CSC148, Assignment #1
Due February 4th, 2015 at 22:00

Part 1 - An Unmarketable Word Processor [45%]

The goal for this portion of your assignment is to use data structures that we have learned about to create a line editor with several features. Your task is to write a program that accepts lines of text from a user and normally appends them to a "document" in memory (represented as a Python list). All lines should be appended in this way unless they are command lines, which do not indicate text to add, but instead indicate that the user wants to do something with the document. The valid commands are as follows.

- *d*: this command, typed by itself, should allow the user to delete a line in the document. After typing d and pressing enter, the user should be asked which line to delete, and it should be removed. To remove the first line, the user would respond with 1; to remove the second line, the user would type 2, and so on.
- *undo*: this command, typed by itself, should allow the user to undo the last action that was made to the document. The two types of actions that can be undone are the addition of a line to the end of the document, and the deletion of a line of text from anywhere the document. (That's because we don't offer any other functionality.)
- *redo*: this command, typed by itself, should allow the user to redo the last action that was undone. (An undo followed by a redo leaves the document as it was before.). For example, if the user deletes line 3 from the document and issues an undo, line 3 should be placed back in the document. Performing a redo at this point would reverse the undo, again yielding a document with line 3 removed.

Importantly, issuing multiple undo operations in a row should continue to undo changes made to the document, and issuing multiple redo operations in a row should continue to redo changes to the document. This is called multilevel undo/redo, as opposed to one level of undo/redo.

When a blank line is entered by the user, the program should terminate. After each time the user presses enter, the document as it currently stands should be output to the screen, and the user should be prompted for the next command. Your code must be robust in the sense that it should not be possible to crash the program (for example, by trying to delete a non-integer line or a line that does not exist, or trying to pop empty stacks or queues).

Here's an example of program execution.

```
Command?first
['first']
Command?second
['first', 'second']
Command?third
```

```
['first', 'second', 'third']
Command?d
Delete which line?2
['first', 'third']
Command?fourth
['first', 'third', 'fourth']
Command?undo
['first', 'third']
Command?undo
['first', 'second', 'third']
Command?redo
['first', 'third']
Command?
```

Implementing the undo/redo functionality should be achieved using one or more stacks or queues (part of the assignment is figuring out what to use here!). In particular, you should ask yourself the following questions:

- Do undos follow the FIFO or LIFO policy? What about redos?
- When actions have just been undone, it is clear what the redo command should do. However, what should the result of a redo be if the last action was not an undo?
- What information has to be stored so that an undo or a redo can be done successfully?

For this part of the assignment, you should submit a Python code file that we can run to interact with your editor from the keyboard entitled `word_processor.py`. You should also submit a file showing various tests of your editor that lend proof to your code working properly `test_word_processor.py`. Also, remember to comment your code. In particular, you should include a high-level description of how your implementation works.

Part 2 - Managing Counts of Letters [35%]

In this part of the assignment we will continue to practice using (1) lists, (2) classes and (3) exceptions. (Yes, exceptions really are classes, but I really wanted to have three points there.) You are to implement a class called `LetterManager` that can be used to keep track of counts of letters of the alphabet. The constructor for the class takes a String and computes how many of each letter are in the String. This is the information the object keeps track of (how many a's, how many b's, etc). It ignores the case of the letters and ignores anything that is not an alphabetic character (e.g., it ignores punctuation characters, digits and anything else that is not a letter).

Your class should have the following methods:

- A constructor. Given a string, it will construct an inventory (a count) of the alphabetic letters in the given string, ignoring the case of letters and ignoring any non-alphabetic characters.
- *Get.* Takes a letter, and returns a count of how many of this letter are in the inventory. Letter might be lowercase or uppercase (your method shouldn't care). If a nonalphabetic character is passed, your method should throw a user-defined exception having a meaningful name.
- *Set.* Takes a letter and a value, and sets the count for the given letter to the given value. The letter might be lowercase or uppercase. If a nonalphabetic character is passed, your method should throw a user-defined exception.

- *Size*. Returns the sum of all of the counts in this inventory. This operation should be "fast" in that it should store the size rather than having to compute it each time this method is called.
- *IsEmpty*. Returns true if this inventory is empty (all counts are 0). This operation should be fast in that it should not need to examine each of the 26 counts when it is called.
- *__str__*. Returns a String representation of the inventory with the letters all in lowercase and in sorted order and surrounded by square brackets. The number of occurrences of each letter should match its count in the inventory. For example, an inventory of 4 a's, 1 b, 1 l and 1 m would be represented as "[aaaablm]".
- *Add*. Takes another `LetterManager`, and constructs and returns a new LetterManager object that represents the sum of this letter manager and the other given letter manager. The counts for each letter should be added together.
- *Subtract*. Given another `LetterManager`, constructs and returns a new `LetterManager` object that represents the result of subtracting the other letter manager from this letter manager (i.e., subtracting the counts in the other inventory from this object's counts). If any resulting count would be negative, your method should return None.

You should implement this class with a list of 26 counters (one for each letter) along with any other data fields you find that you need.

Once you have created this class, use it to write an Anagram Checker program. This program will prompt the user for two words and report whether the two words are anagrams of one another. Two words are anagrams if they both include the same counts for all their letters. (In other words, one word is just a "scrambled" version of the other.) For example, horse and shore are anagrams since they each include one copy of the letters E, H, O, R and S.

For this part of the assignment, you should submit three Python files: a file that contains your letter manager class (entitled `letter_manager.py`), a file that uses your letter manager class to test whether two words are anagrams of one another (entitled `letter_manager_anagram.py`), and a file containing test cases for both of these parts (entitled `test_letter_manager.py`). Be sure that your tests are comprehensive and test boundary cases, typical cases and atypical cases.

Part 3 - Operations on Stacks [20%]

Sometimes it is convenient to do more with a stack than push and pop the top element. In fact, some computer architectures and programming languages are built around using the stack to drive program execution, so more powerful primitives often make the programmers' life easier. For example, if you are writing a sorting program and all you have available is a stack, it would be helpful if you could swap the top two elements on the stack.

In this part of the assignment, you'll write two higher-level functions on stacks whose implementations are based on the stack primitives (push, pop, and so on). Your functions should work on stacks created with the stack classes of `stack.py`. Download the starter code `stackops-template.py` which includes two functions whose bodies you are to fill in. The first, swap, is designed to swap the top two elements of a stack. For example, if a stack contains the string **red** at the top of the stack, and the string **blue** below it, a swap will result in **blue** at the top and **red** beneath it (the rest of the stack is unchanged). The roll operation is designed to take an integer n along with the stack. We can define roll operationally as follows. First, pull out the nth item from the top of the stack (the top of stack is item 1; the item below it is item 2, and so on). Second, push this removed item onto the top of the stack. For example, consider the stack "1 2 3 4", where the left end is the bottom of the stack and the right end is the top (so the number 4 is at the top of

the stack). A *roll* 2 will yield "1 2 4 3" and a *roll* 3 will yield "1 3 4 2".

For this part, you should submit the Python file where you have filled in the function bodies (entitled `stackops.py`). You should include test cases as well to show us that your stack functions are working properly (entitled `test_stackops.py`).

Marking

Part 1: (45%)
- Correctness (60%)
- Completeness/thoroughness of testing (30%)
- Documentation, Clarity and Coding Style (10%)

Part 2: (35%)
- Correctness (60%)
- Completeness/thoroughness of testing (30%)
- Documentation, Clarity and Coding Style (10%)

Part 3: (20%)
- Correctness (80%)
- Documentation, Clarity and Coding Style (20%)