CECS 342
Lab assignment 4
Due date: Friday, March 29
20 points

A decorator is a function that takes another function and extends the behavior of the latter function at run time without explicitly modifying it.

```
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = decorator(say_whee)
```

Here, you've defined two regular functions, decorator() and say_whee(), and one inner wrapper() function. Then you redefined say_whee() to apply decorator() to the original say_whee().

Can you guess what happens when you call say_whee()? Try it in a REPL. Instead of running the file with the -i flag, you can also import the function manually:

```
>>> from hello_decorator import say_whee

>>> say_whee()
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

The so-called decoration happens at the following line:

```
say_whee = decorator(say_whee)
```

In effect, the name `say_whee` now points to the `wrapper()` inner function. Remember that you return `wrapper` as a function when you call `decorator(say_whee)`:

However, wrapper() has a reference to the original say_whee() as func, and it calls that function between the two calls to print().

Put simply, *a decorator wraps a function, modifying its behavior.*

Python allows you to *use decorators in a simpler way with the @ symbol*, sometimes called the pie syntax. The following example does the exact same thing as the first decorator example:

```
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@decorator
def say_whee():
    print("Whee!")
```

So, @decorator is just a shorter way of saying say_whee = decorator(say_whee). It's how you apply a decorator to a function.

Problem 1 (10 points)

Assume you have some decent amount of functions having separate tasks:

```
def  do_A():
  message = "Do A"
  return message
def do_B():
  message = "Do B"
  return message
def do_C():
  message = "Do C"
  return message
```

You want to add authorization on each of the function. So you do something like this:

```
def is_authorized():
  return True
def Do_A():
    if not is_authorized():
        message = "you are not authorized"
    else:
        message = "Do A"
    return message
def Do_B():
    if not is_authorized():
        message = "you are not authorized"
    else:
        message = "Do B"
    return message

def Do_C():
    if not is_authorized():
        message = "you are not authorized"
    else:
        message = "DO C"
    return message
```

As you are checking authorization in each of your function, isn't it redundant, repeated and tiresome and also violates the DRY principle? Rewrite the code using Python decorator. You run the decorator code only during the day.

Grading;

1. Each team member submits the following files to Canvas:

   a. Lab41.py file (-10 points for no submission)
   b. A pdf file which has a copy of lab4.py and runtime output.(- 5 points for missing the pdf file and runtime output)

2. Write on the comment the completion of the lab assignment for each team member. All the team members must agree with the completion of each team member.

   The program run successfully with correct output (100% completion)
   The program run successfully with incorrect output (60% completion)
   The program has syntax errors or is incomplete (40% completion)
   The program has few coding and syntax errors (30% completion)
   There is no submission (0%)

Problem 2 [10 points]

Create a Python program that schedules to call three functions named "Factorial" **concurrently** using async, await, asyncio.sleep(), asyncio.gather, and asyncio.run().

The function Factorial has two format parameter:

- name: the task name associated to the function Factorial
- number: the number to determine a Factorial value

<u>Pseudocode</u>:

Define the function factorial (name, number) as async:

```
factorial(name, number):
    loop from 1 to number +1
        display the task name, currently factorial value, current loop
        variable
        delay 1 second

    display the task name and the final factorial value
    return the final factorial value
```

Define the main function as async:

```
Schedule three calls "factorial" concurrently using asyncio.gather
Print the result
Use asyncio to run the main function
```

<u>Output sample</u>

```
Task A: Compute factorial(3), currently i=1...
Task B: Compute factorial(4), currently i=1...
Task C: Compute factorial(5), currently i=1...
Task A: Compute factorial(3), currently i=2...
Task B: Compute factorial(4), currently i=2...
Task C: Compute factorial(5), currently i=2...
Task A: Compute factorial(3), currently i=3...
Task B: Compute factorial(4), currently i=3...
Task C: Compute factorial(5), currently i=3...
Task A: factorial(3) = 6
Task B: Compute factorial(4), currently i=4...
Task C: Compute factorial(5), currently i=4...
Task B: factorial(4) = 24
Task C: Compute factorial(5), currently i=5...
Task C: factorial(5) = 120
[6, 24, 120]
```

Grading;

1. Each team member submits the following files to Canvas:

   a. Lab42.py file (-10 points for no submission)
   b. A pdf file which has a copy of lab42.py and runtime output.(- 5 points for missing the pdf file and runtime output)

2. Write on the comment the completion of the lab assignment for each team member. All the team members must agree with the completion of each team member.

   The program run successfully with correct output (100% completion)
   The program run successfully with incorrect output (60% completion)
   The program has syntax errors or is incomplete (40% completion)
   The program has few coding and syntax errors (30% completion)
   There is no submission (0%)