Lab #5

Purpose

The purpose of this lab is to learn how to debug Python **for** loops and to develop Python programs with our own custom functions.

Tasks

In this lab we will do more debugging, write Python functions and use them in programs.

Success

To succeed in this lab, we will pay close attention to the idea of how for loops and Python functions work.

Introduction

In this lab we will write and debug Python programs in which we use for loops and define our own functions.

Part 1: Debugging for loops

The following program (see **lab5n1.py**) tries to count the number of each kind of nucleotide in a DNA sequence but the program has some errors:

```
#!/usr/bin/python3
dnaseq = "GATCGCacgtcAGC"
print ("Starting DNA sequence is: ", dnaseq)

dna_array = list( dnaseq )

a_count = 0
c_count = 0
t_count = 0
g_count = 0

for nuc in dna_array:
   if ( nuc == 'A' ):
      a_count += 1           # x += 1     adds 1 to x
   elif ( nuc == 'C' ):
      c_count += 1
   elif ( nuc == 'T' ):
      t_count += 1
   else:
      g_count += 1
#  print counts of A, C, T and G
#
print ( " Counts:" )
print ( "   A: ", a_count )
print ( "   C: ", c_count )
print ( "   T: ", t_count )
print ( "   G: ", g_count )
```

Download the `lab5n1.py` file to your desktop, and use the repl.it web site or a command line to run the program in “debug” mode using the command as seen in Lab #4.

Here are some of the commonly used **Pdb** debug commands (also see Lab #4 for more details) are:

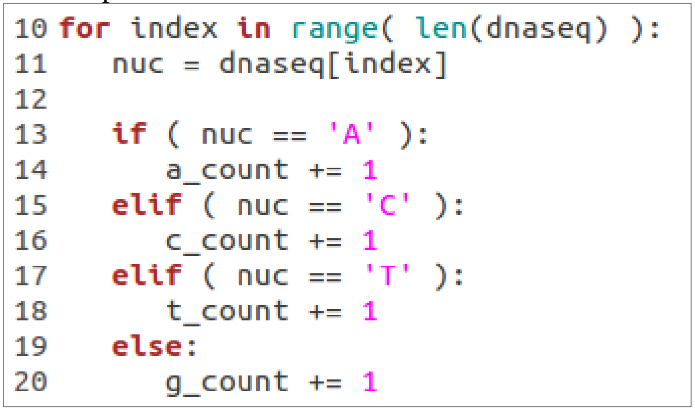
```
List source lines:
  l           List source code
  h [db_cmd]  Get help on command
  a           List arguments of current function
  q           Quit
  exit           Quit

Control script execution: 
  s           Single step
  n           Next, steps over functions
  <CR/Enter>  Repeat last n or s
  run         Run from start
  c           Continue until position
  r           Return from function

Debugger controls:
  b [line]    Set breakpoint
  tbreak [line]  Set breakpoint and disable it when reached
  disable [line] Disable a breakpoint
  enable [line]  Enable a breakpoint
  ignore [line] [n]  Ignore a breakpoint n times
  clear       Delete a/all breakpoints
  condition   Set the condition for a breakpoint

Data Examination:
  p expr         Print expression
  pp expr        Pretty print expression
  whatis var     Print type information for variable
```

What problems do you see in this program as you debug it and how would you fix them? Your program should be able to handle sequences with upper and lower case letters: for example, your fixed program should report the total number of both upper-case 'A's and lower-case 'a's in the count for 'A' etc. For example the 'A' count should be 3 and not 2.

Fix the `lab5n1.py` program without modifying the **`dnaseq`** string and submit the “fixed” version.

Part 2: Writing Python functions

The program in **lab5n2.py** uses another version of the `for`-loop – the “traditional” for loop, which counts using an index variable. This version uses the Python `[]` operator to extract a single character out of a long string. Only the loop is shown here:

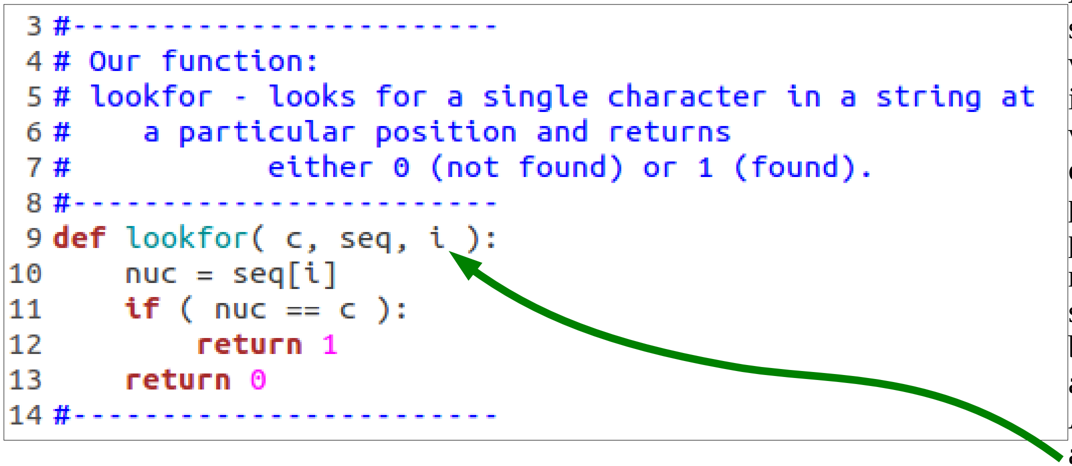


```
for index in range( len(dnaseq) ):
    nuc = dnaseq[index]

    if ( nuc == 'A' ):
        a_count += 1
    elif ( nuc == 'C' ):
        c_count += 1
    elif ( nuc == 'T' ):
        t_count += 1
    else:
        g_count += 1
```

Note that built-in Python functions like “`len`” and “`range`” are shown in light blue in an editor like gedit.

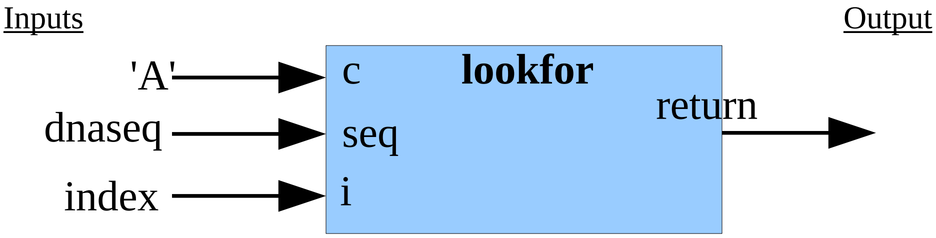
Fix this program without modifying the **dnaseq** string.
Turn in a fixed version of the `lab5n2.py` program so that it works with both upper and lower case letters in the input sequence.

The next program (**lab5n3.py**) shows you how to set up your own function. We use the Python word “**def**” to define a function:



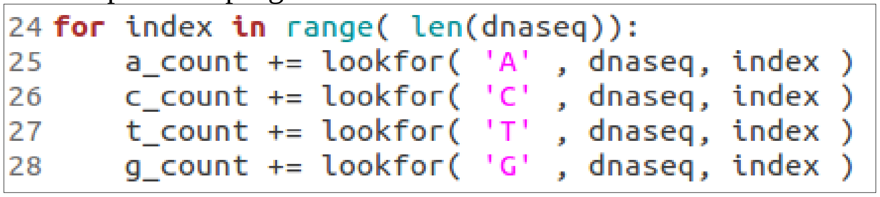
```
#------------------------
# Our function:
# lookfor - looks for a single character in a string at
#    a particular position and returns
#           either 0 (not found) or 1 (found).
#------------------------
def lookfor( c, seq, i ):
    nuc = seq[i]
    if ( nuc == c ):
        return 1
    return 0
#------------------------
```

A function is like a small “program” within a program. It is a piece of code with a name that you can use in other parts of your program. It is often represented schematically as a box with inputs and and an output.
A function can have a number of inputs.



We say that a function is “called” from the main program. For example, we call the built-in “**len**”

function in the for-loop.
Once you set up a function you can use it in a program by knowing its name and what kinds of inputs (and the order of the inputs) the function needs. In `lab5n3.py`, we use the **lookfor** function instead of the if statements in the previous programs:



```
24 for index in range( len(dnaseq)):
25     a_count += lookfor( 'A' , dnaseq, index )
26     c_count += lookfor( 'C' , dnaseq, index )
27     t_count += lookfor( 'T' , dnaseq, index )
28     g_count += lookfor( 'G' , dnaseq, index )
```

We say that at line 25 we "call" the **lookfor** function with the inputs 'A', `dnaseq`, and `index` and we get the output: either a 1 (if an 'A' was at position $index of the string $dnaseq) or a 0 is added to a running total, `a_count`.

As you step through this program (`lab5n3.py`) using the debugger, note that the "**`s`**" debugger command can take you **into** a function like **lookfor**. For example, if you set a breakpoint at line 25 and "continue" execution of the program to that line:

```
$ python -m pdb lab5n3.py
> Lab5/lab5n3.py(9)<module>()
-> def lookfor( c, seq, i ):
(Pdb) b 25
Breakpoint 1 at Lab5/lab5n3.py:25
(Pdb) c
Starting DNA sequence is:  GATCGCacgtcAGC
> Lab5/lab5n3.py(25)<module>()
-> a_count += lookfor( 'A' , dnaseq, index )
```

at this point, we are ready to execute line 25 which uses the **lookfor** function. Entering an "**s**":

```
(Pdb) s
--Call--
> Lab5/lab5n3.py(9)lookfor()
-> def lookfor( c, seq, i ):
(Pdb)
```

gets us into the first line of the **lookfor** function. We can now step through the lines of the `lookfor` method until we "return" to the main program.
If, instead of stepping "into" a function, you want to step "over" it (executing all the lines of the function in one "step"), you can use the "**`next`**" debugger command instead of the "**`s`**".
Use the "**`n`**" debugger command to step "over" each of the lines, 25-28.

Fix this program – `lab5n3.py` – without modifying the **`dnaseq`** string.
Submit a fixed version. The function does not need to be changed.

<u>A note on writing functions:</u>

Functions are used to make a program more "modular" - so that the different parts or "modules" of a program can be tested separately and understood more easily. A good indication of whether we should use a function would be if we saw very similar code being repeated. For example, although we could use the string.count or the list.count functions to count the number of amino acids in a protein sequence, suppose we had to write our own code as shown below to count the occurrences of letters:

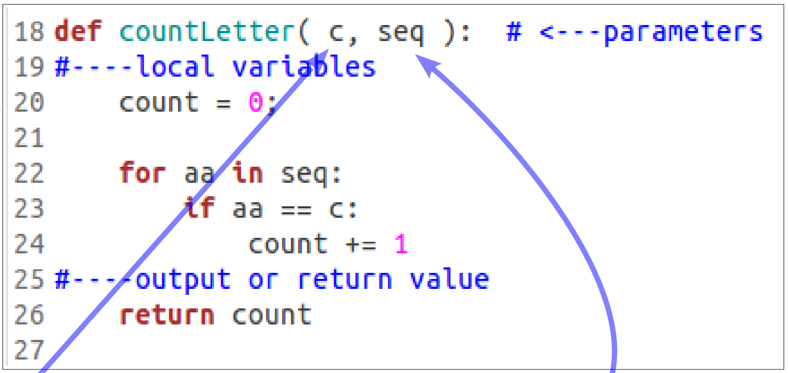
```
for aa in protseq:
    if aa == 'I':
        I_count += 1
for aa in protseq:
    if aa == 'L':
        L_count += 1
for aa in protseq:
    if aa == 'V':
        V_count += 1
```

If we were to do this 20 times, we would have a lot of repeated code. If, instead of the above, we were to use a function called "countLetter" (see functionexample.py), we could have something like:

```
    I_count = countLetter( 'I', protseq );
    L_count = countLetter( 'L', protseq );
    M_count = countLetter( 'M', protseq );
```

We can see that we "pass in" the inputs **'I'** and **protseq** (also called "arguments" to the function). There is an order to the inputs of a function and you may already have guessed that using a different order will result in the program crashing. It is up to us, the Python programmers, who set up the **countLetter** function to decide what the order should be and then we have to stick to that order.

For example, the arguments passed into the function **countLetter** will be placed into "temporary variables" called "parameters" that are only seen inside the function. That is, the argument **'I'** in main is copied into the variable **c** inside **countLetter** and the argument **protseq** in main is copied into the parameter **seq** inside the function



```
def countLetter( c, seq ):  # <---parameters
#----local variables
    count = 0;

    for aa in seq:
        if aa == c:
            count += 1
#----output or return value
    return count
```

In addition to parameters, functions can use any number of "local variables" (such as **count** inside **countLetter**) that are also only visible inside the function.

The output of a function can be stored in a variable in the main program, or can be used in an arithmetic expression (as we did in lab5n3.py) or can be part of a print statement:

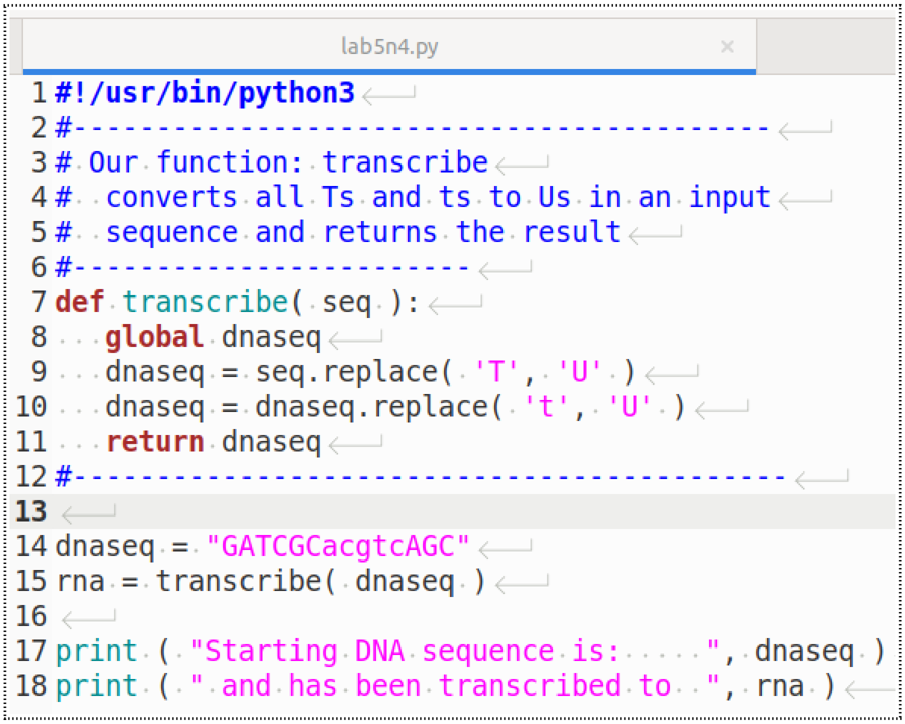
```
    #dnaseq = "GATCGCacgtcAGC"
    print ( "Starting DNA sequence is: ", dnaseq
    print ( "   Length of sequence is: ", len(dnaseq)

    #
    #  print counts of A, C, T and G
    #
    print ( " Counts: "
    print ( "   A: ", countLetter( 'A',  dnaseq ) )
    print ( "   C: ", countLetter( 'C',  dnaseq ) )
    print ( "   T: ", countLetter( 'T',  dnaseq ) )
    print ( "   G: ", countLetter( 'G',  dnaseq ) )
```

In the above example (see functionexample.py), notice that we no longer need a loop in the main program. We put all of that complexity into a function which performs a task that is specific enough that we don't necessarily need to know all of its details but general enough that we can call it with four different inputs and have it do four different things. Although we use it to count nucleotides, we could also use it for other things like counting amino acids in a protein sequence. Lastly, note that the countLetter function calls another function. We can have many such "levels" of function calls and we are only limited by the available memory on our computers.

The last program (**`lab5n4.py`**) shows an example of a common bug that Python programmers encounter when writing programs:

Run the program. Can you spot the problem in the output? Where did we go wrong?

Hint: The problem is in the function. In Python, when we are executing code inside a function, we can access all the variables of the main program as well by using the keyword `global`. We can see and change such variables and as a result we may end up changing variables in the main program unintentionally.
See if you can fix the error in `lab5n4.py` and turn in the fixed program.



```python
#!/usr/bin/python3
#------------------------------------------
# Our function: transcribe
#  converts all Ts and ts to Us in an input
#  sequence and returns the result
#------------------------
def transcribe( seq ):
   global dnaseq
   dnaseq = seq.replace( 'T', 'U' )
   dnaseq = dnaseq.replace( 't', 'U' )
   return dnaseq
#-------------------------------------------

dnaseq = "GATCGCacgtcAGC"
rna = transcribe( dnaseq )

print ( "Starting DNA sequence is:     ", dnaseq )
print ( " and has been transcribed to  ", rna )
```

Submit the 4 programs and list all the ways you tested the program

Appendix I: Python operators

Assignment operator

`=`	assigns a value to a location; ex:	`a = 5`

Arithmetic operators

`+`	add two values; ex:	`a + 5`
`-`	subtract a value from another; ex:	`a - 5`
`*`	multiply two values; ex:	`a * 5`
`/`	divide a value by another; ex:	`a / 5`
`%`	remainder of division of a value by another; ex:	`a % 5`
`**`	raise a value to the power of another; ex:	`a ** 5`

Boolean comparison operators

`>` `>=`	greater than (or equal to); ex:	`a > 5`
`<` `<=`	less than (or equal to); ex:	`a <= 5`
`==` `!=`	equal to or not equal to; ex:	`a != 5`

Appendix II: Python keywords

Python has a small number of "reserved" words – these are words that mean something specific. An example is the keyword, **`if`**, which is used to control program flow: it is a word that is used to execute some code only if some condition is true. Another way to think about reserved words is that these cannot be used as names of variables; for example, a statement like `if = 30` would not be allowed because `if` is a reserved word.

Control flow:

  - **`if`**, **`elif`**, **`else`** - for conditional execution
  - **`for`**, **`while`** – for loops
  - **`break`** – exits the innermost loop it is in
  - **`continue`** – ends the current iteration of a loop and starts the next one
- Functions, classes, and modules
  - **`def`** – define a function
  - **`class`** – define a class, this is used in object oriented programming
  - **`lambda`** – a function without a name
  - **`import`** – use an external module, i.e. another Python file
  - **`from`** – use specific functions or classes in an external module
  - **`return`** – return from a function to calling location
- Exceptions and assertions
  - **`try`**, **`except`**, **`else`**, **`finally`**, **`raise`** – for a try code block that may “raise” exceptions
  - **`assert`** – for assertions: if some condition is not true, an exception is raised
- Expressions
  - **`and`**, **`or`**, **`not`** – boolean operators
  - **`in`** – test for membership in a collection
  - **`is`** – test for identity of two objects
- Other
  - **`del`** – delete local variables or list, tuple, or dictionary members
  - **`exec`** – execute a Python program from inside a program
  - **`global`** – allows access to a variable outside the “local” scope
  - **`pass`** - do nothing
  - **`print`** – print a string to console
  - **`yield`** – return a list-like object that will be read once, a generator

Appendix III: Python Data types

- Numeric types:
  - Floats – Python floats are double-precision (64-bit) floating point numbers
  - Integer – these are 64-bit integers
  - Long Integer - “unlimited” size integers
- Sequence types:
  - String – an immutable list of characters
  - List – a mutable collection of objects; ex: `[ 3, 14, 3.1415, “pi” ]`
  - Tuple – an immutable list; ex: `( 3, 14, 3.1415, “pi” )`
  - Dictionary – A list of key-value pairs; ex:
    `{ “first”:“Abraham”, “middle”:“”, “last”:“Lincoln” }`