Don't forget to finally close the file once you are all done interacting with it.
```

Appending to an existing file. Suppose you don't want to overwrite an existing file, but only add new contents to it. This works just like the write function, only we change the "w" argument to "a", for append.

```
file = open("filename.txt", "a")
```

```
file.write("one more line")
```

```
file.close()
```

```
### ----- Jupyter -----
```

```
# Jupyter is a web-based integrated development environment. To access it, we need to open the command line/terminal, and type 'jupyter notebook', or open it from the Anaconda Navigator. In your default browser, a new tab should open with a directory of folders and files for your primary computer user. Here you can either create new jupyter notebooks, or open ones you have used before.
```

```
# %% Now, pick one of the following end of day projects to work on in our remaining time (or try both as time permits).
```

```
### ----- Day 3 END PROJECT OF THE DAY -----
```

```
# For the final activity: Use what we have learned today to read into jupyter the second file of data I have provided you. You will need to add some code to remove the '\n' terms from the end of each line and to coerce everything to a float. Do you anticipate there being any kinds of errors during coercion? Try writing some try/except statements to handle these potential errors. Finally, once you are done, use the varianceNums function from the stats_module to calculate the variance in the list of numbers you have read into jupyter, and then appended the line of text "The variance is [variance]", having rounded to four decimal places, to the end of the datafile.
```