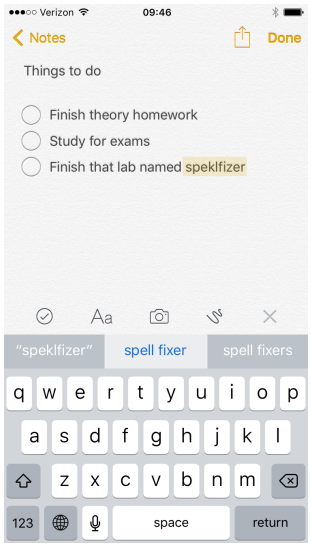
Computational Problem Solving CSCI-603
Spelling Correction Lab 4

6/28/2021



1 Introduction

You will be using string manipulation and built-in data structures to develop a simple spelling error correction system.

2 Problem Solving Session

You will work in a group of three or four students as determined by the instructor. Each team will work together to complete the following activities.

1. Make a list of five *general* types of errors you believe users commonly make when entering text on a keyboard in a language that uses an alphabet without accented letters or other special marks.

2. For this lab assignment, you will correct errors that occur because a user's finger accidentally hit a key adjacent to the intended one *instead of* the intended one. For example, "laboratory" might be entered as "labo5atory" because the `5` key is adjacent to the `r` key on an English QWERTY keyboard.

List any fixed data resources or references your program would need to accomplish correction of the kind of error described above, once a language and keyboard layout are chosen. That is, what kind of standard data would you need to reuse every time your program runs? Provide an example of the information contained on each data resource.

3. Describe, **using pseudocode** , an algorithm that takes a single word (no white space or punctuation in it) as a parameter and returns a potentially different word that exists in the language and differs in at most one letter according to the error described above.

4. What is the big-O time complexity of your algorithm given a word of length `n`?

5. Assume that your program reads one line of text at a time. Make a list of useful string operations that would split a line of text into its component words, free of white space or punctuation.

At the end of problem-solving, verify your work with either the instructor or SLI.

3 Implementation

You will create a program called **spellfixer.py** which reads text from the standard input. As stated above, your program will correct text input errors that occur because a user's finger accidentally hit a key adjacent to the intended one instead of the intended one.

Adjacent is defined as one of the six keys surrounding the given key, not including the space bar or meta keys. For example,

- 3, 4, R, D, S, and W are adjacent to E.
- Z, S, D, and C are adjacent to X.

A *word* here is a substring of an input line that

1. is non-empty,
2. was surrounded by white space and/or was at the start or end of the line,
3. has had any punctuation characters at the start or end of the word removed.

Tip: Having one person on the team who is familiar with regular expressions could make things go more smoothly.

3.1 Legal Words

To make the correction, your program will use two data resources:

- A list of standard English words (file words.txt). Each line in the file is a valid word.
- A list with the distribution of keys in a 104-key PC US English QWERTY keyboard[1] (file keyboard.txt). Each line in the file is a letter and its adjacent letters in the keyboard.

These files are provided to the program on the command line. If they are present, their content are guaranteed to all be valid. If not present, you should print the usage statement and exit:

```
Usage: python3 spellfixer.py {words-filename} {keyboard-filename}
```

3.2 Program Output

The characters (punctuation, whitespace) that were found between words and removed via the criteria above are echoed to standard output unchanged. Newlines are also echoed unchanged.

Any word that is found in the word list is copied unchanged to standard output.

For each word not found in the word list, check if changing a character for one of its neighboring *alphabetic* characters on the standard English QWERTY keyboard produces a legal word. If *one* such change produces a legal word, that revised word is copied to

---

1. Did you know the letter layout was designed to slow your typing down so that you would not jam the typewriter typebars? `https://en.wiktionary.org/wiki/typebar`

standard output. If no such single change produces a legal word, the original word is copied to standard output as if it were legal. The same thing happens if one of the characters in the word is not a neighbor to any English letter on the keyboard.

The comparison between the user input and the words in the given **words** *file must be case insensitive.*

Here is an example of the program execution.

```
$ python3 spellfixer.py words.txt keyboard.txt

Welcome to Spell Fixer
> this line is good
this line is good

> thks linw can bc f8xed
this line can nc fixed

> doj0urrr
doj0urrr

> Jiacofumo dodn't have a clu about the prlblem.
Jiacofumo dodn't have a flu about the problem.

> !*!
Bye!
```

In summary, your program should read in a loop its user input from standard input, and write the corrected text to standard output. The input '!*!' must terminate your program gracefully.

3.3 Implementation Instructions and Help

3.3.1 Additional Requirements

Besides the above requirements, your program must apply **two** other types of spelling corrections from the list below. Those corrections will perform one simple edit to fix a misspelled word of distance 1.

- Transpose two adjacent letters (e.g. assingment → assignment)
- Replace one letter for another (e.g. peaple → people)
- Add a letter (e.g. asist → assist)
- Remove a letter (e.g. lettter → letter)

Your program will try to fix first finger sliding errors. If no legal word is generated, then, your two new corrections will be applied. The order on which you apply the corrections is up to you. If several legal words are generated after applying the corrections, print the first one in the standard output.

3.3.2 Help

Code has been provided to you that implements a crude version of mutable strings (see `ritcs` folder). You may use it, and even modify it.

Choice of data structures for this lab is crucial. If you start writing a solution and feel that things are too difficult, step back and consider changing how you store information.

Extracting words from the input is much easier if you use the Python package `re`. We recommend you looking at the functions `split`, `search`, and `match`. The documentation on this package also includes a lot of detail on how to create a regular expression.

4 Grading

The assignment grade is based on these factors:

- 20%: attendance at problem-solving and results of problem-solving
- 10%: Good design practices
- 65%: Functionality
  - 10%: The ability to read any input and output something without crashing
  - 10%: Outputs properly spelled words without changes
  - 15%: Corrects simple misspelled words as per the spec in this document
  - 10%: Handles punctuation cases successfully
  - 20%: Corrects misspelled words as per the chosen additional spelling corrections
- 5%: Code Style and Documentation

Note: The submitted file must be a Python 3 program.

5 Submission

Create a plain text file **feature.txt** that states the two additional features you chose to add in your spelling correction scheme and your test cases.

Create a zip file named `lab4.zip` containing your `spellfixer.py` and `feature.txt` files. Submit the zip file to the **myCourses dropbox** by the due date for this lab.