Bi 410/510: Introduction to Python

Links

1 Reading: Ch 8

- 1.1 follow instructions, create your own version of `dnacalc`, the example program in Ch 8

2 Hello, World

- 2.1 use your text editor to create a new file called `dnacalc.py`
- 2.2 Mac users enter these four lines:

  text color from TextMate

```
#! /usr/bin/env python3

dna_seq = 'ATGAAC'
print("Sequence:", dna_seq)
```

- 2.3 Windows users can skip the `#!` line, just enter the last two lines above:

```
dna_seq = 'ATGAAC'
print("Sequence:", dna_seq)
```

- 2.4 **NOTE:** the example in the book is based on Python 2.7, so the print statement does not include any parentheses:

  print in Python3

```
print "Sequence:", dna_seq
```

- 2.5 **NOTE:** the book use `DNAseq` instead of `dna_seq`

  see “conventions for names” below

- 2.6 save the file in one of your project directories
  - on my Windows system I saved it in Classes\Bio 410
- 2.7 start your terminal emulator, use `cd` commands to navigate to the folder where you saved the file
- 2.8 Mac users need to type this command to make the file “executable”

```
chmod +x dnacalc.py
```

- 2.9 to run the program, just type the name of the file.
- 2.10 if everything works, you should see the output in the terminal window
  - this is what I see when I run the program on my Mac:

```
[sasquatch:programs] ➤  dnacalc.py
Sequence: ATGAAC
```

3 Assignment Statements

- 3.1 the first line in the program is an example of an **assignment statement**
- 3.2 there is a name on the left of the = and an expression on the right
- 3.3 Python creates a variable and assigns it the value of the expression
- 3.4 rules for names:
  - start with a letter
  - can be arbitrarily long
  - may contain digits and underscore (_) characters

4 Aside: Interactive Python

- 4.1 if you run Python from the command line without specifying the name of a file it starts an interactive session
  - Mac users: type `python3` to launch Python 3.3 or later

- Windows users: type python

- 4.2 you can type Python statements and test them out before you put them in a program

- 4.3 try typing these assignment statements to experiment with names: >>> is Python's prompt

```
>>> n = 21
>>> sample_size = 100
>>> FirstName = 'Harry'
```

- 4.4 to see the value of a variable just type its name:

```
>>> n
21
>>> sample_size
100
>>> FirstName
'Harry'
```

- 4.5 in this example text in blue is what I typed, text in black is Python's response

- 4.6 we can use the value of a variable in an expression on the right side of an assignment:

```
>>> n * sample_size
2100
>>> npoints = n * sample_size
```

5 Interactive Experiments

- 5.1 learn to use interactive sessions to run "experiments"

- 5.2 if you ever have a question see if you can figure out how to answer it using an "interactive experiment"

- 5.3 we'll see lots of examples in this outline

6 Conventions for Variable Names

- 6.1 use descriptive names that help you remember what the variable is used for

- 6.2 start names with underscore letters

- 6.3 prefer underscores for multi-part names

  - `sample_size` instead of SampleSize

  - `dna_seq` instead of DNAseq

7 Operations on Strings

- 7.1 use a built-in function named `len` to count the number of characters in a string

```
>>> s = 'epsilon'

>>> len(s)
7
```

- 7.2 other operations are defined by **methods**

- 7.3 write the name of a variable that refers to a string, a period, the method name, and a pair of parentheses

```
>>> s.upper()
'EPSILON'

>>> s.capitalize()
'Epsilon'
```

- ☐ 7.4 some methods expect us to supply **arguments** when we call them

```
>>> s.find('sil')
2

>>> s.count('i')
1
```

☐ 8 Base Counts

- ☐ 8.1 add the following lines to your dnacalc.py program to have it compute the base counts in the DNA string

```
a_count = dna_seq.count('A')
c_count = dna_seq.count('C')
g_count = dna_seq.count('G')
t_count = dna_seq.count('T')

print(a_count, c_count, g_count, t_count)
```

- ☐ 8.2 test the new version; this is the expected output:

```
[sasquatch:programs] ➤  dnacalc.py
Sequence: ATGAAC
3 1 1 1
```

☐ 9 Percentages

- ☐ 9.1 instead of counts let's print percentages

- ☐ 9.2 before we modify the program we'll do some experiments in an interactive Python session

```
>>> 5 + 6
11

>>> 5 - 6
-1

>>> 5 * 6
30

>>> 30 / 5
6.0

>>> 31 / 5
6.2
```

- ☐ 9.3 what we learned:
  - ☐ if operands to +, -, and * are integers the result is an integer
  - ☐ the division operator / creates "floating point" values — Python3; see note below for Python2
  - ☐ if you want an integer result (and there will be situations where we do) use the // operator

```
>>> 31 // 5
6
```

- ☐ 9.4 in Python 2.7 (used in the textbook) dividing one integer by another produces an integer result

```
>>> 31 / 5
6
```

  - ☐ if we need the result to be a real number (e.g. when we're computing percentages) we need to turn one of the operands into a real before doing the division
  - ☐ use a builtin function named `float` to convert an object into a floating point number

```
>>> 3 / 7
0
```

```
>>> float(3) / 7
0.42857142857142855
```

- the same function converts strings of digits into a floating point number

  `float` is still useful in Python3

```
>>> float("3.14159")
3.14159
```

- 9.5 change the `print` statements in your dnacalc program to print percentages

```
seq_length = len(dna_seq)

print('A:', a_count / seq_length)
print('C:', c_count / seq_length)
print('G:', g_count / seq_length)
print('T:', t_count / seq_length)
```

- 9.6 the result:

```
[sasquatch:programs] ➤  dnacalc.py
Sequence: ATGAAC
A: 0.5
C: 0.16666666666666666
G: 0.16666666666666666
T: 0.16666666666666666
```

10 Comments

- 10.1 as programs get bigger it's important to add documentation
- 10.2 if a line contains a "pound sign" (#) Python ignores it and the rest of the line
- 10.3 put full-line comments at the top of the file to give a short overview of the program
- 10.4 sprinkle additional comments throughout the program

  see also: "docstrings"
- 10.5 include partial comments at the end of a line for short notes
- 10.6 example from my version of `dnacalc.py`

```
# Convert the counts to percentages

seq_length = len(dna_seq)

print('A:', a_count / seq_length)    # note the division operator in Python3
print('C:', c_count / seq_length)    # always returns floats
print('G:', g_count / seq_length)
print('T:', t_count / seq_length)
```

11 Formatted Output

- 11.1 the output from the program is correct but not very pretty
- 11.2 we want to have all percentages appear as numbers of the form NN.NN, e.g. 16.67 instead of 0.16666666666666666
- 11.3 the idea is to define a string with a "placeholder" and use an operator that inserts a value into the string
- 11.4 a placeholder is a percent sign followed by characters that specify how the value should be formatted
  - %d means "integer in decimal format", %f means "floating point value", etc
  - see the table on p. 137 of the textbook
- 11.5 the operation that inserts a value x into a string s is written `s % x`
- 11.6 example

```
>>> s = 'The forecast is for %d degrees'

>>> s % 45
```

```
'The forecast is for 45 degrees'

>>> s % 83
'The forecast is for 83 degrees'
```

- 11.7 the format string can contain extra information, *e.g.* about the number of characters to use

```
>>> s = 'The forecast is for %4d degrees'
>>> s % 77
'The forecast is for   77 degrees'

>>> s = 'The forecast is for %-4d degrees'
>>> s % 77
'The forecast is for 77   degrees'
```

- 11.8 for floating point numbers we can specify the total width and the number of digits following the decimal point

```
>>> 'A: %5.2f' % (float(3) / 6)
'A:  0.50'

>>> 'C: %5.2f' % (float(1) / 6)
'C:  0.17'
```

- 11.9 aside: repeat the second expression above, but leave out the parentheses

```
>>> 'C: %5.2f' % float(1) / 6
```

- 11.10 can you explain what happened?

see discussion of "operator precedence" below

- 11.11 change the print statements so they print formatted strings:

```
print('A: %5.2f' % (100*a_count / seq_length))
print('C: %5.2f' % (100*c_count / seq_length))
print('G: %5.2f' % (100*g_count / seq_length))
print('T: %5.2f' % (100*t_count / seq_length))
```

- 11.12 the output now looks like this:

```
[sasquatch:programs] ➤  dnacalc.py
Sequence: ATGAAC
A: 50.00
C: 16.67
G: 16.67
T: 16.67
```

12 Operator Precedence

- 12.1 Python applies multiply and divide operators before add and subtract

```
>>> 2 * 3 + 4
10

>>> 2 * (3 + 4)
14
```

- 12.2 if operators have the same precedence they are applied left-to-right

x % y means "x mod y"

```
>>> 12 % 5
2

>>> 12 % 5 * 2
4
```

- 12.3 what Python does to evaluate x % y depends on what x refers to
  - if x is a number x % y means "the remainder of x divided by y"
- 12.4 if x is a string x % y means "insert y into a placeholder in x"

```
>>> "n = %d" % 6
'n = 6'
```

```
>>> "n = %d" % 6 / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'

>>> "n = %d" % (6 / 2)
'n = 3'
```

- ☐ 12.5 bottom line: if you get an error message, especially one that complains about types of operands, check your assumptions about precedence

☐ 13 Get the Sequence from the Command Line

☐ 13.1 the book shows how to ask the user to type a sequence

- ☐ Python2: call a function named raw_input
- ☐ Python3: the function is called input
- ☐ example:

```
>>> dna_sequence = input('enter a DNA sequence: ')
enter a DNA sequence: AAACCC
```

- ☐ Python prints the prompt, waits for the user to type a line and hit the return key

☐ 13.2 I prefer to get input from the command line or from a file

☐ 13.3 replace the assignment statement that defines `dna_seq` with these lines

```
from sys import argv
dna_seq = argv[1]
```

☐ 13.4 `sys` is a module that defines lots of useful functions that get information from the operating system

- ☐ the `import` statement tells Python we want to use the item called `argv`

☐ 13.5 argv stands for "argument vector"

- ☐ it is a list of items from the command line that started the program
- ☐ `argv[0]` is the name of the program (in this case the string "dnacalc.py")
- ☐ `argv[1]` is the DNA string which the user enters on the command line

☐ 13.6 example

note the sequence on the command line

```
[sasquatch:programs] ➤  dnacalc.py AAACCC
Sequence: AAACCC
A: 50.00
C: 50.00
G:  0.00
T:  0.00
```

☐ 13.7 we'll look at lists in a later lecture

- ☐ what the notation a[i] means
- ☐ how to create lists, add new items
- ☐ how to check to make sure the user didn't leave out the sequence when starting the program

☐ 14 Upper Case Letters

☐ 14.1 if the input sequence has lower case letters they will not be counted

```
[sasquatch:programs] ➤  dnacalc.5.py atcg
Sequence: atcg
A:  0.00
C:  0.00
G:  0.00
T:  0.00
```

- 14.2 Python (and other languages) are very fussy about spelling and capitalization
  - variable names must be spelled and capitalized exactly how they appear when first defined
- 14.3 common technique: "sanitize" the input by converting all the letters to upper case
- 14.4 add a call to the method named `upper`:

  the second assignment updates the value of dna_seq

```
dna_seq = argv[1]
dna_seq = dna_seq.upper()
```

- 14.5 or you can call `upper` in the first statement:

```
dna_seq = argv[1].upper()
```

15 Challenge

- 15.1 what do you think this program will do if the input contains a letter that is not A, C, G, or T?
- 15.2 for extra credit on Project 2 decide how you want to handle non-DNA letters and modify your program so it implements your solution

16 Snippets

- 16.1 as you learn new concepts add them to your "electronic lab notebook"
- 16.2 if you're using a "snippet" app (e.g. Quiver on Mac OS X) add code and notes
- 16.3 examples: argv, format strings, print statements

17 ★★ Incremental Development

★★

- 17.1 notice how this program was developed: we started with a simple version and gradually added more complexity
- 17.2 test each version!
  - if you test after you make a change you will catch errors while they are still "fresh"
  - you'll know where the errors are and be able to find them more easily
- 17.3 contrast this with a process that leads to an entire program being typed in without testing
  - odds are there are several errors (typos, missing parentheses, logic errors, …)
  - tracking them down and fixing them is much harder
- 17.4 use the incremental development process in ALL your projects!
  - start by copying the basic program outline from your snippet editor
  - add code that gets arguments from the command line
  - start adding the main logic, piece by piece
- 17.5 related terms
  - agile development (specs change, goals change as a project matures)
  - continual testing
  - "extreme programming" (especially when combined with pair programming)
  - see "iterative and incremental programming" at Wikipedia