Black Problem 2: Scrabble Scoring [35 points; individual only]

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/ScrabbleScoringBlack2010 on 3/22/2017

This is the only "individual only" problem this week.

At the end of Problem 1, we did a bit of rudimentary Scrabble scoring. Your ultimate task here is to write a function that takes as an argument a "rack"—a **list** of letters—and returns the highest-scoring word that can be made with those letters (remember that each letter in the rack can only be used once). This is the key ingredient in a computerized Scrabble game!

Please place your solution to this problem in a file called `hw1pr2.py`. That is the file that you will be submitting.

In this problem, you may use recursion as well as the built-in higher-order functions `map`, `filter`, and `reduce`. **In fact**, if you write a function that needs to process a long list (e.g. a dictionary) with more than a few hundred items in it, you are much better off letting `map`, `filter`, or `reduce` cruise through those lists since they have been optimized to work very fast.

There will be a few places here where using anonymous functions (`lambda`) is probably the cleanest and nicest way to do business. Use it when it's the cleanest way to proceed.

One of the objectives of this problem is to have you think about designing a more complicated program that involves several functions. You have complete autonomy in deciding what functions to write in order to reach the ultimate goal (see more on the two required functions below). Try to think carefully about which functions you will need, and try to implement those functions so that they are as simple and clean as possible. Part of your score on this problem will be based on the elegance of your overall design and individual functions.

Our sample solution has fewer than 9 functions, two of which we copied from our solution to Problem 1. The remaining 7 (or so) functions range from one to four lines of code per function. While you are not required to have the same number of functions and you may have a few slightly longer functions, this is intended to indicate that there is not much code that needs to be written here!

All of your functions should be in a file called `hw1pr2.py`. Be sure to have a comment at the top of the file with your name(s), the filename, and the date.

Be sure to include a docstring for every function that you write.

You may wish to include—by copying—some of the functions that you wrote in Problem 1.

A bit later in this problem, we'll give you a fairly large dictionary of English words to use. For now, we recommend that you use the following tiny dictionary during the course of testing and development. Include these lines near the top of your file. That way, they will be global variables that can be used when we—and you—test your functions.

Place these two variables at the top of your file:

```
scrabbleScores = [ ["a", 1], ["b", 3], ["c", 3], ["d", 2], ["e", 1],
                   ["f", 4], ["g", 2], ["h", 4], ["i", 1], ["j", 8],
                   ["k", 5], ["l", 1], ["m", 3], ["n", 1], ["o", 1],
                   ["p", 3], ["q", 10], ["r", 1], ["s", 1], ["t", 1],
                   ["u", 1], ["v", 4], ["w", 4], ["x", 8], ["y", 4],
                   ["z", 10] ]


nanoDictionary = ["a", "am", "at", "apple", "bat", "bar", "babble",
                  "can", "foo", "spam", "spammy", "zzyzva"]
```

Do not change the values of these global variables!

The details...

Ultimately, there are two functions that we will be testing.

- `scoreList(rack, dictionary)` has two arguments: a `rack`, which is a list of lower-case letters, and `dictionary`, which is a list of legal words.

The `scoreList` function returns a list of all of the words in the `Dictionary` argument that can be made from those letters, together with the score for each one. Specifically, this function returns a list of lists, each of which contains a string that can be made from the `Rack` and its Scrabble score. Here are some examples using `nanoDictionary`above:

```
In [1]: scoreList(["a", "s", "m", "t", "p"], nanoDictionary)
Out[1]: [['a', 1], ['am', 4], ['at', 2], ['spam', 8]]

In [2]: scoreList(["a", "s", "m", "o", "f", "o"],
nanoDictionary)
Out[2]: [['a', 1], ['am', 4], ['foo', 6]]
```

The order in which the words are presented is not important.

- `bestWord(rack, dictionary)` accepts a `Rack` and `Dictionary` as above and returns a list with two elements: the highest-scoring possible word from that `Rack`, followed by its score. If there are ties, they can be broken arbitrarily. Here is an example, again using `nanoDictionary` above:

```
In [1]: bestWord(["a", "s", "m", "t", "p"], nanoDictionary)
Out[1]: ['spam', 8]
```

Aside from these two functions, all of the other helper functions are up to you! Some of those helper functions may be ones that you wrote in Problem 1, or slight variants of those functions.

You might find it useful to use a strategy in which you write a function that determines whether or not a given string can be made from the given `Rack` list. Then, another function can use that function to determine the list of *all* strings in the dictionary that can be made from the given `Rack`. Finally, another function might score those words.

Remember to use `map`, `reduce`, or `filter` where appropriate—they are powerful and they are optimized to be very fast.

Test each function carefully with small test arguments before you proceed to write the next function. This will save you a lot of time and aggravation!

A reminder about Python's `in` operator:

Imagine that you are writing a function that attempts to determine if a given string `S` can be made from the letters in the `Rack` list. You might be tempted to scramble ("permute," to use a technical term) the `Rack` in every possible way as part of this process, but that is more work than necessary. Instead, you can test if the first symbol in your string, `S[0]`, appears in the rack with the statement:

```
if S[0] in Rack:
  # if True you will end up here
else:
  # if False you will end up here
```

This `in` operator is very handy. From here, recursion will let you do the rest of the work without much effort on your part!

Although we don't expect that you will end up doing a huge amount of recursion, you should know that Python can get snippy when there are a lot of recursive calls.

By default, Python generally complains when there are 1000 or more recursive calls of the same function. To convince it to be friendlier, you can use the following at the top of your file:

```
import sys

sys.setrecursionlimit(10000)  # Allows up to 10000 recursive
calls; the maximum allowable varies from system to system
```

Trying it with a bigger dictionary

Finally, if you want to test out a big dictionary (it's more fun than the little one above), you can download the file dict.py. It contains a large list of words (a bit over 4000). The list is simply called `Dictionary`.

After you have downloaded the file and placed it in the same directory as your `hw1pr2.py` file, you can import that file at your shell in order to use `Dictionary` in your testing.

Here are some examples showing how to use this larger `Dictionary`

```
In [1]: from dict import *   # this imports the large Dictionary
variable
```

```
In [2]: scoreList(['w', 'y', 'l', 'e', 'l', 'o'], Dictionary)
Out[2]: [['leo', 3], ['low', 6], ['lowly', 11], ['ow', 5],
['owe', 6], ['owl', 6], ['we', 5], ['well', 7], ['woe', 6],
['yell', 7], ['yo', 5]]
```

Notice that "yellow" is not in this dictionary. We "cropped" off words of length 6 or more to keep the dictionary from getting too large. Finally, here are examples of `bestWord` in action (remember that tie scores are OK, too):

```
In [3]: bestWord(['w', 'y', 'l', 'e', 'l', 'o'], Dictionary)
Out[3]: ['lowly', 11]

In [4]: bestWord(["s", "p", "a", "m", "y"], Dictionary)
Out[4]: ['may', 8]

In [5]: bestWord(["s", "p", "a", "m", "y", "z"], Dictionary)
Out[5]: ['zap', 14]
```

And an even bigger dictionary!

Want to try this with an even bigger dictionary? Two students from CS 5 Black a couple of years ago constructed one with over 100,000 words. You should be able to `import` it into your program without difficulty:

Start by downloading bigD.txt. Place this file in the same directory as your `hw1pr2.py` and rename it as `bigD.py`.

Then, you can use the same technique as above:

```
In [1]: from bigD import *    # this imports the very large
Dictionary variable

In [2]: scoreList(['a', 'b', 'v', 'x', 'y', 'y', 'z', 'z', 'z'],
Dictionary)
Out[2]: [['ab', 4], ['aby', 8], ['ax', 9], ['ay', 5], ['ba', 4],
['bay', 8], ['by', 7],
 ['ya', 5], ['yay', 9], ['za', 11], ['zax', 19], ['zyzzyva',
43], ['zzz', 30]]
```

Submitting

Submit all of your functions in a single file called `hw1pr2.py`.

Do not include any of the special dictionary-`import` statements in your code—that will make the graders angry, as they may or may not have those files…

Good luck—have fun!