Lab #4: Debugging Python programs

Purpose

The purpose of this lab is to debug Python programs that may have bugs or errors.

Tasks

In this lab we will write and run Python programs and look for ways to fix programming errors.

Success

To succeed in this lab, we will pay close attention to how Python statements work.

Introduction

In this lab we will use the “debugging” capabilities of Python.

Part 1: A working Python program

The following program (`lab4n1.py`) performs a number of operations (may not be useful ones) on a DNA sequence:

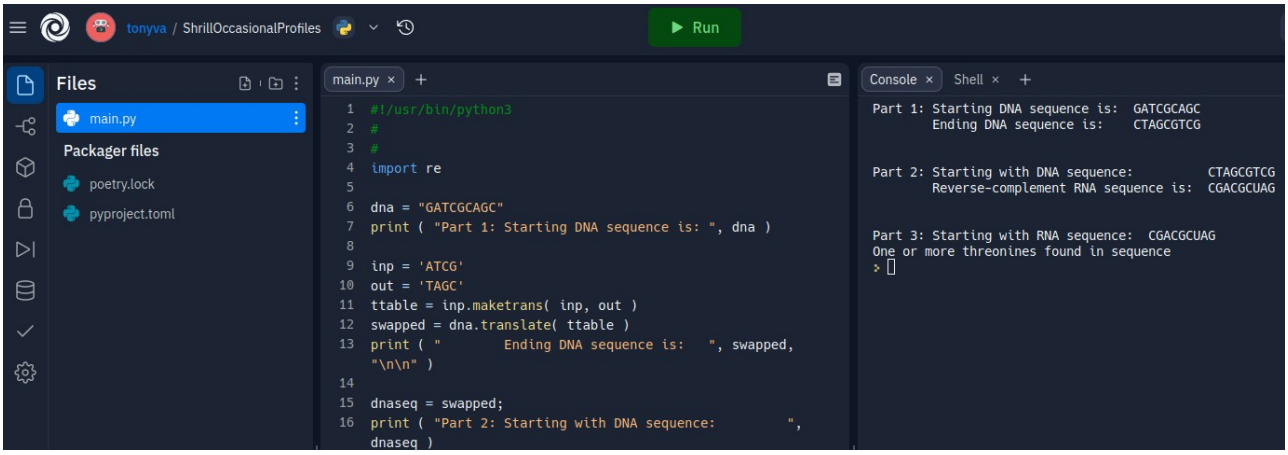
```
#!/usr/bin/python3
#
import re

dna = "GATCGCAGC"
print ( "Part 1: Starting DNA sequence is: ", dna )
inp = 'ATCG'
out = 'TAGC'
ttable = dna.maketrans( inp, out )
swapped = dna.translate( ttable )
print ( "        Ending DNA sequence is:   ", swapped, "\n\n" )

dnaseq = swapped;
print ( "Part 2: Starting with DNA sequence:         ", dnaseq )
rinp = 'ATCG'
rout = 'UAGC'
rtable = dna.maketrans( rinp, rout )
rseq = dnaseq.translate( rtable )
rseq = ''.join( reversed( rseq ) )
print ( "     Reverse-complement RNA sequence is: ", rseq  )

print ( "Part 3: Starting with RNA sequence: ", rseq
rna = rseq
threonines = 'AC[UCAG]'
if ( re.search( threonines, rna ) ):
   print ( "One or more threonines found in sequence" )
else:
   print ( "No Threonines in this sequence" )
```

The above program (`lab4n1.py`) works but may not be completely obvious. To try to understand how it works, we can download the `lab4n1.py` file to your desktop and use a debugger to “step through the program”.

There are a couple of ways to debug Python programs

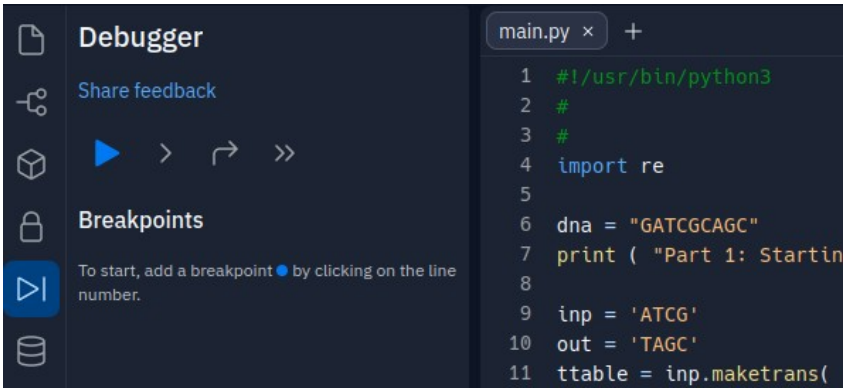
1) **Debugging in the `repl.it` website**

Upload the `lab4n1.py` Python program to `repl.it` and run it. We will likely see something like this:



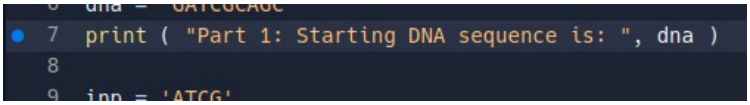
Click on the Debugger icon  in the pane on the left side of the page.

This will give us a “Debugger” pane instead of the “Files” pane:

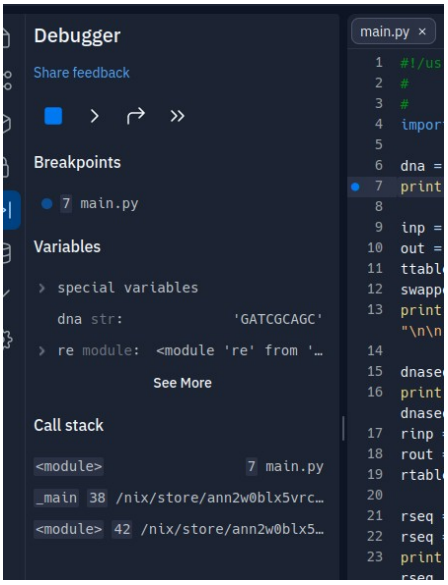


Add a “breakpoint” at line 7 by clicking just left of the “7” or else by right-clicking near the “7” and choosing “Toggle breakpoint”

This should result in line 7 having blue circle next to it:



Click the blue triangle to start execution of the program until line 7 is reached and it will also display lists of Breakpoints, Variables, and the Call Stack in the Debugger pane:

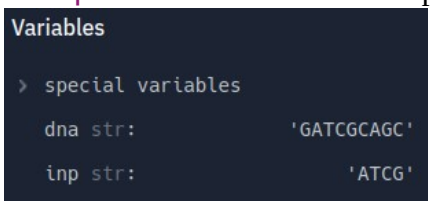


We can see that the variable named “dna” has the value “GATCGCAGC”
The debugger is a way to peer into a running program and see what values the variables in the program have at any given time. We could also “open” up the “special variables” section to see what values these variables have but in this case, these are not very helpful.

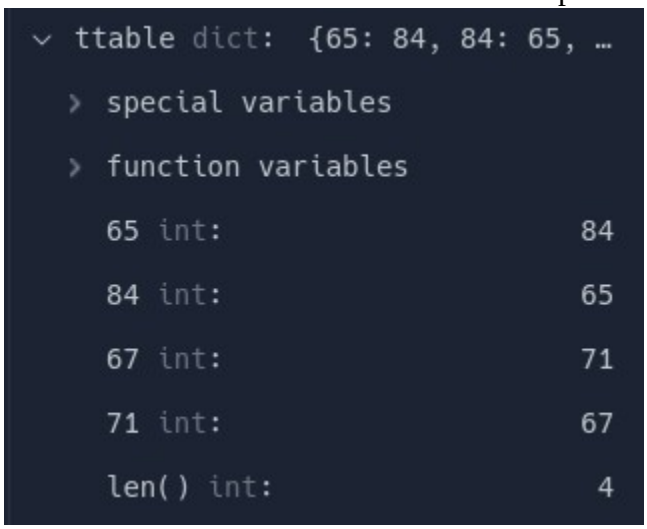
|Click the “Next Step” button  to execute line 7 and see what changes in the Console window. We should see this output show up:

```
Part 1: Starting DNA sequence is:  GATCGCAGC
```

This particular change in the Console window was caused by the execution of the `print()` statement in the program.
Executing any particular statement may not produce any output in the Console or any new things in the Variables part of the Debugger pane. Clicking the "Next Step" button again should execute line 9 and display the `inp` variable in the Variables part:



Click the "Next Step" button to execute line 10 and we will see the `out` variable in the Variables part appear and its value should make sense. Clicking the button to execute line 11 should give us a look at the `ttable` variable in the Variables part:



We see that the `ttable` variable is a "dict" or a Python dictionary and its values look like numbers but they are actually ASCII representations of letters. See if you can make sense of what the Debugger tells you about the `ttable` variable by comparing it with the Python code.

Click the "Next Step" button to execute line 12 and we should see the `swapped` variable appear in the Variables section – note that variables are shown in alphabetically sorted order.

We could also have done this by placing a second breakpoint at line 13 and clicking Debug and then the "Next Breakpoint" button. We can also place breakpoints at lines 21, 26, and 29 and progress to these various points in the program and observe the values of all the variables we would

2) **Debugging from the command line** An alternative to using `replt.it` is to run Python programs from your command line. On a Linux or macOS system, we can use the `chmod` command to make a Python .py program executable and then run it using the command:
`python3 -m pdb lab4n1.py`

Note the "-m" option in the python command tells the python interpreter to load a module, in

this case, the pdb.py module. The pdb.py module is the Python debugger module and this lets us run the lab4n1.py program in "Script debug" mode. In debug mode, we will see something different from ordinary Perl programs when we run it:

```
tony@corei7$ python3 -m pdb lab4n1.pl
> lab4n1.py(4)<module>()
-> import re
(Pdb)
```

When we run the program, we get a few lines of output and a "**(Pdb)**". This is the "**debugger prompt**". This tells us that the debugger is waiting for us to type in a debugger command. Above the prompt there is a line saying which file we are executing.
We can enter the letter h (and the Enter key) or help command will give you a condensed list of commands you can use:

```
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv

(Pdb)
```

This is the first command you want try – the help command. Enter "h", hit the Return key and see what happens.

You do not have to memorize all these commands nor do you have to remember what each command does. Typically, one builds up a repertoire of useful commands and the help page can be consulted.
Try the "**l**" command to list the program. You should see something like:

```
(Pdb) l
  1     #!/usr/bin/python3
  2     #
  3     #
  4  -> import re
  5
  6     dna = "GATCGCAGC"
  7     print ( "Part 1: Starting DNA sequence is: ", dna )
```

```
  8
  9     inp = 'ATCG'
 10     out = 'TAGC'
(Pdb)
```

The "**->**" means that the debugger is ready to execute the 4th line of the program but has not yet executed it. To execute only this line and stop at line 5, use the "**s**" command, followed by the "**l**" command:

```
(Pdb) s
> /home/tony/Lab4/lab4n1.py(5)<module>()

(Pdb) l
  1     #!/usr/bin/python3
  2     #
  3     #
  4     import re
  5
  6  -> dna = "GATCGCAGC"
  7     print "Part 1: Starting DNA sequence is: ", dna
  8
  9     inp = 'ATCG'
 10     out = 'TAGC'
(Pdb)
```

We see that the step command executed one line (line 4) of the program and stopped at line 5. The list command then listed 10 lines starting with the line to be executed next. Enter one more "**s**" command. Since we will have executed line 6, we should be able to see the effect of assigning a string value to the `dna` variable. We can do this by using the "print" debug command, "**p**":

```
(Pdb) p dna
'GATCGCAGC'
```

Note that we can only see changes in a variable after we execute a statement that may change the value of that variable. For example, the variable `swapped` has yet to be used. If we try to print it, we will get output that says that the variable has not yet been "defined":

```
(Pdb) p swapped
*** NameError: NameError("name 'swapped' is not defined",)
```

Use the step command again to execute line 8 (the line with the print statement):

```
(Pdb) s
Part 1: Starting DNA sequence is:  GATCGCAGC
> lab4n1.py(10)<module>()
-> inp = 'ATCG'
```

Note that the output of the print statement shows up immediately after the step command is entered.

Step again four more times so that we step over the statement "`swapped = dna.translate( ttable )`" and then "print" the `dna` and `swapped` variables:

```
(Pdb) s
> lab4n1.py(14)<module>()
-> print "        Ending DNA sequence is:   ", swapped, "\n\n"
(Pdb) p dna
'GATCGCAGC'
(Pdb) p swapped
'CTAGCGTCG'
(Pdb)
```

We see that the **`dna`** has the value, `GATCGCAGC` and **`swapped`** has the result of the string-translation. As expected, we have transliterated each base to its complement. To see where we are in the program, we can use the "list" command, "**`l`**":

```
(Pdb) l
  8
  9       inp = 'ATCG'
 10       out = 'TAGC'
 11       ttable = maketrans( inp, out )
 12       swapped = dna.translate( ttable )
 13  ->   print "        Ending DNA sequence is:   ", swapped,
"\n\n"
 14
 15       dnaseq = swapped;
 16       print "Part 2: Starting with DNA sequence:       ",
dnaseq
 17       rinp = 'ATCG'
 18       rout = 'UAGC'
(Pdb)
```

Note that line #14 is empty – compare with the original file and you should see an empty line in the "source code", the original program. Step and print the various variables you come across until you come to the end of the program. Use the "**q**" command to quit the debugger.

What does this program accomplish with the starting sequence?
Did the debugger help you understand what each line of the program does?

Run the program the "normal" way: **`./lab4n1.py`** and take a look at the output of the program. This particular program does not have any loops or rely on input from the user or from files and so it is a relatively simple program to step through with the debugger. The **`q`** command quits the debugger.

There is nothing to submit for this part of the lab.

Part 2: A Python program with an error

The next program (**lab4n2.py**) has a loop that will search for subsequences within a larger sequence:

```
#!/usr/bin/python3
dnaseq = "TATAGCCATGACCATATAGGA"
dna_array = list( dnaseq )
for nuc in dna_array:
    print ( " ", nuc )
print ( "Done" )

size = len( dna_array )
search1 = "ATG"
search2 = "TATA"
search1found = 0
search2found = 0

for i in range( size ):
    subseq = [ dna_array[i] , dna_array[i+1] , dna_array[i+2] ]
    subseq1 = ''.join( subseq )
    if ( subseq1 == search1):
        search1found = search1found + 1
    subseq.append( dna_array[i+3] )
    subseq2 = ''.join( subseq )
    if ( subseq2 == search2):
        search2found = search2found + 1

print ( search1found, " occurences of ", search1, " found in ", dnaseq )
print ( search2found, " occurences of ", search2, " found in ", dnaseq )
```

Run the program – either in repl.it or from the command line - as is and note that while it seems to work partially, we do get an error message:

```
Traceback (most recent call last):  File "./lab4n2.py", line 19, in <module>
    subseq.append( dna_array[i+3] )
IndexError: list index out of range
```

We want to figure out why we see this error message – it is caused by some bug in the program and we want to fix the bug.

Remember from Lab #2 that lists and loops, especially for-loops go well together:

The for loop

A while loop that adds up all the numbers in an array of numbers has a structure that reappears in many loops:

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

First we initialize the main loop variable, **count**. We check its value (**count < len(listofnums)**) 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**).

These three operations are specified more succinctly in a for loop:

```
for (count in range( len(listofnums) ):
    sum = sum + listofnums[count]
```

The advantage of using a for loop is that all of the looping initialization and control is in one line and the loop body consists of the main operation we are performing – once you get used to it, it is much more readable than the while loop.

If we are indexing through an an array, there is an even more elegant way of “walking” through the array, the foreach loop. For example, the above for loop can be rewritten as:

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

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

Debugging from the command line:
Use the command "**python3 -m pdb lab4n2.py**" to debug the program and step through this program as we did with the first program, printing variables to check their values. We may see a number of statements that we may not have used very much in previous programs.
In the first five lines, we take a string representing a nucleotide sequence and we split it into a list. Then we use a for-loop to print each nucleotide in the sequence out with spaces in between.

As we debug the lab4n2.py program, we may encounter a debug session like this:

```
tony@corei7$ python3 -m pdb lab4n2.py
> lab4n2.py(2)<module>()
-> dnaseq = "TATAGCCATGACCATATAGGA"
(Pdb) l
  1       #!/usr/bin/python
  2  ->   dnaseq = "TATAGCCATGACCATATAGGA"
  3       dna_array = list( dnaseq )
  4       for nuc in dna_array:
  5           print ( " ", nuc )
  6       print ( "Done" )
  7
  8       size = len( dna_array )
  9       search1 = "ATG"
 10       search2 = "TATA"
 11       search1found = 0
(Pdb) s
> lab4n2.py(3)<module>()
-> dna_array = list( dnaseq )
(Pdb) s
> lab4n2.py(4)<module>()
-> for nuc in dna_array:
(Pdb) s
> lab4n2.py(5)<module>()
-> print " ", nuc
(Pdb) s
  T
> lab4n2.py(4)<module>()
-> for nuc in dna_array:
(Pdb) s
> lab4n2.py(5)<module>()
-> print " ", nuc
(Pdb) s
  A
> lab4n2.py(4)<module>()
-> for nuc in dna_array:
(Pdb) s
> lab4n2.py(5)<module>()
-> print " ", nuc
(Pdb) s
  T
> lab4n2.py(4)<module>()
-> for nuc in dna_array:
(Pdb)
```

That is, the print statement at line 5 is executed over and over again – this is what is expected because it is in a loop that will print out each scalar value in the array `dna_array`. Also, the nucleotides are printed after the "debug step command".

If we are reasonably sure that the errors are near lines 15-22, we might be able to safely skip the first loop. To do this we can set a "**breakpoint**" – that is, we can tell the python debugger to stop at a particular line in the program. For example, if we want to stop at line 8, after the first loop, we can use the command "**`b 8`**":

```
tony@corei7$ python3 -m pdb lab4n2.py
> lab4n2.py(2)<module>()
-> dnaseq = "TATAGCCATGACCATATAGGA"
(Pdb) b 8
Breakpoint 1 at lab4n2.py:8
(Pdb) l
  1     #!/usr/bin/python3
  2 ->  dnaseq = "TATAGCCATGACCATATAGGA"
  3     dna_array = list( dnaseq )
  4     for nuc in dna_array:
  5         print ( " ", nuc )
  6     print ( "Done" )
  7
  8 B   size = len( dna_array )
  9     search1 = "ATG"
 10     search2 = "TATA"
 11     search1found = 0
(Pdb) >
```

Note that after setting the breakpoint and then listing the program we see a "B" next to line #8 indicating that there is a breakpoint at that line. Also note that we have a "->" next to line 2 indicating that we are about to execute line 2. Setting a breakpoint does not execute all the code up to that point, it only tells perl that we want to stop there. To execute all statements from the current point to the next breakpoint, we use the "continue" or "c" command:

```
(Pdb) c
T
A
T
A
G
C
C
A
T
G
A
C
C
A
T
A
T
A
G
```

```
  G
  A
Done
> lab4n2.py(8)<module>()
-> size = len( dna_array )
(Pdb) l
  3     dna_array = list( dnaseq )
  4     for nuc in dna_array:
  5         print ( " ", nuc )
  6     print ( "Done" )
  7
  8 B-> size = len( dna_array )
  9     search1 = "ATG"
 10     search2 = "TATA"
 11     search1found = 0
 12     search2found = 0
 13
(Pdb)
```

Note that we did execute all the statements up to the breakpoint and we see the output of the print (in the first for loop) appear after we enter the **c** command. Since the next few statements are rather straightforward, we can set a breakpoint for line 16 and “continue” again to that line:

```
(Pdb) b 16
Breakpoint 2 at lab4n2.py:16
(Pdb) c
> lab4n2.py(16)<module>()
-> subseq1 = ''.join( subseq )
(Pdb)
```

At this point, we are positioned inside the second for loop and we can step through this loop examining the values:

```
(Pdb) s
> lab4n2.py(17)<module>()
-> if ( subseq1 == search1):
(Pdb) s
> lab4n2.py(19)<module>()
-> subseq.append( dna_array[i+3] )
(Pdb) s
> lab4n2.py(20)<module>()
-> subseq2 = ''.join( subseq )
(Pdb) p subseq
['T', 'A', 'T', 'A']
(Pdb) s
> lab4n2.py(21)<module>()
-> if ( subseq2 == search2):
(Pdb) p subseq2
'TATA'
(Pdb)
```

Keep stepping through the code until you encounter the statement that causes the error message. Remember, that you can use the **c** command (for continue) to take large "steps" through the code. Find the cause of the error statements in the `lab4n2.py` . Fix the "bug" without using external functions like the string.count() method – the current code should work in the fixed version.

Possible fix: use `if` statements to set the `subseq`, `subseq1`, and `subseq2` variables only if we have enough characters in the `dnaseq` array. This means that we change the second loop to look like this:

```
for i in range( size ):
    if ( i < (size-2) ):
        subseq=[dna_array[i],dna_array[i+1],dna_array[i+2]]
        subseq1 = ''.join( subseq )
    if ( subseq1 == search1):
        search1found = search1found + 1
    if ( i < size-4 ):
        subseq.append ( dna_array[i+3] )
        subseq2 = ''.join( subseq )
    if ( subseq2 == search2):
        search2found = search2found + 1
```

This appears to work: the modified ptogram finds 1 ATG and 2 TATAs. However, if we test further and change the starting sequence to: `dnaseq = "TATAGCCATGACCATATATG"` the "fixed" program finds 4 ATGs! Our "fix" introduces a new, even more subtle bug!
This is a bad "fix".

Submit a document that is 1 page or longer of single-spaced 12-point font text that **explains in detail**:

- ½ a page or more: what the program does, what "bug" was: what was the cause of the error in `lab4n2.py` and how you used the debugger to find the error.
- ½ a page or more: how we can fix the bug and what strings you used to test your new "fixed" version of the program

Submit your fixed version of `lab4n2.py`

References:

- Debugging python programs in repl.it: https://docs.replit.com/programming-ide/debugging
- Command line debugging: https://realpython.com/python-debugging-pdb/