Lab 6 : Recursion

In this lab, we will discuss recursion, tail recursion, and memoization. We will follow along with two examples: factorial and only_ints.

At the beginning of the lab, you all need to have lab6.py file in your repository.

The file will be on *Canvas*, for students to download to their repo:

- Laboratory/Lab6/lab6.py

❑ Key Concepts

❑ What is factorial?

Factorial is given as n! where n is some number greater than 0. This results in the multiplication of all numbers from 1 to n together. So, 4! is 4*3*2*1.

Recursion

Recursion is used to define structures inductively. Recursion is equivalent to an unbounded loop. Recursion can be broken apart as follows:

- Stopping Point (the base case)
- The loop (the inductive step)

The inductive step handles the following:

- What to do with the data
- How to make the data smaller

Inside of the inductive step, you normally make the data smaller (or larger depending on the problem).

The recursive functions we are looking at here are pretty simple -- 4 to 6 lines at most. This helps introduce the concept to you -- but not all recursion is this simple! There are a couple of ways to complicate it (e.g., the parameters might have extra stuff, or multiple inductive steps, etc.)

- We have the equation

```
Base Case:
    When n = 0, 1
    Otherwise, n * factorial(n - 1)
```

- To write it recursively,

```
def factorial(n):
    if n == 0:
        return 1 # return n
    else:
        return n * factorial(n-1) # This line can be done on another line
```

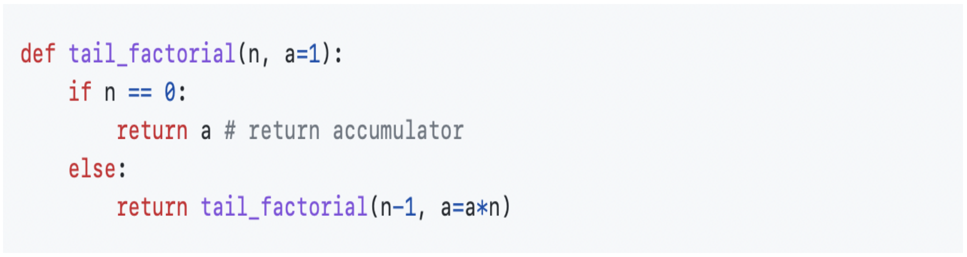
- Notice that the math (in this case, taking the product of n and subsequent values) takes place in the return statement, not within the function call. This means that in order for the outer function to return, we need the recursive call to return first, since it's a prerequisite to the outer function. Then, the math actually gets done once every recursive call down has returned.

- **Example of how to work on only_ints**

```
def only_ints(xlist):
    if xlist == []:
        return []
    elif type(xlist[0]) != int:
        return [] + only_ints(xlist[1:])
    else:
        return [xlist[0]] + only_ints(xlist[1:])
```

❑ **Tail Recursion factorial**

- Tail recursion is like regular recursion without taking up too much memory in a recursion stack ("recurring too deep"). It is often the case that tail recursion uses the same arguments as the regular recursive function, with an added accumulator (usually using a defaulted argument, so the end user doesn't have to define a base accumulator). The accumulator is returned when the base case is reached.



```python
def tail_factorial(n, a=1):
    if n == 0:
        return a # return accumulator
    else:
        return tail_factorial(n-1, a=a*n)
```

- Notice the difference between the recursive case return statement in this and the previous function. Since the math occurs *inside* the recursive function call (look for the "splat", as Dr. D calls it), the math is done progressively, at each step rather than all at the end.

→ **Now try to Work on tail_only_ints**

- ❑ **Memoization factorial**

Definition:

"Technique of saving values that have already been computed"

- When you are writing some programs, you will often repeat calculations over and over again (subproblems). As you go through the calculation, you store the data, so that way you do not waste time solving the same problem twice. You store the values in some sort of data structure to keep track of what parameters were used and what the outcome was.
- You probably do this already when you add up a series of numbers. If asked what 5+4+3 equals, you'd take a minute to find the answer: 12. If then asked what 5+4+3+1 equals, you'd only need a second to add 1 to your previous answer and respond with 13, because you've already done the "heavy" lifting of 5+4+3.
- This can substantially shrink the solution space of a problem. If you face the same situation twice, you can simply retrieve what you'd previously calculated, rather than recalculating this subproblem.

```
d = {}
def memo_factorial(n):
    if n not in d.keys():
        if n == 1:
            d[n] = 1
        else:
            d[n] = n * memo_factorial(n-1)
    return d[n]
```

→ **Now try to work on memo_only_ints**