

# Functional Programming Lab 04

K M Anisul Islam

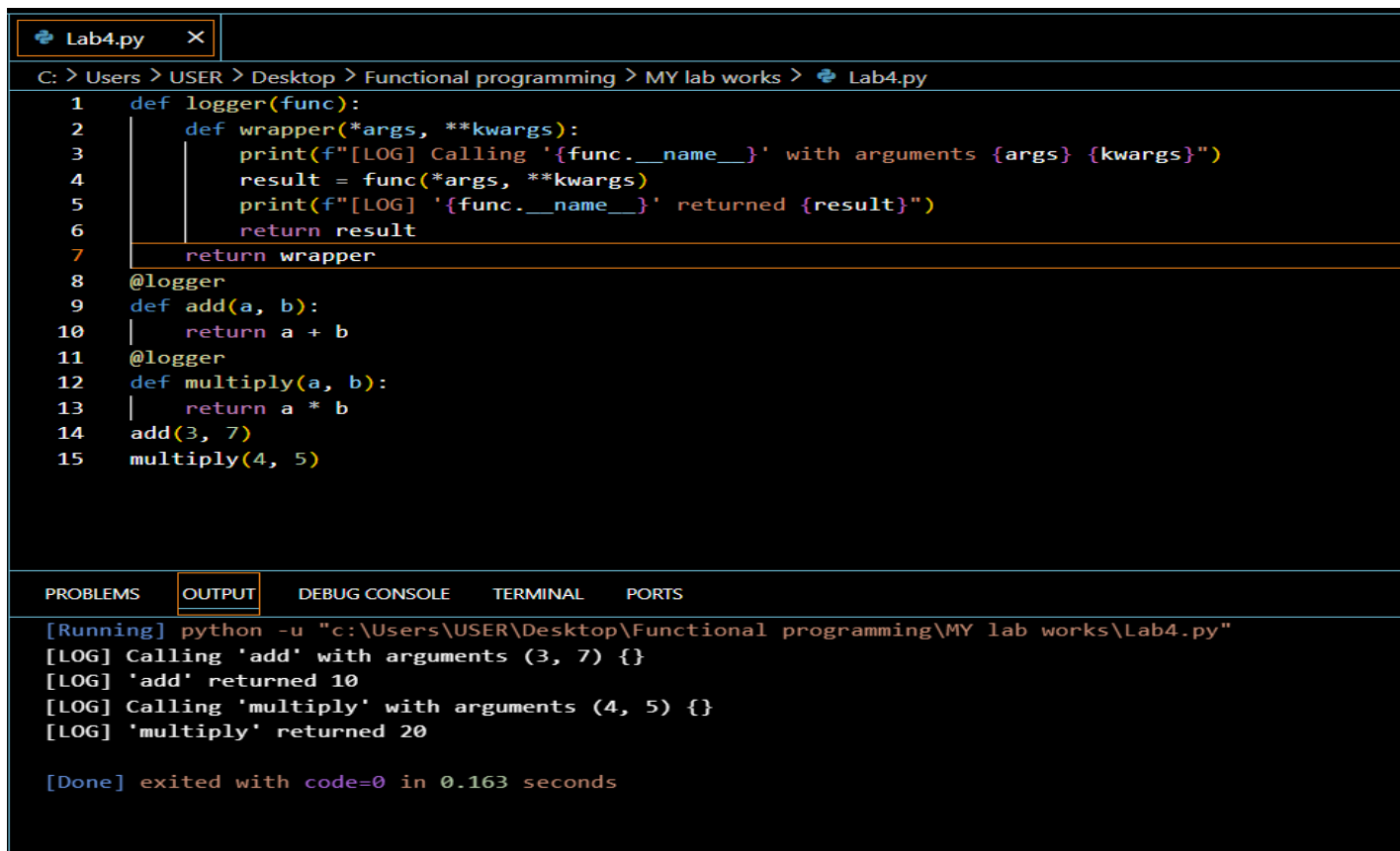
Email: [ISLAM.K@stud.satbayev.university](mailto:ISLAM.K@stud.satbayev.university)

**Objective:** To study and apply advanced recursive techniques and functional patterns in Python. The purpose of this laboratory work is not only to improve students' skills in writing recursive code but also to understand and apply various functional patterns that enhance readability, efficiency, and scalability of programs.

## Individual Task:

Functional Pattern "Decorator" for Logging

Implement a functional pattern using Python decorators that logs the function name, its arguments, and the returned result. This helps in monitoring the execution flow of programs and improves debugging and testing capabilities



```
Lab4.py X
C: > Users > USER > Desktop > Functional programming > MY lab works > Lab4.py
1 def logger(func):
2     def wrapper(*args, **kwargs):
3         print(f"[LOG] Calling '{func.__name__}' with arguments {args} {kwargs}")
4         result = func(*args, **kwargs)
5         print(f"[LOG] '{func.__name__}' returned {result}")
6         return result
7     return wrapper
8 @logger
9 def add(a, b):
10     return a + b
11 @logger
12 def multiply(a, b):
13     return a * b
14 add(3, 7)
15 multiply(4, 5)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[Running] python -u "c:\Users\USER\Desktop\Functional programming\MY lab works\Lab4.py"
[LOG] Calling 'add' with arguments (3, 7) {}
[LOG] 'add' returned 10
[LOG] Calling 'multiply' with arguments (4, 5) {}
[LOG] 'multiply' returned 20

[Done] exited with code=0 in 0.163 seconds
```

1. What are the main principles and techniques used in advanced recursion?  
It uses techniques like tail recursion, where the recursive call is the last operation in a function, and memoization, which stores previously computed results to avoid redundant calculations.
2. How can recursive functions be optimized to prevent stack overflow?  
Recursion can be optimized using tail recursion (though Python doesn't optimize it automatically) and memoization to reduce redundant calls. Sometimes, converting recursion to iteration is the best optimization.
3. What is the functional pattern "Decorator," and how is it applied in Python?  
A decorator is a higher-order function that takes another function and extends its behavior without altering its source code. It's commonly used for logging, authorization, and caching.
4. What are the benefits of lazy evaluation in Python, and how is it implemented?  
It delays computation until the result is needed, improving performance and memory use. It is implemented using generators and the yield.
5. Can you explain how to implement and use the "Strategy" pattern?  
The Strategy pattern defines a family of algorithms and allows selecting one at runtime. In Python, this is usually done by passing functions or objects implementing different algorithms as arguments.
6. What challenges may arise when working with recursion, and how can they be resolved?  
Common challenges include stack overflow and redundant computations. These can be solved using memoization, tail recursion, or by converting the recursive logic into iteration.
7. Provide an example of using the "Observer" pattern in programming.  
Such as when a UI element updates automatically when data changes (like listeners in GUI frameworks).
8. Which types of problems are most effectively solved using recursion?  
repetitive subproblems, such as tree traversal, factorial computation, and searching algorithms (like DFS).
9. How does recursion affect program performance and memory usage?  
Recursion uses the call stack for each function call, which can increase memory usage and slow performance if not optimized.
10. How can functional patterns improve software design and architecture?  
Functional patterns encourage modularity, reusability, and immutability, resulting in cleaner, more testable, and more maintainable code.