Lab 2: Functions, Sequences, and Strings

CSE/IT 107

"Only ugly languages become popular. Python is the one exception"

— Donald Knuth

---

1 Introduction

The purpose of this lab is to introduce you to the bread-and-butter of programming – functions and strings.

Reading

Chapters 5 and 6 in Think Python: How to Think Like a Computer Scientist (Think)

PEP-8 Style Guide for Python Coding `http://www.python.org/dev/peps/pep-0008/`

Coding Conventions

Your code must follow PEP-8 recommendations.

Problems

Make sure your source code files are appropriately named. Make sure your code has a main function; use `boilerplate.py` you created in Lab 1.

1. Write a program that finds the area of a disk (circle).

   Name your source code `disk.py`

2. Write a program that finds the area of a washer.

   Name your source code `washer.py`

3. Think Exercise 3.3

   Name your source code file `think_3-3.py`

   Test `right_justify()` with the following three strings:

   ```
   To be, or not to be, that is the question:
   Whether 'tis Nobler in the mind to suffer
   The Slings and Arrows of outrageous Fortune,
   ```

   Make sure `'tis` is printed, not `tis`. Strings in Python are sequences of characters enclosed in single or double quotes. Unless you are using a single quote inside the string (e.g. `don't`) most people use single quotes for strings. You just have to be consistent. If you start the string with single (double) quotes you have to end it with single (double) quotes. Try the following in IPython and note the use of single and double quotes (note the standard Python interpreter prompt >>> is just for the author's convenience)

   ```
   # it doesn't matter if you enclose strings in single or double quotes
   # if the string doesn't use a single or double quote itself
   >>> s = 'the cat in the hat'
   >>> print s
   the cat in the hat
   >>> s = "the cat the hat"
   >>> print s
   the cat the hat

   # to use a single quote in a string (apostrophe),
   # enclose the string in double quotes
   >>> s = "I'm going home"
   >>> print s
   I'm going home

   # to use double quotes in a string,
   # enclose the string in single quotes
   >>> s = '"Begin at the beginning," the King said.'
   >>> print s
   "Begin at the beginning," the King said.
   ```

4. Think Exercise 3.4

   Name your source code file `think_3-4.py`

5. Think Exercise 5.3

   Name your source code file `think_5-3.py`

   Spam! Spam! and Spam! if you find a counter-example to Fermat's Last Theorem!

   Section 5.11 in Think Python discusses keyboard input.

6. Think Exercise 5.4

   Name your source code file `think_5-4.py`

7. Think Exercise 6.5

   Name your source code file `think_6-5.py`

   See the "Table of Values" section of `http://en.wikipedia.org/wiki/Ackermann_function` before testing this functions with values of your own.

   What happens when you call `ack(4, 1)`?

   Once you have the function working, add the following lines so you can input m and n from the command line:

   ```
   # add to the beginning of the file
   from sys import argv

   #add to main()
   m = int(argv[1])
   n = int(argv[2])
   ```

   Command line arguments are strings. The method `int()` converts the string to an integer.

   You can run the program in IPython like so:

   `run think_6-5.py 4 1`

   where 4 and 1 are command line arguments.

8. Think Exercise 6.6

   Name your source code file `think_6-6.py`. Do not name it `palindrome.py`

9. Think Exercise 6.8

   Name your source code file `think_6-8.py`

   Test values for your program:

   gcd(46332, 71162) is 26.
   gcd(128, 32) is 32.
   gcd(100 101) is 1. 100 and 101 are relatively prime.
   gcd(97, 97) is 97.

Strings and Functions

As mentioned earlier, strings in Python are sequences of characters enclosed in either single or double quotes. At times you want to refer to the whole string. For example,

```
>>> s = 'the cat in the hat'
>>> print s
the cat in the hat
```

Other times you want to be able to refer to individual characters within a string. To access individual elements of a string you use an *index*. The index starts at 0, the first character in the string, and goes up to `len(string) - 1`, the last character in the string. The function `len(s)` returns the number of characters in the string. Try

```
>>> print s[0]
t
>>> print s[len(s) - 1]
t
```

You will get an IndexError if you try to use an index that is greater than or equal to the length of the string. Try

```
>>> print s[len(s)]
IndexError
Traceback (most recent call last)
<ipython-input-74-58463bd73c4f> in <module>()
----> 1 print s[len(s)]

IndexError: string index out of range
```

Why does `s[len(s)]` fail? Because indexing starts at zero.

To print individual characters of a string, you can place them in a for loop:

```
>>> for i in range(len(s)):
...     print s[i]
```

The `for i in range(len(s))` is a common idiom if you have to access individual string elements. The `range(len(s))` creates a list of numbers from 0 to len(s) - 1, which serves as the index. The `in` statement assigns i to the first value in the list (i = 0). The body of the for loop is executed, print s[0]. i is assigned the next value in the list (i = 1), the body is executed, etc. The for loop repeats until there are no more elements in the list.

Using for loops and the string method `join`, you can create strings. Download `bio_strings.py` from Moodle. Run the program. The program generates and prints a string which represents a randomly generated sequence of DNA. The A stands for the nucleobase adenine, found in DNA and RNA. The C stands for the nucleobase cytosine, found in DNA and RNA. The G stands for the nucleobase guanine, found in DNA and RNA. The T stands for thymine, found in DNA only. For RNA, uracil replaces thymine. Uracil is abbreviated as U.

Lets go over the code that generated the string in `bio_strings.py`, line by line:

```
1 def create_random_dna(n):
2     """Return a random string of dna of length n"""
3     s = str()
4     for i in range(n):
5         s += s.join(('ATCG')[randint(0, 3)])
6
7     return s
```

Line 1 is the function definition. The function takes one formal parameter, n.

Line 2 is known as a *docstring*. It starts and ends with three double quotes. Python uses docstrings as a means to document functions. If you add the line `print create_random_dna.__doc__` to the main function, it will print the `create_random_dna`'s docstring. `doc` is preceded and succeeded by two underscores.

Line 3 creates an empty string. This is needed so we can use `join` in line 5.

Line 4 creates a list of numbers from 0 to n - 1. The `for` loop walks `i` though the numbers in the list. Note that we are not using i as an index; it is just a counter.

Line 5 creates the new string. A lot is going on in this line. The first thing that happens is that an individual element of the string 'ATCG' is selected at random by the code `('ATCG')[randint(0, 3)]`. The function `randint(0, 3)` returns a random integer between $[0, 3]$ and acts as the index into the string 'ATCG'. Try this on the Python interpreter. You have to import `randint` from Python's random library. That is what the line `from random import randint` does in the file `bio_strings.py`.

The code `('ATCG')[randint(0, 3)]` is the parameter to the string's `join` method. From `join`'s docstring (or `help(join)`):

```
S.join(iterable) -> string

Return a string which is the concatenation of the strings in the iterable.
```

Strings are iterable (you can walk through the string element by element), so `join` is just a way to concatenate two strings (join two strings together). The operator `+=` is just shorthand for `t = t + join(...)`. So the `join` keeps appending a single base-pair to a string of length i + 1. To see this, add a `print s` after the join statement. Make sure it is inside the body of the `for` loop.

Line 7 returns the new created string.

Biological Functions

For the rest of the lab, you are going to add and test functions to `bio_strings.py` that are useful for studying the central dogma of molecular biology: DNA makes RNA makes protein. This is part of a field known as bioinformatics.

For all functions you write, assume they return strings that consist of capital letters.

Your main function should just test the functions, printing the original string and the transformed string. For instance

```
def main():
    s = create_random_dna(9)
    t = reverse(s)
    print 's = ', s
    print 'reverse of s =', t
```

10. Add the function

    ```
    def create_random_rna(n):
        """Return a random string of rna of length n"""
    ```

    Your function should generate a random sequence of RNA. RNA has base-pairs of A, U, C, and G.

11. Add the function

    ```
      def reverse(s):
        """Return a new string that is the reverse of string s"""
    ```

    For example: if s = 'ATCG', the new string returned is 'GCTA'

    The `range()` function takes 3 parameters: start, stop, step. For instance to create a list of numbers from 0 to 100 counting by 5, use `x = range(0, 100, 5)`. The `range()` function can also generate a sequence (list) of descending numbers.

    At the Python interpreter, try the following:

    ```
    >>> x = range(10, 0, -1)
    >>> print x
    ```

12. Add the function

    ```
      def complement(s):
        """Return a new string that is the DNA complement of string s"""
    ```

    DNA is double stranded, but you only need to know one of the strands when you model DNA behavior as the complement of A is T and vice versa, and the complement of C is G and vice versa.

    If s = 'ACCGTT', the complement is 'TGGCAA'.

13. Add the function

    ```
    def reverse-complement(s):
      """Return a new string that is the DNA reverse-complement of string s"""
    ```

    For example, if s = 'ACCGTT', the reverse complement is 'AACGGT'

    Use the reverse and complement functions you just wrote.

14. Add the function

    ```
    def transcription(s):
      """Return a new string that is the transcription (mRNA) of string s"""
    ```

    mRNA, messenger RNA, is the complement of DNA with U replacing T.

    For example, the transcription of string 'ATTCCGGCT' is 'UAAGGCCGU'

    In later a lab we will finish off the central dogma of biology and convert mRNA into proteins (translation).

15. Add the function

    ```
    def print_codons(s):
      """Prints codons as a space separated list contained in the string s"""
    ```

    Codons are groups of 3 base pairs (e.g. AUC, GCT, etc). The base pairs can either be DNA or RNA. Codons specify a single amino acid. Proteins are chains of amino acids.

    If s = 'TCCGACTAA', your function should print three codons separated by a space: TCC CGA TAA

    Make sure to check that the string length is a multiple of three. Print out codons only if the string length is a multiple of 3. Otherwise, print an error message if the string is not a multiple of 3.

    Try the following code in the interpreter. Note that the index `i + 1` is an expression that gets evaluated.

    ```
    >>> s = 'the cat in the hat'
    >>> for i in range(0, len(s), 2):
    ...     print '{0}{1}'.format(s[i], s[i + 1])
    ```

    Here the print statement is using the string format method. The braces indicate a placeholder, {n}, where n is a number that gets replaced by a positional argument supplied to format.

    Now add a comma at the end of the print statement. Adding a comma to the end of print statement causes the newline to be omitted.

16. Add the function

```
def count(s, c):
    """return the count of the number of times c, a character,
       appears in string s"""
```

    Technically, Python doesn't have a character type. If you want a characters in Python, use a string of length one. For example, `c = 'A'` behaves like a character, albeit its data type is a string.

    The ability to count is fundamental in computer science. For example the following code, counts the number of times the for loop body executes:

```
>>> count = 0
>>> for i in range(10):
...     count += 1
...
>>> print count
```

    The statement `count += 1` is shorthand for `count = count + 1`

    Do not use Python's builtin count method.

17. Add the function

```
def gc_content(s):
    """return the percentage of g and c base pairs in the string s"""
```

    The GC content or GC ratio in DNA can be calculated as

    $\frac{count(G)+count(C)}{count(A)+count(T)+count(G)+count(C)} \times 100.$

    To convert an integer to a float point type, use the `float()` method.

    Among other things, the GC content of DNA has been used to classify bacteria into high-level groups.

18. Once you have tested your functions with strings of small length, test your functions with strings that are much larger.

Submission

Create a tarball of your *.py source code.

```
tar czvf cse107_firstname_lastname_lab2.tar.gz *.py
```

To check the contents of your tarball, run the following command:

```
tar tf cse107_firstname_lastname_lab2.tar.gz *.py
```

You should see a list of your Python source code files.

Upload your tarball in Moodle before the start of you next lab.