

Name :B. Shivani Reddy

Hall ticket no:2303A53025

## Task 1: AI-Generated Logic Without Modularization

### *Fibonacci Sequence Without Functions*

#### Scenario

A quick Fibonacci sequence generator is required for a learning platform prototype.

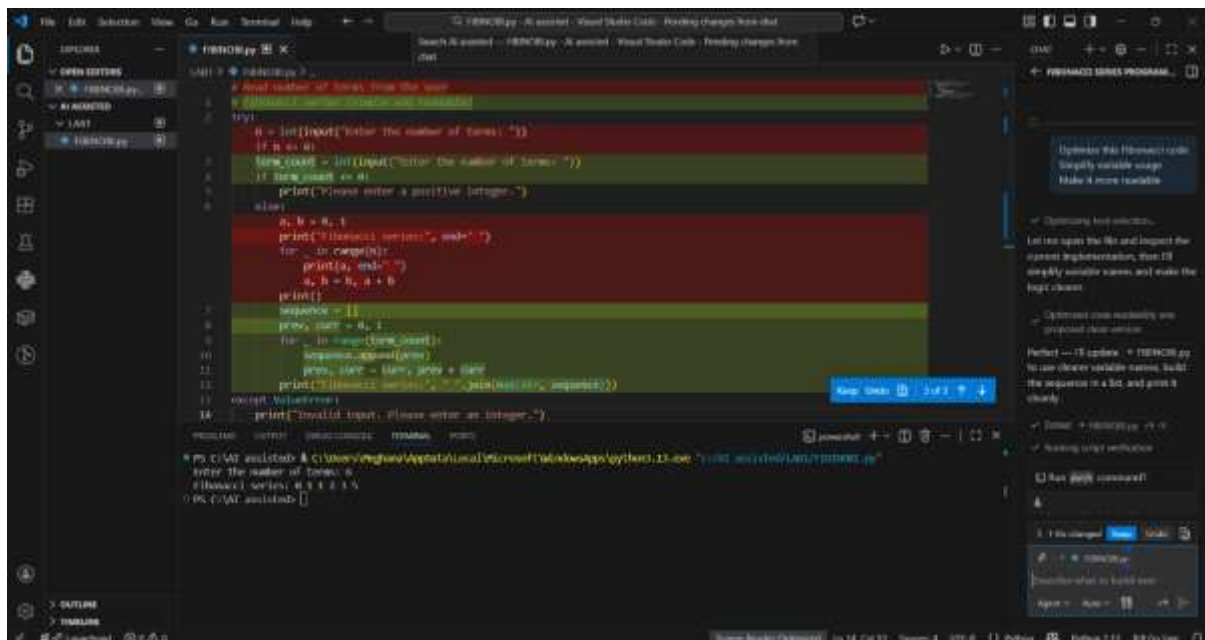
#### Prompt Given to Copilot

# Write a Python program to print Fibonacci series up to n terms

# Take user input

# Do not use functions

#### Copilot-Generated Code



```
1 # Fibonacci Series Program
2
3 # Read number of terms from the user
4 n = int(input("Enter the number of terms: "))
5
6 # Initialize first two terms
7 a, b = 0, 1
8
9 # Print the first two terms
10 print(a, end=" ")
11 print(b, end=" ")
12
13 # Generate the rest of the series
14 for i in range(2, n):
15     c = a + b
16     print(c, end=" ")
17     a, b = b, c
18
19 # Print a newline at the end
20 print()
21
22 # Handle invalid input
23 if n < 1:
24     print("Invalid input. Please enter a positive integer.")
```

#### Sample Input

Enter number of terms: 7

#### Sample Output

0 1 1 2 3 5 8

## Task 2: AI Code Optimization & Cleanup

### Original Code Issues

- Extra temporary variable (c)
- Repeated reassignment logic

- Less readable variable naming
- Initial values printed separately

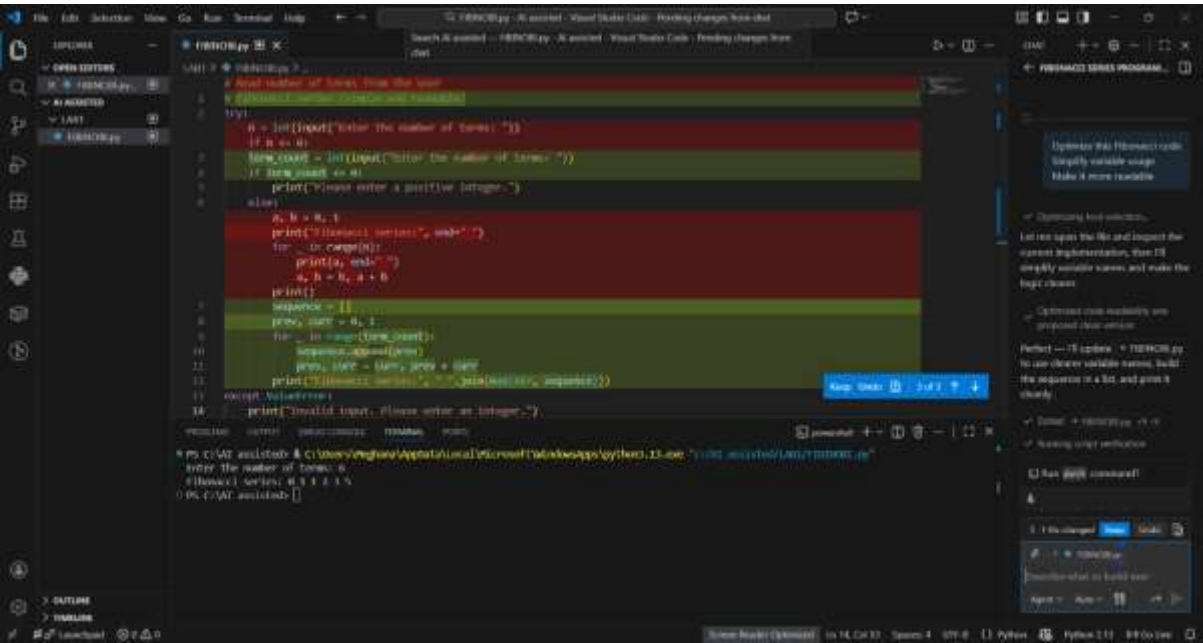
Prompt Given to Copilot

Optimize this Fibonacci code

Simplify variable usage

Make it more readable

Optimized Code



Improvements Explained

| Aspect      | Original Code                | Optimized Code    |
|-------------|------------------------------|-------------------|
| Variables   | Multiple temporary variables | Minimal variables |
| Readability | Moderate                     | High              |
| Loop Logic  | Complex                      | Simplified        |
| Performance | Same complexity              | Cleaner execution |

Conclusion:

The optimized version improves readability, reduces redundancy, and follows Pythonic coding practices while maintaining the same time complexity.

Task 3: Modular Design Using AI Assistance

Fibonacci Using Functions

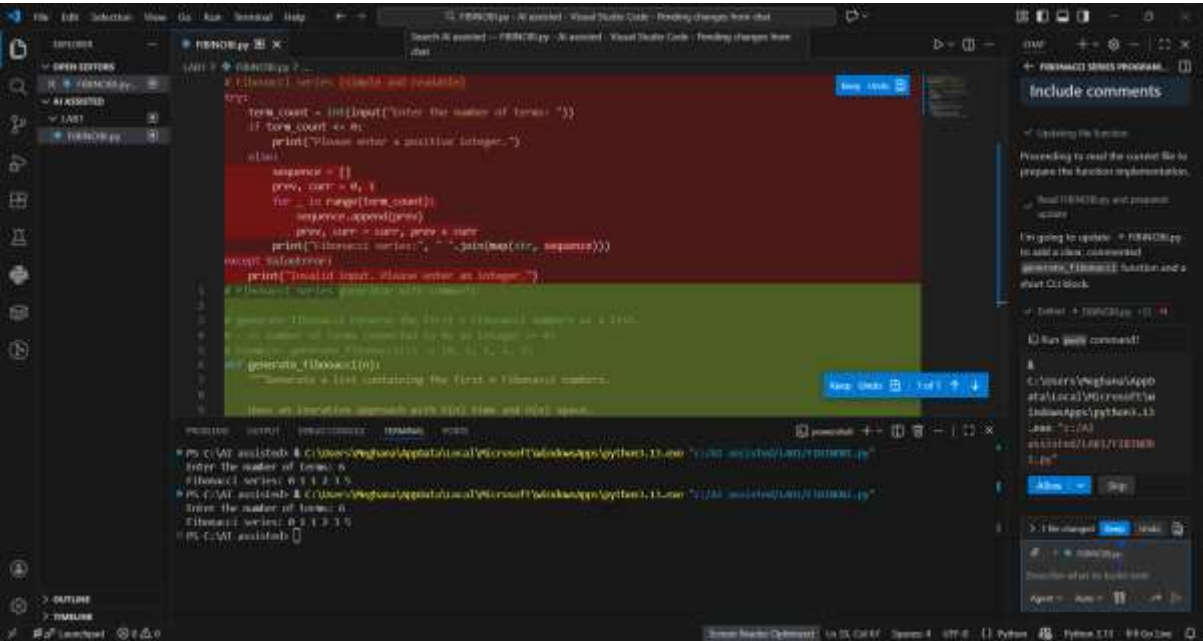
Scenario

The Fibonacci logic must be reused across multiple modules.

**Prompt Given to Copilot**

# Write a Python function to generate Fibonacci series up to n terms  
  
# Include comments

**Function-Based Code**



**Sample Input**

Enter number of terms: 6

**Sample Output**

0 1 1 2 3 5

**Task 4: Comparative Analysis – Procedural vs Modular Code**

| Criteria        | Without Functions | With Functions |
|-----------------|-------------------|----------------|
| Code Clarity    | Lower             | Higher         |
| Reusability     | Poor              | Excellent      |
| Debugging       | Difficult         | Easier         |
| Scalability     | Not suitable      | Suitable       |
| Maintainability | Low               | High           |

**Conclusion:**

Function-based (modular) code is better suited for large systems due to improved readability, reusability, and maintainability.

---

## Task 5: Iterative vs Recursive Fibonacci Approaches

### Iterative Fibonacci (Copilot Generated)

```
def fibonacci_iterative(n):
```

```
    a, b = 0, 1
```

```
    for i in range(n):
```

```
        print(a, end=" ")
```

```
        a, b = b, a + b
```

#### Execution Flow

- Uses loop
- Updates values step-by-step
- Efficient memory usage

---

### Recursive Fibonacci (Copilot Generated)

```
def fibonacci_recursive(n):
```

```
    if n <= 1:
```

```
        return n
```

```
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

```
n = int(input("Enter number: "))
```

```
for i in range(n):
```

```
    print(fibonacci_recursive(i), end=" ")
```

#### Execution Flow

- Function calls itself
- Uses call stack
- Recomputes values repeatedly

---

### Comparison

| Aspect          | Iterative | Recursive |
|-----------------|-----------|-----------|
| Time Complexity | $O(n)$    | $O(2^n)$  |

| Aspect                | Iterative    | Recursive        |
|-----------------------|--------------|------------------|
| Space Complexity      | $O(1)$       | $O(n)$           |
| Performance (large n) | Fast         | Very slow        |
| Stack Overflow Risk   | No           | Yes              |
| Recommended Usage     | Large inputs | Educational only |

**Conclusion:**

Recursion should be avoided for large  $n$  due to high time complexity and memory overhead. Iterative solutions are more efficient and practical.

---

**Overall Conclusion**

This lab successfully demonstrated:

- Installation and usage of GitHub Copilot
- AI-assisted code generation
- Optimization through prompt engineering
- Modular vs procedural design
- Iterative vs recursive algorithmic approaches

GitHub Copilot significantly improves development speed, but human evaluation is essential to ensure correctness, efficiency, and maintainability.