Lab #2: Loops and Lists

Purpose

The purpose of this lab is to write Python programs that have loops and lists and input/output.

Tasks

In this lab we will write and run Python programs that repeat the same operations many times. The programs we will use are in a compressed file called `lab2.zip` or `lab2.tar.xz` – we will have to upload it to replit.com. Another option is to use a Python editor on your computer.

Success

To succeed in this lab, we will have to pay close attention to the exact syntax of Python (as in Lab #1) and also how loops allow us to carry out the same operations on different data.

Introduction

The approach we will use for learning Python will be to: first, look at the **mechanics** of writing and running programs so that we can create our own programs and, second, see how programs change their behavior as they **modified**. To be sure there will be steps that you may not understand fully but, for now, what is more important is that you are able to carry out the steps.

Basic while loop

1. Let's start with a program with a loop. Loops allow us to repeat actions.

   When writing programs with loops we will often encounter statements of the form:

   ```
   x = x + 1
   ```

   Remember that the **=** operator is the assignment operator and is used to assign the value of the right hand side to the variable on the left hand side of the "=".
   The above Python statement is **not** saying that **x** is some how equal to **x+1**. Instead, we are adding 1 to the current value of 1. If we had an earlier statement that set the value of **x** to be 7, then, what the

   ```
   x = x + 1
   ```

   statement does is to take the current value of 7 in **x**, add 1 to it and store the resulting value of 8 into **x**. This operation can be done again – just use the statement

   ```
   x = x + 1
   ```

   again to increase the value in **x** to 9 and so on.

2. The following program uses this idea to repeatedly increment the value of a variable until it reaches a certain value:

   ```
   #
   # Python program with a loop
   #
   target = 12
   x = 1
   while (x < target):
       print( x )
       x = x + 1
   ```

   As long as the condition **x** < **target** is true, do the indented statement(s).

   The while statement has the form: **while (** *condition* **):** *do things* . Remember that as with the

`if` statement, indentation of the line(s) that belong to the "loop body" is critical: use **4 spaces** to indicate which statements belong in the loop body.
The while statement looks similar to an `if` statement but the meaning of a while loop is that as long as the condition is true, do all the things in the "loop body", then check the condition again and if the condition is true, do the loop body again, check the condition, and so on until the condition is false. i.e. while the "condition" is true, do things in the loop body.
Modify the above program so that it prints the values 1 through 12.

A short note on printing:
Note that the `print` statement in the above program is similar to the prints in the previous lab. We can print a more detailed message using any of the following equivalent prints:

```
print ( 'x is ',  x )
print ( "x is %d" % x )
```

Try each of these in the above program.

If we are trying to print an arithmetic expression, do the arithmetic operation outside quotes. Try putting the following code into a program and see which statement "works":

```
x = 10
print ( "x + 3 is x+3" )
print ( "x + 3 is ", x+3 )
```

Getting input from the user

1. The following program uses the built-in `input` function to get input from the keyboard:

```
#
# Program to get input from the user via the keyboard
#
s = input( "Enter a number: " )
print ( " You entered: ", s )
```

The `input` function gets a line of input from the user. If we enter more than one number separated by spaces on a line, we will see that the numbers are actually treated as a single "string" of characters. Try it: run the program and enter a sequence of two or more numbers separated by spaces.
This means that our program is not actually reading our input as a number. We could enter anything – a number, a name, anything we want. Try the program with various inputs: numbers, text (like "hello", etc.), empty lines, etc.

In order to take an input from a user and use it as a number, we will have to convert the input string into, for example, an integer:

```
s = input( "Enter a number: " )
x = int(s)
print ( " You entered: ", x )
```

This will work as long as we only enter one number – try running the program with more than 1 number: the program will crash. For now, we will assume that the user will enter the correct input but remember that "incorrect" input from the user can crash our program.

The next program will try to add two numbers entered by someone using our program:

```
s = input( "Enter a number: " )
x = int(s)
s = input( "Enter another number: " )
y = int(s)
sum = x + y
print ( "The result of adding these 2 numbers is ", sum )
```

Run this program and make sure that the correct sum is printed when you enter numbers. What happens if we enter strings like “Hello” and “there”?

1

Write a Python program that will ask the user to enter a positive integer and then use a while loop to count from 1 to the number the user entered. For example, if the user runs this program and enters 6, our program should print:

```
1 2 3 4 5 6
```

Getting input from a file

1. Once we “open” a file for reading, we can use the `readline` function associated with the open file object to get input from the open file. The following program shows how to open, read from and close a file:

```
#
# Program to read two numbers from a file
#
# The name of the file as a string:
filename = "numbers.txt"
#
# Use the above string to open the file in "read" mode
inpf = open( filename, 'r' )
#
# Now that the file is open we can read from it
s = inpf.readline()
x = int( s )
# Or combine the above two steps into one step:
y = int( inpf.readline() )
#
# Now that we are done reading from the file, close it.
inpf.close()
#
# Compute results and print:
sum = x + y
print ( "Sum of the 2 numbers from ", filename, " is ", sum )
```

   Before we run the above program, use the Text Editor program to make sure there is a file named “`numbers.txt`” containing 2 numbers in the same directory as the above program – the lab2 directory if that's where our program is. If the file is not present use the Text Editor to create a file named “`numbers.txt`” with 2 numbers, one on each line.

2. Suppose we want to read in a file that may contain many numbers and we want to add up the numbers but we don't know how many numbers the file has. We cannot set aside a separate variable for each number because we do not know how many numbers the file will have. However, we can write a loop like:

```
sum = 0                 # initialize the sum to 0
s = inpf.readline()     # try to read a line from the file
```

```
while ( ... ):
    sum = sum + int( s  )     # add the number to a running total
    s = inpf.readline()       # read the next number
```

This kind of a structure can work but how will the loop stop? It turns out that after we have read the last number in the file, and we try to read another one, we will get what is called an **empty string**, denoted: `""` or `''` – a pair of double-quotes or single-quotes with nothing between the quotes. Therefore we can write the loop condition as:

```
while ( s !=  ''):
  ...
```

2 Modify the above program which reads 2 numbers to read as many numbers as there are in the file. To test your program, add values to the "`numbers.txt`" file and run your program. How can we compute the average of all the numbers in a file if we do not know in advance how many numbers the file has?

Storing output in a file rather than printing to the screen

1. Storing output in a file only turns out to be a matter of opening a file for writing and then using a more general version of the print statement to write to the file instead of the screen and, finally, closing the file to make sure that everything does actually get written to the file. The following program accomplishes these three key tasks to print to a file.

```
#
# Program to write output to a file
#
# The name of the file as a string:
filename = "output.txt"
#
# Use the above string to open the file for writing.
outf = open( filename, "w" )
#
# Now that the file is open we can write to it
count = 0
while ( count < 10 ):
    count = count + 1
    s = str( count )        # convert the number to a
string to write
    outf.write( s + "\n" )  # add a new line after each
number
#
# Now that we are done writing, we should close the file.
outf.close()
#
# Print to screen:
print ( "Done writing to ", filename )
```

Check this program by running it and looking at contents of the output.txt file.

3 Write a program that will read all the numbers contained in one file, add them all up, and print all the numbers, the sum, and the average into a second file. Be sure to use loops.

Lists – also referred to as “arrays”

1. So far our variables have all been “scalars” -- each variable refers to a single number or string. There are many instances where it is convenient to treat many numbers, say in a column of a table, as a single unit in order to perform the same operation on all every row. Python uses lists, also called “arrays”, to store a collection of things such as numbers or strings. Python uses a slightly different syntax for variables that are array or list variables:

```
rnas = [ 'A', 'U', 'G', 'C' ]
numbers = [ 84, 2, 5, -2, 35, 1 ]
```

Note that list names look the same as other variables but we can assign a number of values to a list. Once a list has a number of things in it, we can refer to individual values in it using “indexing”. For a number of reasons, lists in Python start at position 0; in other words, the 1st item in a list is said to be at position 0, the second item is at position 1, and so on. For the above **rnas** list, we use **rnas[0]** to refer to the 1st scalar item in the list – the **'A'**, and **rnas[1]** to access the 2nd item in the list, and so on.

2. Suppose we want to read all the numbers in a file, calculate the average, and then for each of the numbers, determine whether the number is above or below the average. To do this, we can read in the numbers into a list, use a loop to compute the average, and then another loop in which we compare each number in the list with the average to determine whether it is above or below. Most of this is done in the next program:

```
#  Program to get input from a file into a list
# The name of the file as a string:
infilename = "numbers.txt"
# Use the above string to open the file.
inpf = open( infilename, "r" )
# Now that the file is open we can read from it
listofnumbers = []
s = inpf.readline()
while ( s != '' ):
   # the “append” adds an item to the end of the list
   listofnumbers.append( int(s) )
   s = inpf.readline()
# Now that we are done reading, close the file.
inpf.close()
# Compute results and print:
size = len( listofnumbers )
print ( "Done reading: there were ", size, " numbers" )
sum  = 0
count = 0
while ( count < size ):
   sum = sum + listofnumbers[count]
   count = count + 1
#
average = sum / size
print ( "sum of numbers from ", infilename, " is ", sum, \
      " and the average is ", average )
```

4

3. Modify the above program so that it prints out each number in the list along with a message as to whether the number was above, equalto, or below the average.
   e.g. print "2 is below the average of 7"
   While it is possible to do this without a list by reading the same file twice, that would be inefficient because reading from a file is slow compared to accessing a list.

Ways to slice and dice lists

1. Let's start with a short program that sets up a list of characters which also happen to be abbreviations for nucleotides:

```
bases = [ 'A', 'C', 'G', 'T' ]
# Now we will print each element of the list
print ( "Here are the list elements:" )
print ( "First element:  ", bases[0] )
print ( "Second element: ", bases[1] )
print ( "Third element:  ", bases[2] )
print ( "Fourth element: ", bases[3] )
```

The above code prints out:

```
First element:  A
Second element: C
Third element:  G
Fourth element: T
```

2. We can print the elements as a list, one a after another like this:

```
bases = ['A', 'C', 'G', 'T']
print ( "Here are the list elements: ", bases )
```

which produces the output:

```
Here are the list elements: ['A', 'C', 'G', 'T']
```

3. We can take an element off the end of a list with pop:

```
bases = ['A', 'C', 'G', 'T']
print ( "Here are the list elements: ", bases )
e = bases.pop()
print ( "Popped element: ", e )
print ( "Here is the list after a pop: ", bases )
```

which produces the output:

```
Here are the list elements: ['A', 'C', 'G', 'T']
Popped element: T
Here is the list after a pop: ['A', 'C', 'G']
```

4. We can reverse a list:

```
bases = ['A', 'C', 'G', 'T']
bases.reverse()
print ( "Here's the list in reverse:", bases )
```

which produces the output:

Here's the list in reverse: ['T', 'G', 'C', 'A']

5. We can get the length of a list:

```
bases = ['A', 'C', 'G', 'T']
print ( "The length of the list is", len(bases) )
```

which produces the output:

The length of the list is 4

6. We can take the first element out of the list with a **pop(0)**:

```
bases = ['A', 'C', 'G', 'T']
e = bases.pop(0)
print ( "Element removed from the beginning: ", e )
print ( "Here's the remaining list: ", bases )
```

which produces the output:

```
Element removed from the beginning: A
Here's the remaining list: ['C', 'G', 'T']
```

7. We can put an element on the end of the list with an `append`:

```
bases = ['A', 'C', 'G', 'T']
e = bases.pop(0)
bases.append( e )
print ( "Element from the beginning put at the end: ", bases )
```

which produces the output:

```
Element from the beginning put at the end: ['C', 'G',
'T', 'A']
```

In the above program, we removed the first element and put it at the end: this is called a "rotate". We end up with the same elements but in a different order: as if we rotated the list.

8. We can insert an element at an arbitrary place in a list using the Python `insert` function:

```
bases = ['A', 'C', 'G', 'T']
bases.insert( 2, 'X')
print ( "List with X inserted at position 2: ", bases )
```

which produces the output:

```
List with X inserted at position 2: ['A', 'C', 'X', 'G',
'T']
```

5

Write a python program that will construct a list with the elements: ['A', 'T', 'C', 'G']. Print the contents of the list on a single line if possible and then add the sequence ['A', 'T', 'G'] to the beginning of the list, print it, and add the sequence ['C', 'A', 'G'] to the end of the list and print the entire list on a line.

Submit 5 programs requested – the numbers in the blue circles.

The Python for loop:

The while loops in the above programs have a structure that reappears in many loops,

```
count = 0                                  # 1. initialization
while ( count < size ):                    # 2. logical condition
    sum = sum + listofnumbers[count]
    count = count + 1                      # 3. update
```

First we initialize the main loop variable, **`count`**. We check its value (**`count < size`**) to see if we should execute the loop. At the end of each iteration of the loop, we update the loop variable **`count=count+1`**

The above loop causes the **`count`** variable to take on the values `0, 1, 2, … size-1` in the statement where we compute the sum – i.e. we end up adding: `listofnumbers[0] + listofnumbers[1] + listofnumbers[2] + … + listofnumbers[size-1]`.

This strategy of having the count variable take on the indices of the list or array can also be accomplished using the range function: `range( size )` creates the list `[0, 1, 2, … size-1 ]`

Thus the three `while` loop operations can be specified in a **`for`** loop using the `range` function:

```
for count in range(size):
    sum = sum + listofnumbers[count]
```

This will compute the same sum. The advantage of using a `for` loop is that it is more succinct and easier to understand after we get used to it. The loop initialization and control is in one line and the loop body consists of the main operation we are performing – no need to explicitly update the count variable. Once we get used to it, it becomes much more readable than the equivalent `while` loop.

If we are indexing through a list, there is an even more elegant way of "walking" through the list using a for loop. For example, the above for loop can be rewritten as:

```
for item in listofnumbers:
    sum = sum + item
```

which can be read as: for each item in the list, do the loop body.

Saving your work

Sites like repl.it will save all your work and you can download your programs to your computer.

If you use a Linux Virtual Machine, it is only available when you start up the VM. So, if you shut the VM down, your work will not be accessible. A couple of ways you can get around this:

1. Save your files on a Shared drive
2. If you have Canvas, save your files in your Canvas "Files" area or, for assignments, create a .zip file and upload it into a Canvas assignment when you are done.