ASSIGNMENT-01

Name: Praneeth Banda

HT.No: 2303A51711

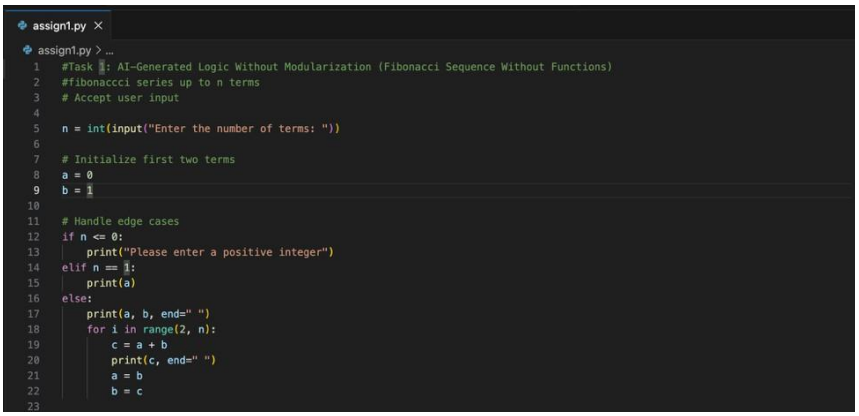
Batch: 20

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

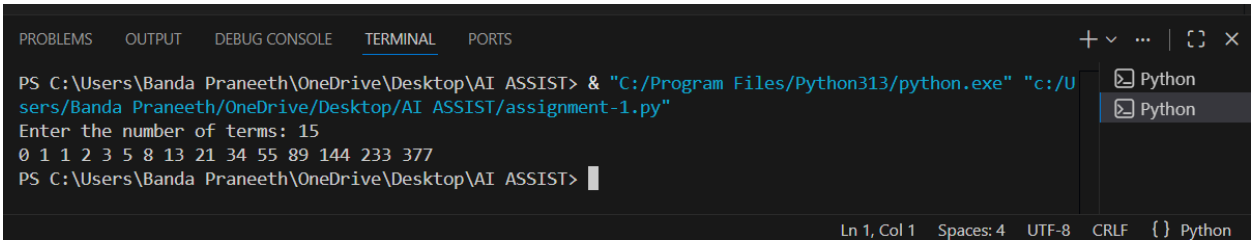
Scenario

You are asked to write a quick numerical sequence generator for a learning platform prototype.

Prompt: #Fibonacci series upto n terms



Result:



Observation: AI-Generated Logic Without Modularization

GitHub Copilot was able to produce an accurate Fibonacci sequence using simple, direct logic without implementing user-defined functions. The program was concise and appropriate for short, one-time tasks. However, the close coupling of computation and input/output reduced the code's clarity and reusability. This activity illustrated Copilot's strength in rapidly generating working solutions while also revealing its limitations in terms of scalability and code organization.

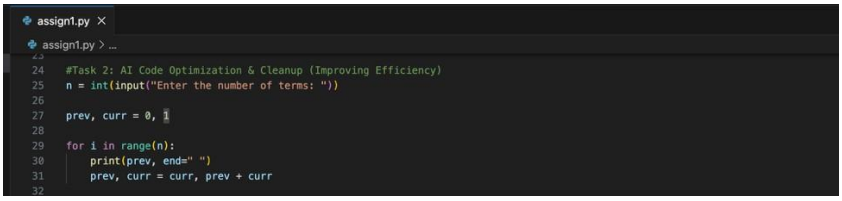
Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

Scenario

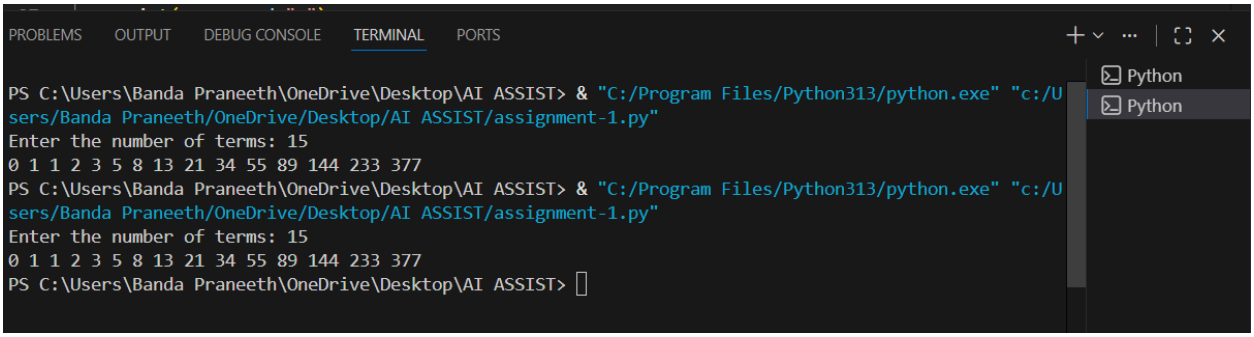
The prototype will be shared with other developers and needs optimization.

Prompt: Optimize this code

Code:



Result:



Observation: AI Code Optimization & Cleanup

When asked to optimize the program, Copilot effectively streamlined the original code by eliminating extra variables and avoiding unnecessary conditional statements. The resulting version was more concise, easier to read, and preserved the same time complexity. This exercise demonstrated Copilot's capability to enhance existing code when provided with clear instructions focused on optimization.

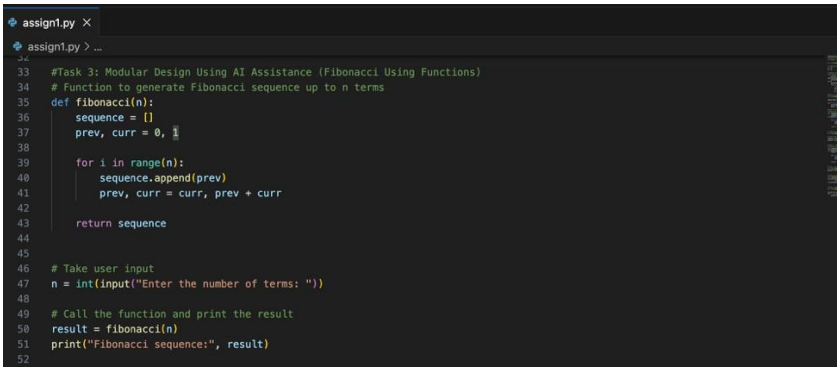
Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Scenario

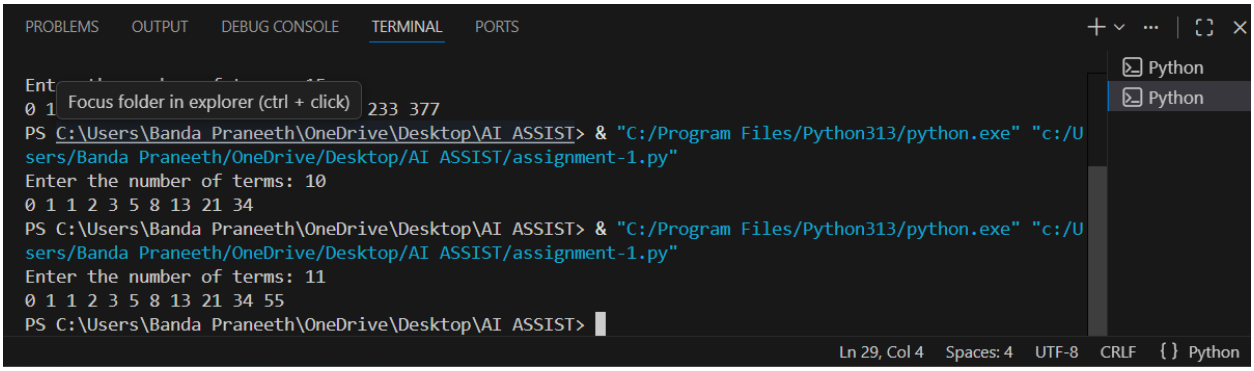
The Fibonacci logic is now required in multiple modules of an application.

Prompt: Optimize this code using functions

Code:



Result:



Observation: Function-Based Fibonacci Implementation

Copilot produced a structured Fibonacci solution built around a user-defined function, along with clear and relevant comments. Encapsulating the logic inside a function enhanced the readability of the code, made it reusable, and simplified testing. This exercise highlighted Copilot's understanding of modular programming concepts and its ability to generate code that can be efficiently reused across different parts of an application

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Scenario

You are participating in a code review session

Prompt: Description on comparision between with functions and without function

Observation: Code Review Comparison

The procedural version of the Fibonacci code is straightforward and suitable for small, one-time programs. It directly combines input handling, logic, and output in a single block, which makes it quick to write and easy to understand for beginners. However, this approach reduces readability as the program grows and makes the code difficult to reuse or test independently.
In contrast, the modular version organizes the Fibonacci logic inside a function. This separation improves clarity by isolating the core logic from input and output operations. It also enhances reusability, as the same function can be called from multiple parts of an application, and simplifies testing and maintenance. Overall, while the procedural approach works well for simple tasks, the modular approach is more scalable, maintainable, and better suited for larger applications

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

Scenario

Your mentor wants to assess AI's understanding of different algorithmic paradigms.

Prompt: give me a code using iterative and recursive approch

Code:

```python
#task-5
#iterative and recursive approach

def fibonacci_iterative(n):
    a, b = 0, 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result


def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)


n = int(input("Enter the number of terms: "))

print("Iterative approach:")
print(fibonacci_iterative(n))

print("Recursive approach:")
for i in range(n):
    print(fibonacci_recursive(i), end=" ")
```

Result:

Observation: Iterative vs Recursive Fibonacci Approaches

The above code demonstrates two algorithmic approaches to generate the Fibonacci sequence: iterative and recursive. The iterative approach uses a loop to compute each term efficiently with minimal memory usage and avoids repeated calculations. The recursive approach follows the mathematical definition of Fibonacci, making it simple and intuitive to understand, but it is less efficient due to repeated function calls. The iterative method is better suited for larger input sizes, while the recursive method is useful for learning and understanding recursion concepts. Selecting the appropriate approach depends on the input size and performance needs of the application.