About Lab 07

In Lab 7 you will write 2 programs that play games with words:

- Program distill.py asks the user for a file name and a number n. It then prints the file, leaving out the n most common words.
- Program anagrams.py asks the user for the name of a dictionary file. It then goes into a loop asking for a string and printing anagrams of that string.

The distill program uses dictionaries and the anagrams program uses sets.

Most of the distill program is straightforward. You want a dictionary to hold words and their counts -- the keys will be words, the value associated with a word is the number of times you have seen it. If your dictionary is called D (that's a crappy name for a dictionary; use something more meaningful) you will have code like this:

```
    if word in D.keys():
        D[word] = D[word] + 1
    else:
        D[word] = 1
```

The keys should be real words and you'll get words with punctuation attached (such as "Bob!") so we suggest that you write a function cleanstring(s) that starts with s, turns it into lower-case, removes all punctuation, and returns the result. The punctuation comes at the end of the word, and occasionally at the beginning.

Just as in the Concordance program, our friend the strip( ) method for strings comes in handy here. If s is a string,

```
        s.strip( punct )
```

returns a copy of s with all of the letters of punct removed from both the start and the end of s.

So make your self a punctuation string and put in it every punctuation mark you can think of.

Once you have built the dictionary you need to find the n most common words.   The easiest way I can find for that is to do the first n steps of SelectionSort. Put the whole dictionary into a list of [word, count] pairs. Make a pass through it, looking for the index of the word with the largest count. Interchange that entry with the entry at index 0. Make a pass starting at index 1, looking for the largest remaining element, and interchange that with the element at index 1, and so forth.

Once you have the n most common words in a list, make another pass through the file. Divide it into words, check to see if the cleanstring( ) version of each word is one of the most common words, and if not print it.

There is only one tricky place. It is possible for cleanstring(s) to return an empty string. For example, one of the files has a "word" that is "---". When you strip off the punctuation there is nothing left. Don't put such words into your dictionary.

The anagrams program asks you to enter the name of a dictionary file, which it loads and stores as a set of words. It then goes into a loop where it asks the user for a string, removes the spaces from the string, and then prints all of the anagrams it can make from the string using words from the dictionary.

For example, if you enter "oberlin student", among the many anagrams it finds are

let none disturb
let in; runs to bed
trust line on bed

For "oberlin conservatory" it finds

boy never controls air
so convert one library
only recover in bars

And for "hermione granger" it finds

ignore green harm

The program is fun to play with and a great time-waster, but it does run on. For every set of words making up an anagram, it will print every possible ordering of the words. I counted almost 35,000 lines of output in response to "oberlin student".

There are really just two major functions that you need to write for this. The first of these is **contains(s, word)**, which returns a pair of values. If string *s* does not contain string *word*, this returns (False, ""). If *s* does contain *word*, this returns (True, t) where string *t* is the same as string *s* with the letters of word removed.

For example, **contains("bombast","bob")** returns (True, "mast") while **contains("bouncy", "bob")** return (False,"") since "bouncy" has only one 'b'.

contains(s, word) should be easy to write. We set a variable t = s, then loop through the letters of word checking to see if they are in t and if so removing them. If we are able to remove all of the letters of word we return True and whatever is left of t. If one of the letters of word is not in t we return (False, "")

Here is an easy way to remove the first instance of letter 'a' from t:

```
        t = t.replace( "a", 1)
```

Technically that says to replace the first instance of "a" with the empty string.

The other function you need to write is **grams(s, words, sofar)** which is a recursive function that prints anagrams. *s* is a string we want to find anagrams for. *words* is our dictionary set. *sofar* is a list of words we have taken so far out of the string. The function looks for words that *s* contains and recurses on what is left, with the word added to the *sofar* list. If it recurses down to where s is the empty string, it prints the *sofar* list.

For example, if we call

```
        grams( "hermionegranger", words, [ ])
```

we eventually find that

```
        contains("hermionegranger", "ignore")
```

returns (True, "hmeranger") so we recurse with

```
        grams("hmeranger", words, ["ignore"])
```

We eventually find that

```
        contains("hmeranger", "green")
```

returns (True, "hmar") so we recurse with

```
        grams("hmar", words, ["ignore", "green"])
```

grams("hmar", words, ["ignore", "green"]) eventually finds that

contains("hmar", "harm") returns (True, "") so we recurse on

grams("", words, ["ignore", "green", "harm"])

and since the string argument is now empty we print the anagram:

ignore green harm

The trickiest thing about grams(s, words, sofar) is that each time you find a match for s you need to rebuild the sofar list. This is the reverse side of mutability -- if you only append to sofar then you only have one list and there is no way to remove anything from it. So if s contains word t with s1 left over then you will recurse on

```
        grams(s1, words, L)
```

where we get the new list L with

```
        L = [ ]
        L.extend(sofar)
        L.append(t)
```

Alternatively, you could recurse with

```
grams(s1, words, sofar+[t])
```