Recursion Muscles

This problem asks you to write the following Python functions **using recursion** and to test these functions carefully.

You may use recursion, conditional statements (`if`, `else`, `elif`), list or string indexing and slicing. Some of these problems can be written without using recursion, e.g. using `map`, `filter`, `reduce`, or other looping structures. However, the objective here is to build your recursion muscles, so please stick to recursion.

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 (outputs) and what its inputs are or what they mean.

Please be sure to name your functions exactly as specified for grading purposes.

- `dot(L, K)` should output 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 these two lists are both empty, `dot` should output `0.0`. Assume that the input lists contain only numeric values.

  ```
  >>> dot([5,3], [6,4])   <-- Note that 5*6 + 3*4 = 42

  42
  ```

- `explode(S)` should take a string `S` as input and should return a list of the characters (each of which is a string of length 1) in that string. For example:

  ```
  >>> explode("spam")

  ['s', 'p', 'a', 'm']

  >>> explode("")

  []
  ```

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 the string you must use one at the end of the string (and similarly for double quotes). For example:

```
>>> "spam" == 'spam'
True
```

- `ind(e, L)` takes in an element `e` and a sequence `L` where by "sequence" we mean either a list or a string; fortunately indexing and slicing works the same for both lists and strings, so your `ind` function should be able to handle both types of input!. 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 at least the length of `L`. Remember, don't use the `len` function explicitly though! Your recursive implementation can do this by itself.

```
>>> ind(42, [ 55, 77, 42, 12, 42, 100 ])
2
>>> ind(42, range(0,100))
42
>>> ind('hi', [ 'hello', 42, True ])
3
>>> ind('hi', [ 'well', 'hi', 'there' ])
1
>>> ind('i', 'team')
4
>>> ind(' ', 'outer exploration')
5
```

- `removeAll(e, L)` takes in an element `e` and a **list** `L`. Then, `removeAll` should return another list that is identical to `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:

```
>>> removeAll(42, [ 55, 77, 42, 11, 42, 88 ])

[ 55, 77, 11, 88 ]

>>> removeAll(42, [ 55, [77, 42], [11, 42], 88 ])  <-- NOTICE HERE THAT 42 IS NOT TOP-LEVEL - IT'S "HIDING" DEEP INSIDE OTHER LISTS AND IS NOT ITSELF AN ELEMENT OF THE LIST L

[ 55, [77, 42], [11, 42], 88 ]

>>> removeAll([77, 42], [ 55, [77, 42], [11, 42], 88 ])  <-- NOTICE HERE THAT THE LIST [77, 42] IS TOP-LEVEL; IT'S AN ELEMENT OF THE LIST L.

[ 55, [11, 42], 88 ]
```

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

- Recall that Python has a built-in function called `filter` that takes two inputs: The first input is a function `f` that takes as input a single input and returns either `True` or `False`. Such a function is called a *predicate*. The second input is a list `L`. The `filter` functions 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:

```
>>> def even(X):

...     if X % 2 == 0 : return True

...     else: return False

...

>>> filter(even, [0, 1, 2, 3, 4, 5, 6])

[0, 2, 4, 6]
```

In this example, the predicate `even` returns `True` if and only if its input 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 different kinds of predicates and all different kinds of lists. It's a very general and powerful function! Your job is to write your own version of `filter` called `myFilter`. Remember, your implementation may use recursion, indexing and slicing, and concatenation but no built-in python functions.

- `deepReverse(L)` takes as input 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:

```
>>> deepReverse([1, 2, 3])

[3, 2, 1]

>>> deepReverse([1, [2, 3], 4])

[4, [3, 2], 1]

>>> deepReverse([1, [2, [3, 4], [5, [6, 7], 8]]])

[[[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 to test whether or not `x` is a list:

```
if isinstance(x, list):
  # if True you will end up here
else:
  # if False you will end up here
```