# CSC148 Summer 2018: Lab 5

## Introduction

The goals of this lab are:

- To get you  familiar with how recursion works
- To give you practice in identifying base cases
- To give you practice in identifying your recursive step
- To give you practice in writing recursive functions

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

## General Lab Notes

1. Make sure you have lab_pyta.txt downloaded and placed in the directory (or directories) where you'll be working.
2. To use PythonTA, include the following code (if you already have a main block, just add the body to the end of it):

```
if __name__ == '__main__':
    import python_ta
    python_ta.check_all(config="lab_pyta.txt")
```

Your lab_pyta.txt should be in the same folder as the .py files you're running. PythonTA will raise errors regarding style, specifying the lines you need to fix. You should get familiar with what the errors mean, and how to fix them: this will be important for your exercises and assignments.

## Getting Started

Below are various functions for you to implement. You should be implementing these recursively, even if you know how to solve it using only loops.

## count_odd(x)

`x` is either an int or a (possibly nested) list of lists/ints. `count_odd` should return the number of odd ints that appear in `x`. Before writing any code, answer the following:

1. What is the base case for `count_odd`? In what case do you not have a recursive call and can simply return a value?
2. Suppose we called `count_odd([1, [2, 3], [4, [5], [[6, 7], 8], 9]])`: What would we expect this to return?
3. If `x` is `[1, [2, 3], [4, [5], [[6, 7], 8], 9]]`, what recursive calls would we want to make?
4. Assuming the recursive calls worked perfectly, what do you expect each call to return?
5. How would you use the results of the recursive calls from (4) to get what we want to return in (2)?

Once you've answered the above, implement `count_odd`. Put the following example calls in your `if __name__ == '__main__'` block to make sure `count_odd` is working correctly:

```
assert count_odd(1) == 1
assert count_odd(2) == 0
assert count_odd([1, 3, 4]) == 2
assert count_odd([[1, 5, [[4, 6], 7]], 9]) == 4
assert count_odd([1, [2, 3], [4, [5], [[6, 7], 8], 9]]) == 5
```

## count_longer_than(x, length)

`x` is either a string or a (possibly nested) list of lists/strings. `count_longer_than` should return the number of strings with a length > `length` that appear in `x`. Before writing any code, answer the following:

1. What is the base case for `count_longer_than`? In what case do you not have a recursive call and can simply return a value?
2. Suppose we called `count_longer_than(['a', [['baby'], 'cat'], [['doll'], 'hat', [['cake'], 'hats']]], 3)`: What would we expect this to return?
3. If `x` is `['a', [['baby'], 'cat'], [['doll'], 'hat', [['cake'], 'hats']]]` and `length` is 3, what recursive calls would we want to make?
4. Assuming the recursive calls worked perfectly, what do you expect each call to return?
5. How would you use the results of the recursive calls from (4) to get what we want to return in (2)?

Once you've answered the above, implement `count_longer_than`. Put the following example calls in your `if __name__ == '__main__'` block to make sure `count_longer_than` is working correctly:

```
assert count_longer_than('cat', 3) == 0
assert count_longer_than('cat', 2) == 1
assert count_longer_than(['', 'a', 'at', 'hat'], 1) == 2
assert count_longer_than([['yes', 'no', [['ok', 'hat'], 'cat']], 'a'], 2) == 3
assert count_longer_than(['a', [['baby'], 'cat'], [['doll'], 'hat', [['cake'],
'hats']]], 3) == 4
```

## get_max_depth(x)

`x` is anything. We define the 'depth' of an object to be how many lists it's nested inside. For example:

- The int 5 has a depth of 0 since it's not nested in anything.
- The int 5 in [5] has a depth of 1 since it's nested inside 1 list.
- The int 5 in [[5], 4] has a depth of 2 since it's nested inside 2 lists, while 4 has a depth of 1 since it's only nested in 1 list.
- The int 5 in [[[5], 4]] has a depth of 3 since it's nested inside 3 lists, while 4 has a depth of 2 since it's nested in 2 lists.

`get_max_depth` should return the depth of the most nested item in `x`. Before writing any code, answer the following:

1. What is the base case for `get_max_depth`? In what case do you not have a recursive call and can simply return a value?
2. Suppose we called `get_max_depth([1, [2, [3]], [[[4]], 5]])`: What would we expect this to return?
3. If `x` is `[1, [2, [3]], [[[4]], 5]]`, what recursive calls would we want to make?
4. Assuming the recursive calls worked perfectly, what do you expect each call to return?
5. How would you use the results of the recursive calls from (4) to get what we want to return in (2)?

Once you've answered the above, implement `get_max_depth`. Put the following example calls in your `if __name__ == '__main__'` block to make sure `get_max_depth` is working correctly:

```
assert get_max_depth(5) == 0
assert get_max_depth([1, 2, 3]) == 1
assert get_max_depth([[1], 2]) == 2
assert get_max_depth([1, [[3]], 8]) == 3
assert get_max_depth([1, [2, [3]], [[[4]], 5]]) == 4
```

## get_at_depth(x, depth)

`x` is anything. We define the 'depth' of an object to be how many lists it's nested inside. For example:

- The int 5 has a depth of 0 since it's not nested in anything.
- The int 5 in [5] has a depth of 1 since it's nested inside 1 list.
- The int 5 in [[5], 4] has a depth of 2 since it's nested inside 2 lists, while 4 has a depth of 1 since it's only nested in 1 list.
- The int 5 in [[[5], 4]] has a depth of 3 since it's nested inside 3 lists, while 4 has a depth of 2 since it's nested in 2 lists.

`get_at_depth` should return all of the items that have depth == `depth` in `x`. Before writing any code, answer the following:

1. What is the base case for `get_at_depth`? In what case(s) do you not have a recursive call and can simply return a value?
2. Suppose we called `get_at_depth([1, [2, [3]], [[[4]], 5]], 2)`: What would we expect this to return?
3. If `x` is `[1, [2, [3]], [[[4]], 5]]` and `depth` is 2, what recursive calls would we want to make?
   **HINT:** We'll want to adjust `depth` for our recursive calls. Why?
4. Assuming the recursive calls worked perfectly, what do you expect each call to return?
5. How would you use the results of the recursive calls from (4) to get what we want to return in (2)?

Once you've answered the above, implement `get_at_depth`. Put the following example calls in your `if __name__ == '__main__'` block to make sure `get_at_depth` is working correctly:

```
assert get_at_depth(5, 0) == [5]
assert get_at_depth(5, 1) == []
assert get_at_depth([1, 2, 3], 1) == [1, 2, 3]
assert get_at_depth([[1], 2, [3], 4], 1) == [2, 4]
assert get_at_depth([1, [[3], 2, [4]], 8, [[5]]], 3) == [3, 4, 5]
assert get_at_depth([1, [2, [3]], [[[4], 6], 5]], 3) == [3, 6]
```