CS1110 Spring 2016

Assignment 5: Due Friday April 8 at 6pm

Clarification and typo correction, Sunday Apr 3 3:45pm

You must work either on your own or with one partner. If you work with a partner, you and your partner must first register as a group in CMS (this requires an invitation issued by one of you on CMS and the other of you accepting it on CMS) and then submit your work as a group. As mentioned in class, we strongly recommend against a "divide-and-conquer"/Henry-Ford-assembly-line approach: each member of a group should work on each part of each problem to get the full educational value out of the assignment.

You may discuss background issues and general solution strategies with others outside your CMS group, but the program(s) you submit must be the work of just you (and your partner). We assume that you are thoroughly familiar with the discussion of academic integrity that is on the course website. Any doubts that you have about "crossing the line" should be discussed with a member of the teaching staff before the deadline, *and you should also document such situations as comments in the header of your submission file(s).*

Topics. Lists of strings. Lists of numbers. List methods. Return-from-loop body. More intricate boolean/string computations. A `main` function. Reading data from a text file. Before you begin, check out Lab 6 and (when available) Lab 7 and lectures 3/17, 3/22, and 3/24.

1 Background

I'm shopping for colleges and use Google to search for "Corenell." The response is "Do you mean cornell?" I'm texting my spouse with the message "Honey I'm Home"[1] but it goes out as "Money I'm Home." I'm checking out some sequences of nucleotides and want to know how close a match 'AGTCGTC' is to 'TAGTCGT'.

Each of these examples points to an underlying "nearness" problem that involves strings. The big question that we will explore in this assignment is this: When might we regard one string as being close to another? Or better, when might we regard one string as a "neighbor" of another?

Here are three possible definitions of string neighbors based on how Professor Fat Fingers makes mistakes when texting:

- Two strings are neighbors if they are the same except in one position. (`"abc"` and `"abe"`)
- Two strings are neighbors if they are different but we can obtain one by swapping an adjacent pair of characters in the other. (`"abc"` and `"acb"`)
- Two strings are neighbors if deleting one character from one string produces the other string. (`"abc"` and `"abxc"`)

2 Getting Set Up

Download and unzip `A5.zip` into a folder/directory called (for example) `A5`. It contains a text file called `EnglishWords.txt` and a module called `GetData.py` (Lab 7 and the 3/24 lecture are useful here), as well as a skeleton file `CloseWords.py`.

You will be submitting a revised version of `CloseWords.py` to which you have added functions and code that showcases your implementation work.

3 Three Definitions of "Neighbor"

The first order of business is to develop three Boolean-valued functions each of which checks whether two strings are neighbors or not. They correspond to the "definitions" given above.

[1] Why I would text my spouse this, as opposed to speaking out loud like in the old days, is an issue we will not address here.

3.1 Off-By-One

Two non-empty strings `s1` and `s2` are said to be *off-by-one* if they have the same length and differ in exactly one position. Here some examples:

s1	s2	Off-By-One?
"read"	"rexd"	True
"read"	"xexd"	False
"read"	"readx"	False
"read"	"eadx"	False
"a"	"x"	True
"a"	"a"	False
"a"	"A"	True

Implement a function `offByOne(s1,s2)` that takes two nonempty strings `s1` and `s2` and returns `True` if they are off-by-one and `False` otherwise.

3.1.1 Optional but potentially very helpful: Ideas for how to test

If you haven't tried this already, one way to test your function is to get into Python interactive mode, and do:

```
>>> import CloseWords
>>> print CloseWords.offByOne("read", "rexd")
```

and similarly for all the test cases we've given you above, and any others you think might be good to try out. But remember: every time you change your code in your editor, you need to exit interactive mode and then re-enter it in order for Python to "see" your changes. *Or*, in Python 2, with some caveats, you can do

```
>>> reload(CloseWords)
```

after making changes.

Alternatively, you can make a testing function that loops through test cases, like what you may have noticed appeared in the practice-prelim solution code:

```
def testem():
    # test offByOne
    for answer in [["read", "rexd", True],
                   ["read", "xexd", False],
                   ["read", "readx", False],
                   ["read", "eadx", False],
                   ["a","x", True],
                   ["a", "a", False],
                   ["a", "A", True]]:
        response = offByOne(answer[0], answer[1])
        right = answer[2]
        if response != right:
            print "problem with", answer
    print "done testing off-by-one"
```

Then, in your Application Script, you can temporarily put in a call to `testem`.

For your convenience, `testem` has been provided in the skeleton `CloseWords.py`. You can modify it any way you want, including adding tests for other functions you write later in this assignment; we won't be looking at `testem` when grading.

Finally, a really effective, if non-obvious, strategy is to think up test cases on paper and *before* you write your code; try, just from reading the specification, to create test cases that cover all the "types" of inputs you could get. If nothing else, this activity helps you figure out the scope of the problem your code needs to solve. Also, it's incredible hard to think about cases your code doesn't handle once you've written your program: unfortunately, "most code tends to look right".

3.2 Off-By-Swap

Two non-empty strings `s1` and `s2` are said to be *off-by-swap* if `s1` and `s2` are different, but `s2` can be obtained by swapping two adjacent characters in `s1`. Here some examples:

s1	s2	Off-By-Swap?
"read"	"raed"	True
"read"	"erad"	True
"reaxd"	"read"	False
"read"	"erda"	False # two swaps
"read"	"erbx"	False # one swap, but other changes too
"x"	"Y"	False
"aaa"	"aaa"	False # same strings

Implement a function `offBySwap(s1,s2)` that takes two nonempty strings `s1` and `s2` and returns `True` if they are off-by-swap and `False` otherwise.

We strongly suggest you think of other test cases to try beyond those we have supplied you, and we make no guarantees that the test cases we've provided cover all the possible types of inputs.

3.3 Off-By-Extra

Two non-empty strings `s1` and `s2` are *off-by-extra* if removing one character from `s1` gives `s2` or if removing one character from `s2` gives `s1`. Here are some examples:

s1	s2	Off-By-Extra?
"abcd"	"abxcd"	True
"abxcd"	"abcd"	True
"abcda"	"abcd"	True
"abcd"	"bcda"	False
"abcd"	"abcdef"	False
"abcd"	"abcd"	False

Implement a function `offByExtra(s1,s2)` that takes two nonempty strings `s1` and `s2` and returns `True` if they are off-by-extra and `False` otherwise.

We expect you to implement this Boolean-valued function with a helper function `ListOfDeleteOnes(s)` that takes a nonempty string `s` of letters and returns a list of all those strings that can be obtained from `s` by deleting one character, e.g.,

```
"abcd"  ----> [ "bcd", "acd", "abd", "abc" ]
```

Note that if `w` is a string and `w in ListOfDeleteOnes(s)` is `True`, then `w` and `s` are off-by-extra.

And to repeat, your implementation of `offByExtra` *must* make effective use of `ListOfDeleteOnes`.

3.4 Testing

Thoroughly test your implementations of `offByOne`, `offBySwap` and `offByExtra` before proceeding. You are free to implement additional helper functions if you wish. Make sure that they are fully specified.

4 Computing a List of Neighbor Words

In the next part of the assignment you write a pair of functions that return lists of strings. We need to define what we mean when we say that one string is a neighbor of another:

> Two strings `s1` and `s2` are *neighbors* if they are not the same string and they are either off-by-one, off-by-extra, or off-by-swap.

Implement the following two functions so that they perform as specified:

```
def ListOfNeighbors(s,L):
    """ Returns a list of all entries in L that are neighbors of s.

    PreC: s is a nonempty string and L is a list of nonempty strings.
    """

def ListOfNeighbors2(s,L):
    """ Returns a sorted list of all entries in L that are
    neighbors of neighbors of s. The returned list has no
    duplicate entries.

    PreC: s is a nonempty string and L is a list of nonempty strings.
    """
```

Clarification: when we talk about the entries in `L` that are neighbors of neighbors of `s`, we mean the entries in L that are neighbors of something ***in** `L`** that is a neighbor of `s`. See the example given in section 4.1.
The Python functions `list` and `set` can be used to remove duplicate entries from a list:

```
>>> x = [20,20,10,10,10,40,40,30,30,30,30]
>>> x
[20, 20, 10, 10, 10, 40, 40, 30, 30, 30, 30]
>>> y = list(set(x))
>>> y
[40, 10, 20, 30]
>>>
```

Warnings: *your code should not alter the input list `L`*. Note: it is legal for `L` to be the empty list.

4.1 Testing/debugging

One way to debug `ListOfNeighbors` and `ListOfNeighbors2` is to temporarily use a "dictionary" that is smaller than `EnglishWords.txt`. You can either do this "by hand", e.g.,

```
smallList = ["abc", "abcd", "abcde"]
```

or by pulling out a small slice from the list `L` above, e.g.,

```
smallList = L[1000:1050]
```

Another very small test case is that, for the full dictionary `EnglishWords.txt`, the only neighbor of "transform" is "transforms", and the only neighbor of its neighbors is "transform" again. An additional example is "largest", which has two neighbors, "larges"[2] and "largess"; there are 11 neighbors of neighbors.

In any case, make sure your implementations are correct before proceeding.

5 Scrabble

In the game of Scrabble there are 98 *letter tiles*. Each of the 26 letters has a certain value and there are a specific number of each letter tile. These strings and lists[3] capture it all:

```
Letters = string.ascii_uppercase
        #A B C D  E F G H I J K L M N O P  Q R S T U V W X Y  Z
Supply= [9,2,2,4,12,2,3,2,9,1,1,4,2,6,8,2, 1,6,4,6,4,2,2,1,2, 1]
Value = [1,3,3,2, 1,4,2,4,1,8,5,1,3,1,1,3,10,1,1,1,1,4,4,8,4,10]
```

[2] As in someone at a pizzeria saying, "I'll buy two of the larges, please".

[3] Thought question: can you figure out why we made `Supply` and `Value` to be lists, not strings? Would it have been better or worse to make `Letters` a list instead of a string?

For example, there are 4 "D" tiles and each one is worth 2 points. Note that (originally, there was a typo where the text below said that "`i` will have value 4")

```
i = Letters.find('D')   # i will have value 3
numberOfLetter = Supply[i]
valueOfLetter  = Value[i]
```

assigns 4 to `numberOfLetter` and 2 to `valueOfLetter`. The *Scrabble score* of a string of uppercase letters is the sum of the individual character values, e.g.,

```
"ZOOLOGY"  ----->  10 + 1 + 1 + 1 + 1 + 2 + 4 = 20.
```

But there's *one important exception*: the Scrabble score of a string of letters that requires more of some letter tile than is available is 0. An example is `"ZYZZYVA"`: two too many "Z"s.

Implement a function `ScrabbleScore(s)` that takes a string of letters (of any case or mixed case), converts all the characters to uppercase, and then returns the Scrabble score of the result (as an `int`). We've given you some skeleton code for `ScrabbleScore` to save you some typing.

6 The Final Application Script and `main` function

After all your functions are working, change the application script so that *all it does* is call a function called `main()`, which takes no arguments. This function must be based on the following skeleton *and*, when completed, obey the desiderata given after the skeleton.

```
def main():
    """Showcases the code in this module."""

    # Read in the list of English words...
    L = fileToStringList('EnglishWords.txt')

    # Until the user complies, keep prompting for a string of lowercase letters.
    while True:
        s = raw_input('Enter a nonempty string of lowercase letters: ')




    # Print all the neighbors of s and their Scrabble scores...
    print '\nNeighbors...\n'



    # Print all the neighbors of the neighbors of s and their Scrabble scores...
    print '\nNeighbors of the Neighbors...\n'
```

Reminder: when we talk about the entries in `L` that are neighbors of neighbors of `s`, we mean the entries in L that are neighbors of something ***in** `L`** that is a neighbor of `s`. See the example given in section 4.1. The `while` loop body should break as soon as the string `s` is made up entirely of lowercase letters; use a `break` statement appropriately to accomplish this. It is not necessary to pretty print the output, i.e., a statement like this:

```
print theString, theScrabbleScore
```

is an absolutely fine way to display a neighbor string and its Scrabble score.

In checking whether letters are lowercase, it's convenient to consult the string `string.ascii_lowercase` (as opposed to typing in `'abcdefghijklmnopqrstuvwxyz'`, and assuming you've imported module `string`).

7 CMS

One file should be uploaded to CMS: `CloseWords.py`. It must contain functions named *exactly* the following, and taking exactly the number of parameters described in the text above. (Any changes in capitalization, spelling, number of parameters, etc. will cause an error in the auto-testing code and result in point deductions for you, because we're going to call the functions by exactly the names we've listed below and with exactly the number of parameters described above. Of course, you can name the *parameters* anything you want.[4])

`offByOne`

`offBySwap`

`offByExtra`

`ListOfDeleteOnes`

`ListOfNeighbors`

`ListOfNeighbors2`

`ScrabbleScore`

`main`

Your `CloseWords`.py may also include any (documented-via-docstrings) helper functions that are part of your overall implementation. And, your "final" Application script should contain only a call to `main`.

Reminders about submission and CMS:

1. Make sure your submitted `.py` files begin with header comments listing:

   (a) The name of the file

   (b) The name(s) and netid(s) of the person or group submitting the file

   (c) The date the file was finished

   If there were other people who contributed to your thought processes in developing your code, mention them by name in header comments. (You don't have to mention course-staff members, although it makes sense to do so.)

2. Make sure all your functions have appropriate docstrings. These should include explanations of what the parameter variables "mean" and what preconditions (constraints) are assumed on their values, and what the user can expect as a result of calling your functions.

3. Do not "reload"/"refresh" the CMS assignment submission page after the submission deadline passes: this will trigger another upload of your files, which can result in your submission being counted as late. The safest policy is to close your browser tab/window after upload.

4. If you submit earlier than two hours before the submission deadline, CMS allows you to download your files to check that they are what you think you submitted. We thus strongly suggest that you submit a version early and do this check. (If you want to make changes, no problem: CMS allows you to overwrite an older version by uploading a new version.)

[4] Within reason. We reserve the right to deduct style points for parameter names that are inappropriate, e.g., offensive. Generally, one should choose parameter names indicative of the information they are meant to store.