Lab 10: Functional Programming

INFSCI 0201
Intermediate Programming (Spring 2025)

It's all about functions now.
Remember, No Objects!
This week's lab tests your knowledge of basic functional programming ideas. You need to complete two tasks using functional programming principles.

Submission

You are to write a complete Python program that meets the requirements outlined in the Lab 10 Tasks section.

- As with all programming assignments in this course, you must submit Lab 10 through GitHub in the git repository the instructor created for you. You must place your Lab 10 eclipse project in the folder **/labs/lab10/**
- The code for each task should be in their separate Python file.

Testing

The main test for this lab is slightly different from earlier labs.

- First, your programs should be able to run without any issues and generate the expected output.
- Second, in task one, you are required to use the functional programming paradigm to finish the task. Be careful to make sure you are using pure functions in your code and all the data/variables are immutable.
- Third, for task two, there is a speed comparison against a standard recursive implementation. Your decorator decorated function should be considerably fast in comparison.

Note: Style guide violations will count as test failures! Be sure you're strictly adhering to the Python coding style guide on Canvas.

Lab 10 Tasks

Task 1: Remember the modified CaesarCipher you created using the principle of object-oriented programming back in lab 3?

In this task, you will rewrite your program to adhere to the principle of functional programming. So no more classes, only functions. And no more mutable variables.

To achieve this, you can refactor your code in Lab 3 or write a completely new functional programming version of the CaesarCipher.

For the detailed requirements, please refer to the instruction document for Lab 3.

Task 2: **Decorators for Cache or Memoization**
A standard recursive implementation of generating a Fibonacci Sequence would look like the following:

```
def recur_fibo(n):
   if n <= 1:
       return n
   else:
       return(recur_fibo(n-1) + recur_fibo(n-2))
```

However, running recursively takes a lot of computational resources. Especially in this case, the recursive execution will have a lot of unnecessary calculations during the execution. A good optimization route is to cache or memorize the calculated numbers, saving you a significant amount of computational time.
So for a sequence like 1, 1, 2, 3, 5, 8, 13…, you don't have to calculate the fourth number (3) again and again when calculating the following numbers. Once is enough as long as your code memorizes them.
Your task is to make a decorator that decorates the recur_fibo function to serve as the cache or memory for all the calculated numbers.

You should include an execution speed comparison between the decorated recur_fib() and the original recur_fib() when n = 35. You should observe a significant speed improvement.