Black Lab 1: Recursion Muscles [30 points; individual or pair]

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/FunctionFrenzyBlack2010 3/22/2017

This problem asks you to write the following Python functions *using recursion* (not loops!) and to test these functions carefully.

You may use recursion, conditional statements (`if`, `else`, `elif`), and list or string indexing and slicing. Note that some of these problems can be written without using recursion, e.g. using `map`, `filter`, `reduce`, or other "mass-processing" structures. However, the objective here is to build your recursion muscles, so please stick to recursion. **DO NOT** use any loop structures (`while` or `for`). Loops will make your recursion muscles weak and flabby.

Try to keep your functions as "lean and clean" as possible. That is, keep your functions short and elegant.

Do not use built-in functions (e.g. `len`, `sum`, etc.). However, your functions may call other functions that you write yourself. Calling another function for help will mostly be unnecessary, but it may be handy in a few places.

Be sure to include a docstring under the signature line of each function. The docstring should indicate what the function computes (returns) and what its arguments are or what they mean.

Please put all of your functions for this problem in a **single** file called `hw1pr1.py`. Thus, all of the parts of this problem will be submitted in a single file. **Please be sure to name your functions exactly as specified so that the graders can find them.**

- `dot(L, K)` should return the dot product of the lists `L` and `K`. Recall that the dot product of two vectors or lists is the sum of the products of the elements in the same position in the two vectors. You may assume that the two lists are of equal length. If they are of different lengths, it's up to you what result is returned. If the two lists are both empty, `dot` should return `0.0`. Assume that the argument lists contain only numeric values.
- `In [1]: dot([5,3], [6,4])  <-- Note that 5*6 + 3*4 = 42`

- `Out[1]: 42`

  Besides the example above, try `dot([1], [])` and `dot([], [42])` and a few others of your own devising.

- `explode(S)` should take a string `S` and return a list of the characters (each of which is a string of length 1) in that string. For example:
- `In [1]: explode("spam")`
- `Out[1]: ['s', 'p', 'a', 'm']`
-
- `In [2]: explode("")`
- `Out[2]: []`

  Note that Python is happy to use either single quotes or double quotes to delimit strings—they are interchangeable. But if you use a single quote at the start of a string you must use one at its end (and similarly for double quotes). For example:

  ```
  In [1]: "spam" == 'spam'
  Out[1]: True
  ```

- `ind(e, L)` accepts an element `e` and a sequence `L`, where by "sequence" we mean either a list or a string (fortunately indexing and slicing work the same for both lists and strings, so your `ind` function should be able to handle both types of arguments!). Then `ind` should return the index at which `e` is **first** found in `L`. Counting begins at 0, as is usual with lists.

  If `e` is NOT an element of `L`, then `ind(e, L)` should return an integer that is ***exactly*** the length of `L`.
  Remember, don't use the `len` function explicitly though! Your recursive implementation can find the length by itself.
- `In [1]: ind(42, [ 55, 77, 42, 12, 42, 100 ])`
- `Out[1]: 2`
-
- `In [2]: ind(42, list(range(0,100)))`
- `Out[2]: 42`
-
- `In [3]: ind('hi', [ 'hello', 42, True ])`
- `Out[3]: 3`
-
- `In [4]: ind('hi', [ 'well', 'hi', 'there' ])`
- `Out[4]: 1`
-
- `In [5]: ind('i', 'team')`
- `Out[5]: 4`

```
In [6]: ind(' ', 'outer exploration')
Out[6]: 5
```

- `removeAll(e, L)` accepts an element `e` and a **list** `L`.
  Then `removeAll` should return another list that is the same as `L` except that all elements identical to `e` have been removed. Notice that `e` has to be a top-level element to be removed, as the examples illustrate:

```
In [1]: removeAll(42, [ 55, 77, 42, 11, 42, 88 ])
Out[1]: [ 55, 77, 11, 88 ]

# Below, 42 is NOT top-level!
In [2]: removeAll(42, [ 55, [77, 42], [11, 42], 88 ])
Out[2]: [ 55, [77, 42], [11, 42], 88 ]

# Below, [77,42] IS top-level!
In [3]: removeAll([77, 42], [ 55, [77, 42], [11, 42], 88 ])
Out[3]: [ 55, [11, 42], 88 ]
```

  Aside: It's possible to write `removeAll` so that it works even if the second argument is a string instead of a list, but you do not need to do so here.

- Recall that Python has a built-in function called `filter` that takes two arguments: The first is a function `f` that accepts a single argument and returns either `True` or `False`. Such a function is called a *predicate*. The second argument to `filter` is a list `L`. The `filter` function returns a new list that contains all of the elements of `L` for which the predicate returns `True` (in the same order as in the original list `L`). For example, consider the example below:

```
In [1]: def even(x):
...     if x % 2 == 0:
...         return True
...     else:
...         return False
...

In [2]: filter(even, [0, 1, 2, 3, 4, 5, 6])
Out[2]: [0, 2, 4, 6]
```

  In this example, the predicate `even` returns `True` if and only if its argument is an even integer. When we invoke `filter` with predicate `even` and the list `[0, 1, 2, 3, 4, 5, 6]` we get back a list of the even numbers in that list. Of course, the beauty of `filter` is that you can use it with all kinds of predicates and all kinds of lists. It's a very general and powerful function! Your job is to write your own version of `filter`, called `myFilter`, that uses recursion. Remember, your

implementation may use recursion, indexing and slicing, and concatenation—but no built-in Python functions.

- `deepReverse(L)` accepts a list of elements, where some of those elements may be lists themselves. `deepReverse` returns the reversal of the list where, additionally, any element that is a list is also deepReversed. Here are some examples:

```
In [1]: deepReverse([1, 2, 3])
Out[1]: [3, 2, 1]

In [2]: deepReverse([1, [2, 3], 4])
Out[2]: [4, [3, 2], 1]

In [3]: deepReverse([1, [2, [3, 4], [5, [6, 7], 8]]])
Out[3]: [[[8, [7, 6], 5], [4, 3], 2], 1]
```

  For this problem, you will need the ability to test whether or not an element in the list is a list itself. To this end, you can use the following line of code, which tests whether or not `x` is a list:

```
if type(x) == type([1, 2, 3]):
  # if True you will end up here
else:
  # if False you will end up here
```

  The list `[1, 2, 3]` could be replaced by any list—it's just important that it be *some* list (even `[]`). Python is answering the question, "is it the case that `x` is a thing of the same type as `[1, 2, 3]` ?" Since `[1, 2, 3]` is a list, this is really just a sneaky way of asking "is `x` a list?

- `letterScore(letter, scorelist)` accepts a single letter string called `letter` and a list, where each element in that list is itself a list of the form `[character, value]`. In those inner lists, `character` is a single letter and `value` is a number associated with that letter (e.g., its Scrabble score). The `letterScore` function then returns a single number, namely the value associated with the given `letter`. For example, you can cut and paste the following Scrabble score list into your `hw1pr1.py` 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] ]
```

If you include this in your file (outside of any function you define—for example right after the header comments in your file)—
then `scrabbleScores` is a "global variable"; it can be referred to by any function defined in that file and, more importantly for this example, it can be used once we load in that file.

```
In [1]: letterScore("c", scrabbleScores)
Out[1]: 3

In [2]: letterScore("a", scrabbleScores)
Out[2]: 1
```

If the `letter` is not in the `scorelist`, `letterScore` should not crash. Instead, it should return something sensible (such as 0). This is an example of *input validation*—making sure your program behaves well even if it is misused. (Bad input validation is the number-one cause of computer security problems!)

- `wordScore(S, scorelist)` should accept a string `S` and a `scorelist` in the format described above, and should return the Scrabble score of that string. Again, `wordScore` should behave well if `S` contains letters not found in `scoreList`. However, you are allowed to crash badly if `scoreList` is in the wrong format (such as not being a list at all); that's because we haven't yet learned the way to protect against that kind of crash.

  Here are some examples:

```
In [1]: wordScore('spam', scrabbleScores)
Out[1]: 8

In [2]: wordScore("wow", [['o', 10], ['w', 42]])
Out[2]: 94
```

Submit

Make sure your name and date are at the top of your file. Then, please submit your functions on the submission system in a file called `hw1pr1.py`. Please remember to name your functions exactly as they appeared in this problem -- thanks!