Lab 7 Problems

Problem 1 The following recursive function `reverse_string(s)` takes a string `s` as an argument and returns a new string that is a reverse of `s`.

```
def reverse_string(s):
    if len(s) == 0:
        return
    return s[-1] + reverse_string(s[:-1])
```

Step 1: Type in the code for the function and run it.

a) What error do you get?

b) What is the cause of the error?

Step 2: Fix the code and run it again. Write out the arguments and the return values for each call for this example call:

```
reverse_string("beak")
```

Problem 2 Write a recursive function `count_occurrences(alist, value)` that counts the number of times `value` occurs in a list. For example,

`count_occurrences([2, 8, 2, 6, 2, 9], 2)` returns `3`.

Write your code below or in your IDE.

Problem 3 In a recent ICA, you wrote a recursive function `sum_diag(grid)` that returns the sum of the diagonal from upper left to bottom right in a grid, i.e., it sums `grid[0][0], grid[1][1],` and so on. Slicing the 2-d list `grid` on each recursive call handles going to the next row, but to handle the change in the column, we used a *helper function* to introduce an additional parameter `col` for the column. Here is the solution:

```
def sum_diag(grid):                 # the original function

    return sum_diag_helper(grid, 0)

def sum_diag_helper(grid, col):     # the helper function

    if grid == []:
        return 0
    else:
        return grid[0][col] + sum_diag_helper(grid[1:], col + 1)
```

Step 1: Circle the first call to the helper function above (where the argument 0 is provided).

Step 2: Use a helper function to write the code for the following problem: write a recursive function `times_pos(alist)` that takes the list `alist` as an argument and returns a new list consisting of the elements of `alist` multiplied by the position number of the element. For example, the call

`times_pos([2,4,6,8,10])` returns `[0, 4, 12, 24, 40]`.

Write your code below or in your IDE.

Problem 4 In this problem, you will be working with the `LinkedList` and `Node` classes, however, the `Node` class has been modified to include an attribute than refers to another `LinkedList` object:

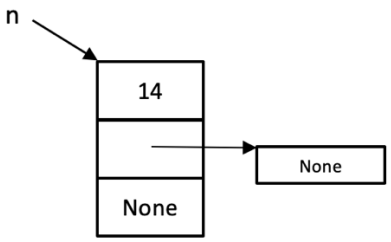
```
class Node:
    def __init__(self, value):
        self._value = value
        self._inner_list = LinkedList()
        self._next = None

    def get_inner_list(self):
        return self._inner_list
```

For example, for the code below,

```
n = Node(14)
```

the diagram for the variable `n` and the `Node` object would look like this:



Step 1: Given the code definition of the `Node` class above, draw the diagram of `my_ll` and `n` after the following two lines have been executed:

```
my_ll = LinkedList()
n = Node(4)
```

Step 2: Draw the diagram again after the following two lines have been executed:

```
n.get_inner_list().add(Node(2))

my_ll.add(n)
```

Step 3: Get the starter code `lab7_starter.py` from the class website (use the Labs link).

Step 4: Read the code in main and the do the steps in the comments for Steps 4 (a) through (d) in the `main()` function.

Step 5: Draw the current diagram for `my_ll`. (Note: the outer list `my_ll` should have three nodes; each of those nodes has an inner list containing one element.)