CS 1110, LAB 6: LISTS; MAP; COMMAND-LINE ARGUMENTS; FILES

http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab06.pdf

Lab materials. Create a *new* directory on your hard drive and download the following files into that directory.

http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab06/unscramble.py
http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab06/unscrambletest.py
http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab06/scowl_utf-8.txt
http://www.cs.cornell.edu/courses/cs1110/2014sp/labs/lab06/small_dict.txt

New policies. We're implementing some new policies to reduce waiting times in labs, and get your questions answered sooner. Starting with Lab 6:

- As announced on Piazza, you are encouraged to work in and check-off in *pairs*. Both people in the pair must be present in the same lab.
- The lab staff will more vigorously employ their right to only check off students who are registered in their lab. (The staff have the official enrollment lists, and so have the ability to verify official lab registration.)

Getting credit for lab completion. When done, show your code and/or this handout to a staff member, who will ask you (both) a few questions to see that you understood the material and then swipe your ID card to record your success.

If you do not finish during the lab session, you have until the beginning of lab **two weeks from now** to finish it, **i.e., the "lab 8" slot; "lab 7" has been pre-empted by prelim preparation and grading**. But you should always do your best to finish during lab hours. Remember that labs are graded on effort, not correctness.

1. Your job

You'll be writing a program `unscramble.py` that solves anagrams, like the ones that form part of many word games and word puzzles. For instance, given the string "opython", we would like the program to tell us that that string is an anagram of "typhoon"[1] – meaning that it contains the same letters, but possibly in a different order.

When completed, your program should be used by running it from the command line with a word or words as arguments. For instance,

```
python unscramble.py opython CS1110
```

should produce the output

```
"opython" is an anagram of "typhoon"
"cs1110" is not an anagram of anything
```

Lab authors: S. Marschner, L. Lee

[1]And you thought that joke in an early lecture about 1-800-OPYTHON wasn't going to resurface. Yes, this is a course that is meticulously honed to the finest detail (except when it isn't).

1.1. **Key idea: a different representation scheme.** Here is a handy observation: one can tell if two strings contain the same letters by sorting the letters in both strings and comparing the results. Try this on paper if you don't believe it! Taking the letters of "opython" and sorting them results in "hnoopty". Sorting the letters of "typhoon" produces the same result. In other words, if we think of strings as represented by their letter-sorted versions, then anagrams are exactly those strings that have the same representation!

So we can easily solve anagrams by taking a list of all the words in English and sorting the letters in each of them. Then we can find out whether some new string is an anagram of an existing English word by sorting the letters in the new string and looking for an English word that has the same representation as the new string.

1.2. **Outline of programming steps.** As given in more detail in the sections below, we need to do the following.

(1) Get the input(s) from the command line.
(2) Write the function that gives us the letter-sorted representation (*LSR*) of a string.
(3) Set up the lookup lists that tell us the correspondence between a string and its LSR.
(4) Use those lists appropriately.

1.3. **Useful references.** The handouts and programs from the most recent lectures should serve as useful models. They are posted at `http://www.cs.cornell.edu/courses/cs1110/2014sp/lectures`.

Documentation for string *methods* can be found here: `http://docs.python.org/2/library/stdtypes.html#string-methods`. You call a string method like this: `'THanx'.lower()`, which returns `'thanx'`.

Documentation for *functions in the module `string`* can be found here: `http://docs.python.org/2/library/string.html#string-functions`. Like any time a function from a module is called, you use the module name, then a dot, then the function name, like this: `string.lower('THanx')`. So the thing in front of the dot is literally s-t-r-i-n-g, and this call returns `'thanx'`.

Documentation for list methods can be found here: `http://docs.python.org/2/tutorial/datastructures.html`.

Documentation for file methods can be found here: `http://docs.python.org/2/tutorial/inputoutput.html#methods-of-file-objects`.

2. Getting the Input Words from the Command Line

Open the program unscramble.py in Komodo Edit and look at the application-code section (the stuff after the line `if __name__ == '__main__':`). Right now it contains mostly just temporary code to remind you that command-line arguments to Python are stored as a list in the `sys.argv` variable. The temporary code shows you how to print the *entire* list all at once, and how to do something for *each* item of the list, *one at a time*, via a *for-loop*.

Try the following commands at the command line to see how the list sys.argv differs for different command-line arguments:

```
python unscramble.py
python unscramble.py opython
python unscramble.py opython CS1110
```

What changes if you replace the line `for item in sys.argv:` with `for item in sys.argv[1:]:`, and why? Hint: to check your answer, implement the change in Komodo Edit, save, and then run `python unscramble.py opython CS1110`.

What changes if you replaced the same line with `for item in sys.argv[1]:` (no colon in the brackets this time), and why?

What changes if you replaced the same line with `for item in 1:`, and why?

Now, comment out all the temporary code in the application-code section. Using the commented-out lines as a guide, write code in the application-code section such that:

- If at least one word is given to the program unscramble.py on the command line, then for each such word, call function `unscramble` on the *lowercased version* of that word. You may find some of the references listed in Section 1.3 helpful.
- If no words are given to the program unscramble.py on the command line, the program prints out a usage message.

If your code is working, when you run unscramble.py, you should get one message per word, where that message is generated by function unscramble. Then you can remove the lines you commented out.

If you are having trouble with this, ask a staff member now!

3. Creating the Letter-Sorted Representation (LSR) of a String

Implement the function sort_string_letters according to its specification. A straightforward way to do this is by building a list that contains the letters of the argument string, sorting the list, and then using `"".join` appropriately to put it back into a single string. Be careful about which list or string functions return values and which don't!

We've given you a unit test, unscrambletest.py, so you have some test cases with which to test your code. If you get the message "finished test of sort_string_letters" when running the unit test, you've completed this task!

4. Creating a Lowercased List of "All" English Words

We want the global variable `dictionary` to be a list of all English words, converted to lowercase.

To learn how to do this, look at the small set of words contained in file `small_dict.txt`. In Python interactive mode running in the same directory as all your lab 6 code, execute the assignment statement `DICT_FILE = 'small_dict.txt'`. Then, try a few lines that use `open(DICT_FILE)`, the function `read()` — a file method that gets us the file contents as a single line, newlines included — and the `split` string method to create a list wherein each item is one of the words from the file. For small_dict.txt you should get `['Edda', 'Lee', 'Python', 'eel', 'python', 'typhoon', 'dead']`.

Let's suppose you stored that list in a variable `newlist`. We're going to want to map `string.lower` over that list to get a lowercased version of newlist.

First, we need to understand something important about `map`. Try the following in Python interactive mode:

```
map(round, [1.0, 1.4, 1.7])
```

Now try `map(round(), [1.0, 1.4, 1.7])`.
Why does the latter result in an error, while the former doesn't? That is, what's the difference in *meaning* between `round` and `round()`?

With the understanding we just gained, we can now finish this task. In Komodo Edit, add a statement near the top of unscramble.py that imports the `string` module. Then use your new-found knowledge to replace the line `dictionary = []` with one or more statements that store in the global variable `dictionary` a list of lowercased words from DICT_FILE.

5. Create the List of LSRs of Strings in `dictionary`

Replace the line `rep_dict = []` with one or more statements that store in the global variable `rep_dict` a list that is the result of mapping your function sort_string_letters over the list `dictionary`.

If you have succeeded, then you will have two parallel lists, `dictionary` and `rep_dict`, where each item of `rep_dict` is the letter-sorted representation of the corresponding item in `dictionary`.

Test your understanding of this by opening up unscrambletest.py in Komodo Edit and looking at function test_unscramblefn_idea. Then, run unscrambletest.py. If all is well, you should see the message "finished test of main idea in function unscramble". Also, for the file small_dict.txt you should have rep_dict being `['adde', 'eel', 'hnopty', 'eel', 'hnopty', 'hnoopty', 'adde']`.

6. Implement Function `unscramble`

OK, last step! Implementing function `unscramble` is just a matter of using our two parallel lists. Remove the temporary code from unscramble, and then proceed as follows. The code for test_unscramblefn_idea may give you some implementation hints.

Given an input string `anagram`, the function should first create the LSR for `anagram`. It should then check if that LSR is in rep_dict. If it is, find the index `i` of the LSR in rep_dict, and print out a message about how the item at index `i` of *dictionary* is an anagram; see Section 1 for the exact format of the output. If the LSR isn't in rep_dict, print a sorrowful message such as depicted in Section 1.

Demonstrate your unscramble.py program to your lab instructor, and you are done for this lab.

If you now want to use a full dictionary to show off your program to your friends (or enemies), comment out the line `DICT_FILE = 'small_dict.txt'` and uncomment the line `DICT_FILE = 'scowl_utf-8.txt'`.

7. Optional Extra Challenges

(1) Modify your program so that it will find *all* anagrams of the user's strings, not just the first one.
(2) Use Python's *dictionary* mechanisms instead of lists to achieve cleaner code and faster performance.