# Homework 4

*Due 4:00pm, Wednesday, February 27, 2019*

## Installation and Handin

**Homework Setup.** For each homework assignment, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a Brown CS Terminal window. For this homework type:

```
cs4_install hw04
```

There should now be a `hw02` folder within your homeworks directory. Using Terminal, you can move into the `hw02` folder with the cd command:

```
cd ~/course/cs0040/homeworks/hw04
```

**Homework hand-in.** Be sure to turn in all the files requested and that they are named exactly as specified, including spelling and case. When you're ready to submit the files, run:

```
cs4_handin hw04
```

from a Brown CS Terminal window from your `~/course/cs0040/homeworks/hw04` directory. The entire contents of your `~/course/cs0040/homeworks/hw04` directory will be handed in. Check for a confirmation email to ensure that your assignment was correctly submitted using the `cs4_handin` command. You can resubmit this assignment using the `cs4_handin` command at any time, but be careful, as only your most recent submission with be graded.

## Part I: Tracing Program Flow (10)

Place your answers for Part 1 in the plain-text file named `hw04_1.py`.

### Problem 4.1a) Global / Local Variables

Consider the following Python program:

```
def foo(a):
    b = a * 3
    print('foo:', a, b)
    return b

def bar(b):
    a = foo(b) + foo(b // 2)
    print('bar:', a, b)
    return a

a = 2
b = 5
print(a, b)
b = bar(b)
print(a, b)
a = foo(a)
print(a, b)
```

Refer to the tables in the stencil for this problem, and complete them to illustrate the execution of the program.

```
global variables (ones that belong to the global scope)
  a  |  b
-----------
  2  |  5
```

```
     |
```

```
bar’s local variables
  a  |  b
-----------
     |
     |
```

```
foo’s local variables
  a  |  b
-----------
     |
     |
```

```
output (the lines printed by the program)
------
2 5
```

Note that you must include four tables: one for the global variables (the ones that belong to the global scope), one for the local variables that belong to the function foo, one for the local variables that belong to the function `bar`, and one for the output of the program. *Don’t forget to follow the rules for variable scope.* We have started the first and fourth tables for you. You should:

- complete the first three tables so that they illustrate how the values of the variables change over time
- complete the last table so that it shows the output of the program (i.e. the values that are printed).

## Problem 4.1b) Recursive Flow

Consider the following recursive function:

```
def mystery(a, b):
    if a < 0:
        return 1
    else:
        myst_rest = mystery(a - b, b)
        return 2 + myst_rest
```

1. Trace the execution of `mystery(20, 6)`. You may use any reasonable approach (e.g. a table as in part 1 or the indented style from lecture) provided that you show *all* of the calls to the function.
2. What is the value returned by `mystery(20, 6)`?
3. During the execution of mystery(20, 6), stack frames are added and then removed from the stack. How many stack frames are on the stack when the base case is reached? You should assume that the initial call to mystery(20, 6) is made from the global scope, and you should include the stack frame for the global scope in your count.
4. Give an example of specific values of a and b that would produce infinite recursion, and explain why it would occur.

You may find it helpful to use Python Tutor to build intuition about how recursion works, to check your answers to parts 1-3 of this problem, and to test and debug the functions that you write in the later problems.

## Part II: First Recursions (30)

Create a file called `hw04_2.py` in the `hw04` directory and place your solutions (**including test cases**) in this file. Please refer to the following guidelines for creating your own Python files:

- **Use the Test Driven Design approach when you create your function. See Lecture 3 for more details. All your major functions must include test cases.**
- Your functions must have the **exact names specified** below, or we won't be able to test them. Note in particular that the case of the letters matters (all of them should be lowercase), and that some of the names include an underscore character ( _ ).
- As usual, each of your functions should **include a docstring** that describes *what your function does* and *how to use it*, including *what its inputs are*.
- Make sure that your functions *return* the specified value, rather than printing it. **None of these functions should use a `print` statement.**
- Unless the problem states otherwise, the **functions that you write must use recursion**. They may *not* use any iterative constructs that you may be aware of (`for`, `while`, etc.), and they may *not* use list comprehensions and they may not use `filter`, `map`, or `reduce`.
- **More generally, you should *not* use any Python features that we have not discussed in lecture or read about in the textbook.**
- Your functions **do *not* need to handle bad inputs** – inputs with a type or value that doesn't correspond to the description of the inputs provided in the problem.
- Leave **one or two blank lines** between functions to make things more readable.
- Here again, we encourage you to use the Python Tutor visualizer, as described at the end of the previous problem.
- Please please be sure to run your code before submitting! If it doesn't work on your computer, it won't work on ours. Check Piazza if you don't know how to do this!

## Problem 4.2a) Compare Lists

Write a function `compare(babyList, puppyList)` that takes as inputs two lists of numbers of babies or puppies, `babyList` and `puppyList`, and that *uses recursion* to compute and *return* the number of values in `babyList` that are larger than their corresponding value in `puppyList`. In other words, the function should compare `babyList[0]` with `puppyList[0]`, `babyList[1]` with `puppyList[1]`, `babyList[2]` with `puppyList[2]`, etc, and it should return the number of positions at which the value from `babyList` is larger than the value from `puppyList`. For example:

```
>>> compare([5, 3, 7, 9], [2, 4, 7, 8])
2
```

The above call returns `2`, because:

- in position 0, 5 > 2
- in position 1, 3 < 4
- in position 2, 7 == 7
- in position 3, 9 > 8

Thus, there are two positions (0 and 3) at which the value from `babyList` is larger than the value from `puppyList`.

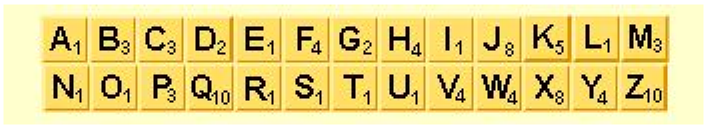
**Note:** It is possible for the two lists to have different lengths, in which case the function should only compare the positions at which the two lists both have values.

```
>>> compare([4, 2, 3, 7], [1, 5])      # 4 > 1; 2 < 5; can't compare 3 or 7
1
>>> compare([4, 2, 3], [1, 5, 0, 8])   # 4 > 1; 2 < 5; 3 > 0; can't compare 8
2
>>> compare([5, 3], [])
0
>>> compare([], [5, 3, 7])
0
```

## Problem 4.2b) Scrabble Tile

Write a function `letter_score(letter)` that takes a lowercase letter as input and *returns* the value of that letter as a Scrabble tile. If `letter` is not a lowercase letter from `'a'` to `'z'`, the function should return 0. **This function does *not* require recursion.**

Here is the mapping of letters to scores that you should use:



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A_1$ | $B_3$ | $C_3$ | $D_2$ | $E_1$ | $F_4$ | $G_2$ | $H_4$ | $I_1$ | $J_8$ | $K_5$ | $L_1$ | $M_3$ |
| $N_1$ | $O_1$ | $P_3$ | $Q_{10}$ | $R_1$ | $S_1$ | $T_1$ | $U_1$ | $V_4$ | $W_4$ | $X_8$ | $Y_4$ | $Z_{10}$ |

You will need to use conditional execution (`if-elif-else`) to determine the correct score for a given letter.

Using the mapping above, here are some examples of expected output:

```
>>> letter_score('w')
4
>>> print(letter_score('q'))
10
>>> letter_score('%')      # not a letter
0
>>> letter_score('A')      # not lower-case
0
```

## Problem 4.2c) Scrabble Score

Write a function `scrabble_score(word)` that takes as input a string `word` containing only lowercase letters and ***uses recursion*** to compute and *return* the Scrabble score of that string – i.e., the sum of the Scrabble scores of its letters. For example:

```
>>> scrabble_score('python')
14
>>> scrabble_score('a')
1
>>> scrabble_score('babies')
10
>>> scrabble_score('puppies')
13
```

Use your `letter_score` function and recursion to solve this problem. You may find it helpful to use the `num_vowels` function from lecture as a model. In that function, every vowel essentially had a value of 1, and every consonant had a value of 0. In this function, each letter has its letter score as its value.

## Part III: More Recursion (30)

Place your solutions (including test cases) to Part III into `hw04_3.py`. The same guidelines described at the beginning of Part II apply here as well. These functions should look familiar! They are! But this time, we'll be using recursion instead of iteration :)

### Problem 4.3a) Double String

Write a function `double(s)` that takes an arbitrary string `s` as input, and that *uses recursion* to construct and *return* the string formed by doubling each character in the string. Here are three examples:

```
>>> double('hello')
'hheellllloo'
>>> double('puppy')
'ppuuppppyy'
>>> double('')
''
```

**Note:** you will need more than one base case.

### Problem 4.3b) Find Index

Write a function `index(elem, seq)` that takes as inputs an element `elem` and a sequence `seq`, and that *uses recursion* to find and *return* the zero-based index of the first occurrence of `elem` in `seq`. If `elem` is not an element of `seq`, `index` should return -1. **You are not allowed to use the `in` operator in this function.**

The sequence `seq` can be either a list or a string. If `seq` is a string, `elem` will be a single-character string; if `seq` is a list, `elem` can be any value. For example:

```
>>> index(5, [4, 10, 5, 3, 7, 5])
2
>>> index('hi', ['well', 'hi', 'there'])
1
>>> index('b', 'babies')
0
>>> index('a', 'babies')
1
>>> index('k', 'puppies')
```

```
-1
>>> index('hi', ['hello', 111, True])
-1
>>> index('a', '')     # the empty string
-1
>>> index(42, [])      # the empty list
-1
```

**Note:** In the recursive case, the decision about what to return will need to take into account the result of the recursive call. As a result, it is especially important that you store in a variable the value that is returned by the recursive call.

## Part IV: A Taste of Higher Order Functions: Map (20)

Place your solutions (including test cases) to Part IV into `hw04_4.py`

### Problem 4.4a) DIY Map

Next week, we will learn about three higher order functions: `filter, map` and `reduce.` These functions are incredibly useful because they can simplify a good amount of code into just one line, but they're tricky to completely grasp. In this portion of the homework, you'll implement your own version of one of these functions, `map`. Your version will be called `my_map`.

The `map` function works by taking in a function `f` and a list `seq`. It then returns a new list in which `f` has been applied to every element of `seq`. Be sure to use the Test Driven Design and the Recursive Design Recipe while developing your functions.

Here are the function signatures you should use:

```
def my_map(f, seq):
    """
    Returns function f applied to each element of seq.  Works on lists and
    strings, but always returns a list.
    """
    # ...
```

## Problem 4.4b) Using Map

Hm. One of the functions we've written for this homework sounds like it could be written using your `map` function. Figure out which recursion problem from this homework can be solved using `map`, and use your `my_map` function to solve that problem.

**Important:** You should not use explicit recursion in your solutions.

If (and only if) you can not make your `my_map` from Problem 4.4a work, you may use the built-in `filter`, `map`, and `reduce` functions, but note that doing so will result in a loss of credit for Problem 4.4a.

## Testing

You are required to write test cases for every function you implement in this homework. Be sure to develop (or include) appropriate a set of test cases for each problem you solve. **Note that if for any function your code does not compile, you can at most receive half credit on that problem.** You have to not only write your tests, but run them as well!

---

*Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.*