CompSci 107 – Computer Science Fundamentals
S1 – 2015
Lab/Assignment 7 – Recursion

Assignment Due Date: 7pm May 15, 2015

Worth: This assignment is This assignment is worth 2.5% of your final grade. The lab will be marked out of 10. The programming exercises will be marked out of 9, with the remaining 1 mark from the code review process.

Topics covered:

- Using recursion to solve problems

NOTE: Download the "A7.zip" file from the assignments website:

https://www.cs.auckland.ac.nz/courses/compsci107s1c/labs/

NOTE(VERY IMPORTANT): **Your solutions for this assignment should ALL use recursion (no loops).**

PLEASE NOTE

The following exercises **must be developed on your computer (or on your USB, or on your university drive). Note that you may be asked to produce the code you developed.** Once you are happy that your answer for each question is correct, you will then need to insert the part of the answer required into CodeRunner.

Exercise 1 (1 mark).

Complete the function, `is_descending()`, which takes 2 parameters: a list and a position in the list.

The function uses recursion and returns `True` if all the elements **from the parameter position to the end of the list** are in descending order, `False` otherwise, e.g., the following code:

```
a_list = [7, 6, 5, 3, 2, -4]
going_down = is_descending(a_list, 0)
print(a_list, "is descending:",  going_down)
```

prints:

```
[7, 6, 5, 3, 2, -4] is descending: True
```

To test your completed implementation, run the `IsDescending` application. The output produced should be:

```
[8, 6, 5] is descending: True
[8, 6, 7] is descending: False
[7, 6, 5, 3, 2, -4] is descending: True
[7] is descending: True
[8, 7, 7] is descending: False
[] is descending: True
[1, 3, 5, 7, 2, 1] is descending: False
```

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 7 exercise 1.

Exercise 2 (1 mark).

Complete the recursive function, `is_a_pal()`, which takes one string parameter and returns `True` if the parameter string is a palindrome, `False` otherwise. Note that the parameter string may contain characters which are not alphabetic – these should be ignored by the function (use the `isalpha()` string method to test if a character is an alphabetic character). You can assume that all the alphabetic characters in the string are lower case characters.

The following code:

```
phrase = "Lisa Bonet ate no basil"
print(is_a_pal(phrase.lower()))
print(is_a_pal("Palindrome!".lower()))
```

prints:

```
True
False
```

To test your completed implementation, run the `IsAPalindrome` application. The output produced is a list of phrases which are palindromes followed by a list of phrases which are not palidromes (see the correct output within triple quotes at the bottom of the application).

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 7 exercise 2.

Exercise 3 (2 marks).

Complete the function, `join_multiply()`, which takes 5 parameters: a first list of numbers, a second list of numbers, a third list to be filled with elements, position in the first list (initially 0), position in the second list (initially 0).

The function uses recursion to fill the the third parameter list with elements which are the product of each element from the first list with each element in the second list (in this order) , e.g., the following code:

```
a_list = [4, 5]
list2 = [3, 6, 8]
result = []
join_multiply(a_list, list2, result, 0, 0)
print(a_list, "join multiply",  list2, "=", result)
```

prints:

```
[4, 5] join multiply [3, 6, 8] = [12, 24, 32, 15, 30, 40]
```

Note that if either of the first two parameter lists is empy, the function returns, i.e., it does not add any elements to the third list.

To test your completed implementation, run the `JoinMultiply` application. The output produced should be:

```
[3] join multiply [4, 5, 6] = [12, 15, 18]
[4, 5] join multiply [3, 6, 8] = [12, 24, 32, 15, 30, 40]
[1, 2, 3] join multiply [4, 5, 6] = [4, 5, 6, 8, 10, 12, 12, 15, 18]
[] join multiply [4, 5, 6] = []
[1, 2, 3] join multiply [] = []
[6, 2, 3] join multiply [4, 3, 2, 5] = [24, 18, 12,
                                        30, 8, 6, 4, 10, 12, 9, 6, 15]
```

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 7 exercise 3.

Exercise 4 (2 marks).

Complete the function, `level_out()`, which takes 2 parameters: a list and a position in the list (initially 0). Note that the parameter list can contain elements which are Python lists.

The function uses recursion to take all the levels out of the parameter list , i.e., the function returns a new list with all the elements on the one level, e.g., the following code:

```
a_list = [4, [6, [8]], [2], 3]
result = level_out(a_list, 0)
print(a_list, "levelled:",  result)
```

prints:

```
[4, [6, [8]], [2], 3] levelled: [4, 6, 8, 2, 3]
```

Note: the `instanceof` function can be used to check if an element is a `list` object, e.g.,

```
if isinstance([2, 3], list):
    print("Yes it is")
```

prints:

```
Yes it is
```

To test your completed implementation, run the `OneLevel` application. The output produced should be:

```
[3, 6, 5] levelled: [3, 6, 5]
[3, [2, 1]] levelled: [3, 2, 1]
[3, [2, [4, 5], 9, 6]] levelled: [3, 2, 4, 5, 9, 6]
[[5, 8], 3, [2, [4, 7], 9, [2]]]  levelled: [5, 8, 3, 2, 4, 7, 9, 2]
[[4, []], [[[6], 7], [8]], [2], [[[]]]] levelled: [4, 6, 7, 8, 2]
```

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 7 exercise 4.

Exercise 5 (3 marks).

Use a recursive solution to complete the `get_coins_required()` function which is passed:

a goal amount (in cents),
the number of available big coins
and,
the number of available small coins.

A big coin is worth 25 cents and a small coin is worth 5 cents. The function returns a tuple containing the number of big coins and the number of small coins which are required to make up the goal amount. If the amount cannot be achieved with the coins available, the function returns the tuple (-1, -1).

Note the your solution should use as many big coins as needed to make up the goal amount., i.e., if a goal amount can be achieved using 3 big coins and 1 small coin or using 2 big coins and 6 small coins, then the tuple returned by the function should be (3, 1).

The following code:

```
goal = 65
big_small = get_coins_required(goal, 2, 7)
print(big_small)
```

prints:

```
(2, 3)
```

To test your completed implementation, run the `CoinsRequired` application. The output produced should be:

```
50 cents: (1,5)
110 cents: (3,7)
205 cents: (8,1)
550 cents: (22,0)
5000010 cents: (200000,2)
20 cents: (0,4)
Not possible!
65 cents: (2,3)
Not possible!
```

Once you are happy that your function is correctly coded, insert the function code into CodeRunner, assignment 7 exercise 5.

Exercise 6 (1 mark). Participation in the code review of the programming exercises.