Problem Set 3, CSCI1101, Sections 1—3, Fall 2017

Due: at 6PM, Friday, October 6th 2017, on GitHub.
Number of pages: 5

Note on the format of your submission: Each one of your files must start with the following lines, followed by your program/code (please don't disable pylint conventions anymore and try to resolve all the pylint errors/warnings/conventions in your files before submitting your work):

```
# Problem Set 3, Part I (if the file is for Part I); otherwise, Part II.
# Name: your last name, your first name
# Collaborators: last name1, first name1; last name2, first name2; etc.
```

**Feel free to collaborate with anyone you want (or post your questions to Piazza). However, when it comes to sitting down and writing your code, please write your own code. Thank you.

**If you encounter any problems in completing the assignment or in the submission process, please don't hesitate to ask for help (come to the office hours).

****NOTE**: In this problem set, you may **not** use the **`in`** operator (for verifying if a substring appears in a string) or call any string methods provided by Python, except the **`lower`** method. You MUST use **LOOPS** and **INDEXING**.

Part I: In your repository, first please create a new file, named ***pattern.py***.

In this part, we will do some basic pattern matching (similar to what is done in DNA testing, however on a much smaller scale). String-matching is an important subject in the wider domain of text processing. String-matching algorithms are basic components in the implementations of practical software packages existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. String-matching consists of finding at least one of the occurrences of a string (more generally called a pattern) in a text (a bigger string). To do this, you need to consider two strings: a pattern and a text. The text is n characters long, while the pattern (the smaller string) has a length of m characters where $m \leq n$. Both strings consist of a finite set of characters, limited to the standard letters in the English alphabet.

The brute-force algorithm (which you should implement) consists in checking, at all positions (indices) in the text between 0 and $n - m$ (why stop at $n - m$?), whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

Let's look at an example:

Brute-force example:

TEXT

```
G C A T C G C A G A G A G T A T A C A G T A C G
```

PATTERN

```
G C A G A G A G
```

First attempt to find a match

```
G C A T C G C A G A G A G T A T A C A G T A C G
1 2 3 4
G C A G A G A G
```

Second attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
  1
  G C A G A G A G
```

Third attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
    1
    G C A G A G A G
```

Fourth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
      1
      G C A G A G A G
```

Fifth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
        1
        G C A G A G A G
```

Sixth attempt

```
G C A T C G C A G A G A G T A T A C A G T A C G
          1 2 3 4 5 6 7 8
          G C A G A G A G
```

A Match Has Been Found!!!!

Now, what will your program do? First, you will define a function, called `match`, that implements the simple brute-force pattern matching described above. **This function takes two strings (the text string and the pattern string) as its arguments, and returns `True` if the pattern string matches some portion of the text string; otherwise, the function returns `False`.** Test your function with several test pairs to make sure it's working correctly.

Then, define a main function. Your main function must **keep asking the human user for a new text string and then a new pattern string and print a nice message that states whether the pattern was found in the text or not** (something like "Yay, a match is found" or "Too bad no match", please be clever). **The program stops when the user enters `qqq` as the text string** (your program must stop regardless of the case of the letters; so, any of the inputs `qqq`, `qqQ`, `qQQ`, `Qqq`, etc. must stop the program). Please have a call to the main function in your main program.

Part II: In your repository, create a new file, named ***encoding.py***. We will define two functions in this file (one function corresponds to Part A below and the second function will be the solution to Part B below).

In Part A below, we create a message encoder, and in Part B, we write the corresponding decoder.

A. In ***econding.py***, write a function, named `encoder`, that takes in a string and encodes it based on the following encoding rules: All the occurrences of **a** in the string are replaced with **b**. All the occurrences of **b** in the string are replaced with **a**. All **e** occurrences in the string must be replaced with #. After making all these character substitutions, the encoder changes the order of the characters in the string based on the length of the string:

- If there are an **even number of characters** in the string, the encoder breaks the string into two halves with equal lengths and puts the second half before the first half to create the final encoded message. For example, assume that, after making all the substitutions, the string is $c_0c_1c_2c_3c_4c_5c_6c_7$ (there are 8 characters in this string). The encoder puts the last 4 characters before the first 4 characters, hence, $c_4c_5c_6c_7c_0c_1c_2c_3$ is the final encoded message.
- If there are an **odd number of characters** in the string, say $\mathbf{2n+1}$ **characters**, the encoder **keeps the middle letter (character)** and swaps the first and last n characters to create the final encoded message. For example, assume that, after making all the substitutions, the string is $c_0c_1c_2c_3c_4c_5c_6c_7c_8$ (there are 9 characters in this string). The encoder keeps the mid character, c_4, and swaps the first and last 4 characters; therefore, $c_5c_6c_7c_8c_4c_0c_1c_2c_3$ is the final encoded message.

The `encoder` function returns the final encoded string (message). **We assume that the original string does not contain the character & (look at your keyboard, this is SHIFT+7) or the character # (SHIFT+3)**. However, the string may contain any other characters. Also, we don't have to worry about the case of the letters in the message because we assume that all letters in the input string are lowercase.

Hint: you might find it helpful to write a separate function to replace all the occurrences of a given character with another given character in a string and call that function in your `encoder` function a number of times.

Some examples:

```
Original word: banana
Encoded word: bnbabn
Original word: boston college
Encoded word: coll#g#aoston (this has a whitespace at the end)
Original word: melissa
Encoded word: ssbim#l
Original word: ram
Encoded word: mbr
Original word: how are you today?
Encoded word: ou todby?how br# y
```

B. In the same ***econding.py*** file, define another function, named `decoder`, that accepts an encoded string (that has been encoded based on the rules described in **Part II.A** above) as its argument and decodes the encoded string to obtain the original string. The `decoder` function must <u>return</u> the decoded string (message).

**Please test both of your functions with several different strings to verify they're working correctly. However, please do not print anything. We will test your functions using our own test strings.

Please note that, in the main program, the chained function call `decoder(encoder(strin))` must evaluate to `strin` for any string `strin` that contains only lowercase letters and doesn't contain **& or #. So, make sure to test this composed (chained) function call with several values for `strin` to verify if both of your functions are working as expected.