Homework 2-3

March 13, 2012, 2:25 pm

In this homework, you will finish one of the goals of the section and finally find out the vocabulary size of *Moby Dick*! **Some words of wisdom**: your program will most likely not work the first try. Here is some advice to avoid sitting there clueless about what went wrong:

- Run your program as often as possible, even if it does not do what it's supposed to do. This will help you make sure there are no errors, so that when you do get an error you know the source (from where you are and where you last ran it).
- Whenever you write a function, test it in the interactive environment (Python Shell) as thoroughly as possible (by supplying contrived arguments) to make sure it does the right thing. Then if your program consists of function calls, you will be able to narrow down the places where error(s) could occur.
- Build your code ground up and make sure you understand what all your code does. Not doing that will be the equivalent of building a plane and not knowing where to put a left over screw.

Task 1:

The first task makes sure that the starter code works for you and provides you examples of iterating through lists in a different way.

a. Download the starter code `HW2-3.py`. This starter code contains a function named `vocab` which returns the vocabulary list of your text.

   Then, I put several lines demonstrating how to iterate through a list in two different ways. You should be familiar with the second one (if you are not, please let us know), but you should notice that the first one gives you extra power. Note how I am able to print `'Apple comes after Steve Jobs'` using the first way, but I cannot do that with the second.

   After the examples, I defined more functions to be used later in this homework (do not worry about them now).

b. At the end of the starter code, call our vocabulary-computing function and assign the result to a variable by typing (make sure that your path is correct)

```
v = vocab('MobyDick.txt')
```

Then, inspect the value of variable `v`. You should see a list of strings (words). Now inspect the length of `v` (using the built-in function `len(v)`). It should say `843`. We are only computing the vocab list for the first `10000` characters in *Moby Dick* (Can you see why the program does not compute the vocab list for the whole text?) If the program does not work as expected, please email `cs0931tas@cs.brown.edu` with what you did and the errors you got.

Task 2:

In reality, getting rid of duplicates of a list is slow. We conceived a faster way to do it assuming we can sort a list fast enough. Let us write a function called `fastNoReplicates` that takes a list of words and returns a unique word list. The first lines are provided for you. They sort the word list and initialize our result vocab list with the first word in the sorted list.

a. You want to iterate through the sorted word list in the iterating-by-index way (see how I iterated through the Steve Jobs list using the indices). One subtle difference is that you want to start with the *second* word, since (1) you already put the first one in the result list, and (2) later you will compare a word to the previous one, but the first word has no previous. Start a for-loop as such, by supplying `1` instead of `0` as the first argument to `range`.

b. Now we write the body of the loop (statements that will be executed multiple times). Suppose you called your looping variable (the variable right after `for`) `index`. Then each time the loop is run, `index` takes a different value (`0, 1, 2, 3, 4, ...`). What is the expression that evaluates to the element of the list at position `index` (Put it in a variable called `current`)? At the previous position (Put it in a variable called `previous`)?

c. (Still writing the body of the for-loop; be careful with indentation throughout the program!) If `current` is the same as `previous`, we want to say `pass` (`pass` is a special word in Python that means do nothing in this line. Do not try to put quotes or brackets around it);

otherwise, we want to append `current` to our result(it's the first time we see it). **Be careful: `current` is a string and `result` is a list.**

d. After the loop is completed (again, be careful with the indentation), the result should hold the list without duplicates. Return it.

Task 3:

Now, let's deploy our new method of removing replicates.

a. Modify the `vocab` function so that it uses `fastNoReplicates` instead of `noReplicates`.

b. Run the program as in Task 1 and inspect the vocabulary list and its size. You should get the same answer as before (both methods are correct, just that the new one is faster).

c. Now, in the function `readFile`, instead of returning `myString[:10000]`, return `myString`. We are no longer afraid of processing large lists!

d. Run the program again. Are you impressed with how fast it computes the vocabulary list? This time, do not try to inspect the whole list because it takes too long to print. You can first look at how many words are in there. Then you can look at slices of this list, for example, `v[1000:2000]`.

Congratulations! You have just completed a **software upgrade**.

Task 4:

As you inspected different portions of the vocabulary list, you may see that many improvements can still be made! There are numbers, punctuations, mixed cases (`whale` and `Whale` should really be the same word). Now let us fix that.

Essentially, we want to do some clean-ups to the big string (of text) before we split it into words. Here are some possible options: turn all letters into lowercases, and replace all numbers and punctuations with whitespaces (so that `eat,pray,love` can be split as if it is `eat pray love`).

a. I've written a function called `cleanup` for you. *It takes a string and returns a cleaned-up string* (always ask yourself what kind of arguments a function takes and what kind of value it returns). As you

can see, I have solved a problem by creating more problems (I have to define more functions for this to work!). First of all, note that I am using a built-in function called `lower` to turn all letters in my string to lower cases (You can convince yourself it works by running a simple example in the Shell environment). Then I am using the functions `removeNumbers` and `removePunctuations`. What do you think their function is? What type of values do they take? What type of values do they return?

b. I also wrote `removeNumbers` for you as an example. Read it and make sure that you understand how it works. There are a couple of things to notice:

   - You can iterate through characters in a string the same way you iterate through items in a list.
   - However, you cannot modify the string. So I have to create an empty string first and as I iterate, append new characters to it.
   - When I append a new character, I use the `+=` short-hand because my result string will eventually grow really big.
   - A string that contains a whitespace is not the same as an empty string.

c. Now write the function `removePunctuations` that *takes a string and returns another string, replacing punctuations with whitespaces.*

Task 5:

a. Now, we need to insert this cleanup step into our assembly line. Our assembly line is in the function `vocab` (read→split→remove duplicates). Use the function `cleanup` to cleanup your big string before you split it.

b. Run your program and inspect your vocab list (partially). You may discover that you may have missed a lot of possible punctuations. You can go back and improve your function if you are a perfectionist.

c. Hoooooraaaay! Now the vocab list looks pretty good. Yes it is not perfect: what about `create, creates, creating, created`? We will deal with them next time.

Task 6:

Now you will create a function that computes the average length of a word in a given string that **is greater than** a given length. For example, this function should be able to find the average length of a word in Moby Dick among the words that are greater than n characters. There is some stencil code already written for you in a function called `avgWord`. Some (hopefully) helpful guidelines before you begin:

a. Think about what arguments you'll need. **Hint:** What aspects of this function can be customized according to what you are trying to accomplish? The function can work with just two arguments.

b. You will need four variables, three of which have already been created for you.

   - `myWordList` has already been created for you and holds a list of words from your text file. This variable is created from calling `vocab`, which we had just previously used. This means that `myWordList` has been cleaned up already (so repeated words, numbers, and punctuation marks have already been removed).
   - `sumOfLengths` has also already been created. This variable should start out as zero and will hold the current sum of the word lengths as your loop iterates. As you iterate, any time you encounter a word in `myWordList` that is greater than or equal to the minimum length required (n), add its length to `sumOfLengths` and update the value of `sumOfLengths`.
   - `numOfWords` has also already been created. This variable should first hold the number of words in `myWordList`. As you iterate through `myWordList`, decrease `numOfWords` by **1** if a word does not meet the minimum length requirement.
   - Your last variable should be named `avg`. This variable should start out as zero and holds the average length of a word (which in this case is given by `sumOfLengths/numOfWords`). Think about when this average will **NOT** be able to compute. Specifically, the denominator cannot take a certain value. **Make sure to handle this in your code**.

c. After creating the variables `myWordList`, `sumOfLengths`, and `numOfWords`, you will need to iterate through `myWordList`. Which method of iterating should you use of the two described methods earlier? Do you need

to handle specific indices differently, or is each word in `myWordList` handled the same way?

Tips for testing. Remember that the average word length returned by the function should always be greater than or equal to the minimum length required **unless** all of the words in the text file are less than the minimum required length.

Handin

Email your program to `cs0931tas@cs.brown.edu` and title the file `'YOURNAME'HW2-3.py` — for example, `BobJonesHW2-3.py`.