Week 2

Create a folder for this week.
Open IDLE and open a file and safe it to this week's folder. This file will be for trying things out. You can name it something like "week2practice.py"

Adding comments quickly in the IDLE editor

You can block out code simply by turning it into comments
In Windows (not sure in MacOS):

Highlight the code, and use the Alt + 3 key
This adds a double #
Then you can remove this using the Alt + 4 key

The builtins module

When python loads (e.g. start python at the command line, or you open IDLE) the builtins module is automatically loaded.

```
dir (__builtins__)
```

These are the things that are available when Idle starts. You do not need to import the __builtins__ module because it happens automatically.

Look at some of the things in __builtins__ using `dir()` and `help()`

Working with text files

Objects of type "file" can be created using the `open()` function (that comes in __builtins__.

Read about the `open()` function.

```
help(open)
```

Try creating a file:

```
a = open("test.txt","w")
type (a)
dir (a)
a.write("hello file world")
a.close()
```

Now go look at this file (e.g. open it in a text editor)

Download the file Dromel_Adh.fasta from Canvas (under week2) to your folder .
Open the file using "r" mode:

```
fname = "Dromel_Adh.fasta"
adhfile = open(fname,"r")
```

Read the lines into a list using the readlines() function (which belongs to things of type file). Then close the file:

```
linelist = adhfile.readlines()
adhfile.close()
```

Look at linelist:

```
type(linelist)
len(linelist)
linelist[0]
linelist[1]
```


Controlling the flow of a program, aka 'Flow Control'

A program can be just a simple list of statements which are executed in succession.

But for many programs you will want to repeat some tasks or decide which tasks to execute.

Python, like virtually all programming languages, has a set of special statements for controlling program flow.

Python uses a colon ":" after a flow control statement

After a colon all of the statements that belong to that flow control statement are indented (use the tab key).

You can have multiple flow control statements, nested within one another, just remember that every time you put in a flow control statement, the statements that belong to that get indented.

Conditional statements, using keywords if and else

```
a=1
if (a==2):
    print (" a equals 2")
else :
    print (" a does not equal 2")
```

while loops – statements that cause sections of code to be repeated as long as something is True

while loops use a the keyword while and a boolean value or expression (something that evaluates to True or False). So long as the conditional expression evaluates to True, the statements below the while will be executed


```
a = 0
while a < 5 :
    print (a)
    a = a + 1
```

Often it is useful to use a break statement inside a while loop. How does this example differ from the previous one?

```
a = 0
while True:
   if a>=5:
       break
   print (a)
    a = a + 1
```

Iterator, a special kind of object (of type iterator, or list iterator) that can be created in association with a list or a string

Iterators are used to move through lists or strings one item at a time in order from beginning to end.

```
a = [1,2,"hello", ["x","y"],3e-9]
ai = iter(a)
next(ai)
```

try it again

```
next(ai)
```

Make a list from what the iterator has not yet returned:

```
ail = list(ai)
print(ail)
```

For loops – statements that repeat sections of code using an iterator

for loops work on lists or strings. They use the keywords `for` and `in`.
The variable following the `for` becomes an iterator for the list or string following the word `in`.

In a `for` loop there is no need to use the `next()` function as it gets called automatically and the code inside in the `for` loop is repeated, for each item in the list or string in order from beginning to end.

If you already have a list you can go through each element of the list. In this example `cpart` becomes the iterator associated with `c`

```
c=["dog","cat","fish","duck","butterfly"]
for cpart in c:
    print (cpart)
```

The same thing works with strings:

```
    for char in "abcdef":
        print (char)
```

A useful trick – if you want to loop thru a series of integers but you do not have a list of those integers handy, you can create an iterator that works list a list on the fly using the `range()` function.

```
    for i in range(0,5):
        print (i)
```

Functions - pieces of code that form their own separate program within a program .

Functions within a program can be called and executed from other parts of the program

Functions use the keyword def and often use the keyword return.
The first line of a function should be a string that begins and ends in triple quotes (`"""`). This string can then be used by the help function to return information about the function


```
def myprint (toprint) :
    """simple print function example"""
    print "now printing: ",toprint

 myprint(“hello”)

def plusone (addto) :
    """ simple addition function example"""
    return addto + 1

a= plusone(3.5)
```


Review calc_mean_var.py

Download the file calc_mean_var.py. This is a simple program that demonstrates a number of flow control examples.

Let x be a list of variable. We have a file of a list of n values for x.

Then the mean is

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

And the variance, s^2 is

$$s_x^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}$$

This can be easily calculated by accumulating the sum and sum of squares of x and calculating

$$s_x^2 = \frac{\sum_{i=1}^{n} {x_i}^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}$$

Review primefactor.py

Download the file primefactor.py. This is a simple program that demonstrates a number of flow control examples.

Assignment:

There are 3 parts. Each part involves writing a program that reads a file and writes a file. For each part turn in the program and the file written when you ran the program (total of 6 files).

Part 1.

Suppose that you measure two variables on each of several individuals, e.g. x= height and y=weight. The correlation for x and y using the formula for the Product Moment correlation

$$\rho = \frac{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x}) * (y_i - \bar{y})}{s_x s_y}$$

Here n is the number of individuals and s_x and s_y are the standard deviations for x and y respectively. A version of this formula that is easier to calculate is

$$\rho = \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)\left(\sum_{i=1}^{n} y_i\right)}{\sqrt{\left(\sum_{i=1}^{n} {x_i}^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n-1}\right)\left(\sum_{i=1}^{n} {y_i}^2 - \frac{\left(\sum_{i=1}^{n} y_i\right)^2}{n-1}\right)}}$$

Write a program that:

- Asks the user for the name of a file containing two columns of numbers
- Reads the file with two columns of numbers
- calculates the product moment correlation
- Writes the value of the correlation to a file named "pmc_calc.txt"
- Be sure to include explanatory comments in your program
- Do not import a module that calculates the coorelation

Download the file paired_number_list.txt from Canvas. This contains 1000 pairs of numbers. Run your program on this file.

- Turn in your program file and the file pmc_calc.txt

Part 2.

Working with matrices. A matrix is a two dimensional array such as this table of numbers

0.578789	0.709068	0.133332
0.256684	0.217213	0.918261
0.059334	0.349801	0.208787
0.709623	0.751263	0.614539
0.692426	0.688036	0.309793
0.736131	0.861346	0.455957
0.844321	0.120789	0.836109

This example has 7 rows in one dimension and 3 columns in the other dimension.

Matrices are used in a great deal of any computational science. They are the fundamental unit of linear algebra, and all programmers need to be good at working with them.

In python we can make a matrix by working with a list of lists, each of which has numbers in it.
For example, the matrix above could be represented in python by

```
a = [[0.578789,0.709068,0.133332],
     [0.256684,0.217213,0.918261],
     [0.059334,0.349801,0.208787],
     [0.709623,0.751263,0.614539],
     [0.692426,0.688036,0.309793],
     [0.736131,0.861346,0.455957],
     [0.844321,0.120789,0.836109]]
```

We access an individual element by using dual bracket notation, e.g. a[1][2]
The first bracket access the position of a list within the list of lists, and the second bracket accesses the position within that list. You can think of the lists as the rows and the positions in the lists as columns.
The transpose of this matrix would be

```
atranspose =
[[0.578,0.256,0.059,0.709,0.692,0.736,0.844],
```



```
[0.709,0.217,0.349,0.751,0.688,0.861,0.120],
[0.133,0.918,0.208,0.614,0.309,0.455,0.836]]
```

Write a program that

- Asks the user for the name of a file containing two columns of numbers
- Reads the file with two columns of numbers into a list of lists, where each row of the file becomes are row (list) in the list of lists that is the matrix
- Create a transpose of this matrix
- Write this new matrix to a file named "matrix_transpose.txt"
- Run the program on the file named paired_number_list.txt
- Turn in your program and the file named "matrix_transpose.txt"

Part 3

Download to this week's folder the file Dromel_Adh.fasta

Write a python program that
- opens Dromel_Adh.fasta and loads the contents (e.g. using readline() or readlines())
- count how many times the sequence GCAA occurs in the file
- print this number to the screen
- determines the reverse complement of the DNA sequence in the file
    - reverse the sequence
    - replace every G by C, every C by G, every T by A and every A by T
- count how many times the sequence GCAA occurs in the reverse complement sequence
- print this number to the screen
- write a new file with name "reverse_complement.txt" that contains only the reverse complement sequence.
- do not import a module that calculates the reverse complement.

Turn in your program and the file reverse_complement.txt

Some additional tips and pointers:

1) Your programs need to run when the instructor tries to run it. Make sure it does not contain any reference to specific folders or modules or other things that are unique to your machine.

2) be sure to understand how `readlines()` works if you use it:
- it creates a list of strings.
- Each one ends in a newline character `("\n")`

3) For part 3, if your sequence is in multiple strings and you search each string for GCAA you will miss those times when GCAA occurs partly at the end of one string and partly at the beginning of the next string.

4) For part 3, if you replace G by C then C by G you will end up with all of the original G's and C's being G's. You need to do the first replacement to an intermediate value that does not occur in the sequence, then the second replacement, then a final third replacement that changes the intermediate value to the final value. e.g. G to X, C to G, X to C.

5) For part 3, if counting GCAA requires more than one line of code you should write a function to do it so you don't have to repeat the code when you count occurrences in the reverse complement.

6) You can nest flow control statements:
for example

```
if var == 'A' :
    var = 'X':
else :
    if var == 'T' :
        var = 'Y':
    else :
        and so on ...
```

for another example, you can nest different kinds of flow control statements

```
for  var in mylist:
```



```
if var == 'A' :
    var = 'X':
else :
    if var == 'T' :
        var = 'Y':
    else :
        and so on ...
```

7) There are several ways to reverse a string, but all are a little bit tricky. Try googling for "python "reverse a string" " and see if you find any useful advice.

One approach is to make a list out of the string and then reverse the list and make a string out of the reversed list. This seems like a lot of work, but lists have a `reverse()` function, whereas strings do not.

If you want to make a list out of the characters in a string, you can use the `list()` function. e.g. `mylist = list("abc")`

If you want to make one long string by joining the strings in a list you can use the `join()` method. Suprisingly `join()` actually belongs to strings and not to lists. It works by taking the string you want to use as the separater between the strings you want to join. You pass the list to `join()`
e.g. `joinedupstring = "".join(mylist)`

8) You cannot change the value of a part of a string. If you want to do this it is usually best to just make a new string.
for example

```
mystring = "ABC"
newstring = ""
for char in mystring :
    if  char == "B" :
        newstring = newstring + "X"
    else :
        newstring = newstring + char

mylist = ["A","B",C"]
newlist = []
```



```
for val in mylist :
    if  val == "B" :
        newlist.append("X")
    else :
        newlist.append(val)
```

9) Don't forget to close any open files using the close() method.

