ASSIGNMENT-01

Name: Vaishnavi Bairagoni

HT.No: 2303A51347

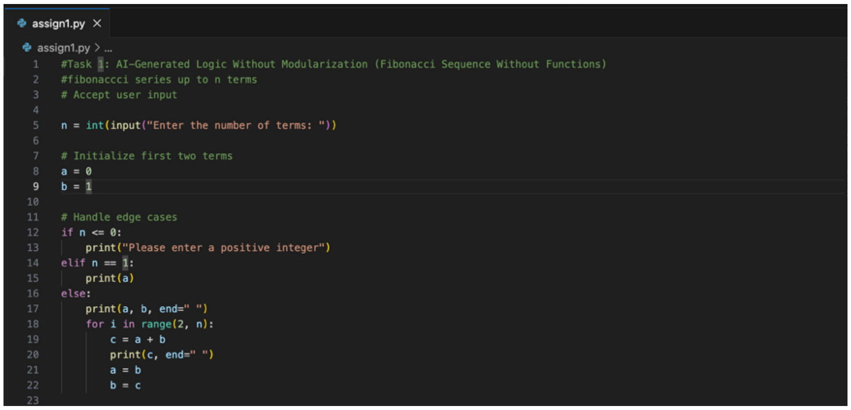
Batch: 20

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

Scenario

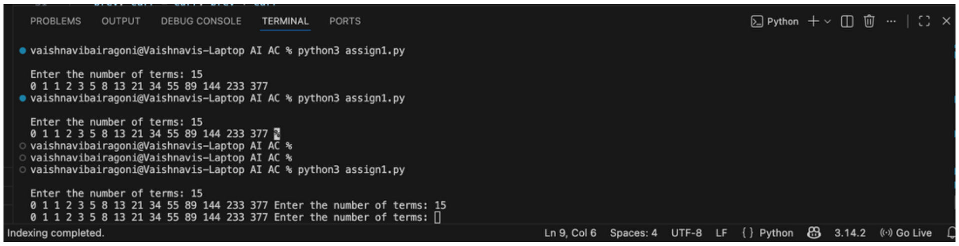
You are asked to write a quick numerical sequence generator for a learning platform prototype.

Prompt: #Fibonacci series upto n terms



```python
#Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)
#fibonaccci series up to n terms
# Accept user input

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

# Initialize first two terms
a = 0
b = 1

# Handle edge cases
if n <= 0:
    print("Please enter a positive integer")
elif n == 1:
    print(a)
else:
    print(a, b, end=" ")
    for i in range(2, n):
        c = a + b
        print(c, end=" ")
        a = b
        b = c

```

Result:



```
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 
```

Observation: AI-Generated Logic Without Modularization

GitHub Copilot was able to generate a correct Fibonacci sequence using direct logic without user-defined functions. The code was simple and suitable for small, one-time scripts. However, the logic and input/output were tightly coupled, which reduced readability and reusability. This task demonstrated Copilot’s ability to quickly produce working code but also highlighted limitations in scalability.

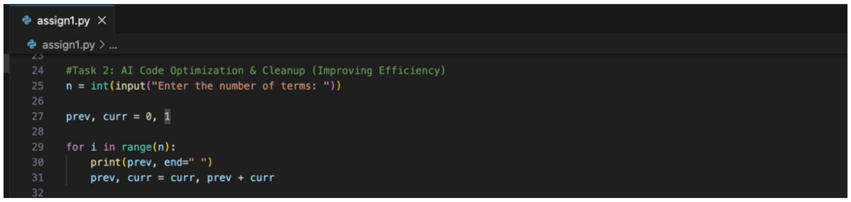
Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

Scenario

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

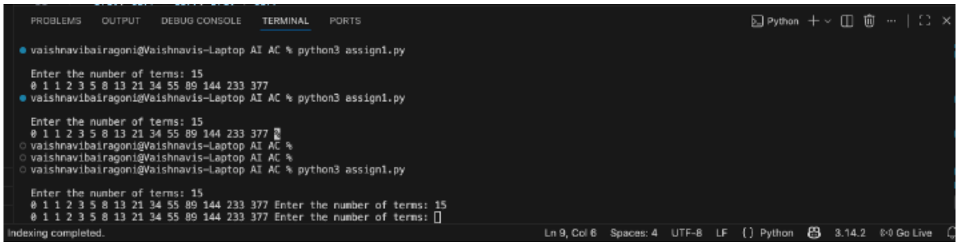
Prompt: Optimize this code

Code:



```python
#Task 2: AI Code Optimization & Cleanup (Improving Efficiency)
n = int(input("Enter the number of terms: "))

prev, curr = 0, 1

for i in range(n):
    print(prev, end=" ")
    prev, curr = curr, prev + curr
```

Result:



```
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 
```

Observation: AI Code Optimization & Cleanup

When prompted for optimization, Copilot successfully simplified the original code by removing redundant variables and unnecessary conditional checks. The optimized version was shorter, more readable, and maintained the same time complexity. This task showed that Copilot can effectively improve existing code when given clear optimization-related prompts

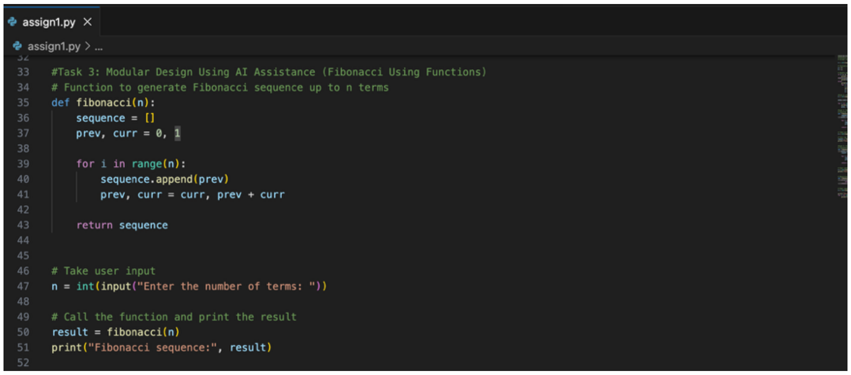
Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Scenario

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

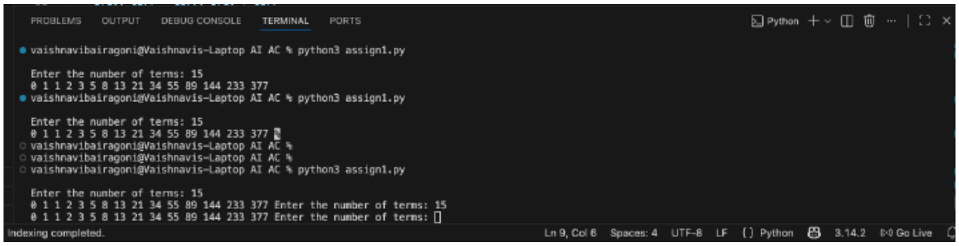
Prompt: Optimize this code using functions

Code:



```python
#Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)
# Function to generate Fibonacci sequence up to n terms
def fibonacci(n):
    sequence = []
    prev, curr = 0, 1

    for i in range(n):
        sequence.append(prev)
        prev, curr = curr, prev + curr

    return sequence


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

# Call the function and print the result
result = fibonacci(n)
print("Fibonacci sequence:", result)
```

Result:



```
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 
```

Observation: Function-Based Fibonacci Implementation

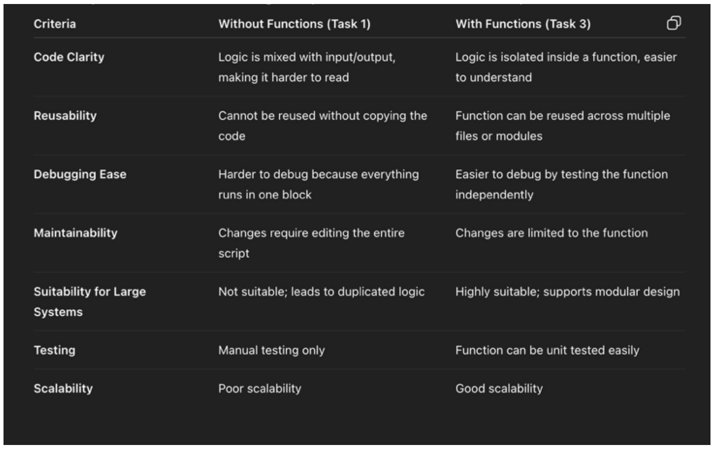
Copilot generated a modular, function-based Fibonacci implementation with meaningful comments. The use of a user-defined function improved code clarity, reusability, and ease of testing. This task demonstrated that Copilot understands modular programming principles and can produce code suitable for reuse across multiple modules

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 functions



Criteria	Without Functions (Task 1)	With Functions (Task 3)
Code Clarity	Logic is mixed with input/output, making it harder to read	Logic is isolated inside a function, easier to understand
Reusability	Cannot be reused without copying the code	Function can be reused across multiple files or modules
Debugging Ease	Harder to debug because everything runs in one block	Easier to debug by testing the function independently
Maintainability	Changes require editing the entire script	Changes are limited to the function
Suitability for Large Systems	Not suitable; leads to duplicated logic	Highly suitable; supports modular design
Testing	Manual testing only	Function can be unit tested easily
Scalability	Poor scalability	Good scalability

Observation: Code Review Comparison

The comparison between non-function and function-based implementations revealed that function-based code is more maintainable, easier to debug, and better suited for large systems. Copilot-supported modular code aligned more closely with software engineering best practices, emphasizing separation of concerns and reusability

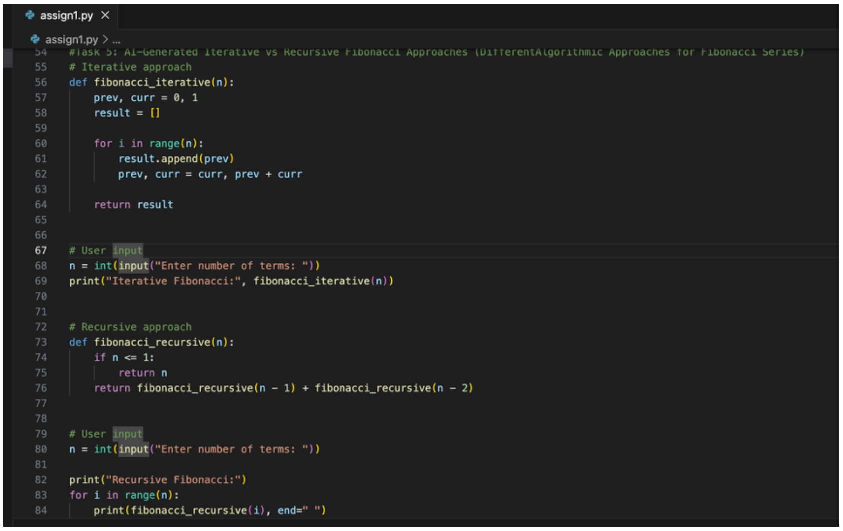
Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

Scenario

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

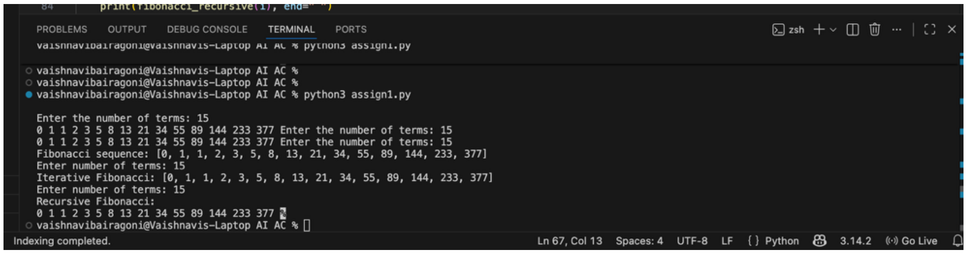
Prompt: give me a code using iterative and recursive approch

Code:



```python
#Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (DifferentAlgorithmic Approaches for Fibonacci Series)
# Iterative approach
def fibonacci_iterative(n):
    prev, curr = 0, 1
    result = []

    for i in range(n):
        result.append(prev)
        prev, curr = curr, prev + curr

    return result


# User input
n = int(input("Enter number of terms: "))
print("Iterative Fibonacci:", fibonacci_iterative(n))


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


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

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

Result:



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

```
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC %
vaishnavibairagoni@Vaishnavis-Laptop AI AC % python3 assign1.py

Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 Enter the number of terms: 15
Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
Enter number of terms: 15
Iterative Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
Enter number of terms: 15
Recursive Fibonacci:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 %
vaishnavibairagoni@Vaishnavis-Laptop AI AC % 
```

Observation: Iterative vs Recursive Fibonacci Approaches

Copilot successfully generated both iterative and recursive Fibonacci implementations. The iterative approach was efficient in terms of time and space complexity, while the recursive approach was simpler but computationally expensive for large inputs. This task demonstrated Copilot's understanding of different algorithmic paradigms and their trade-offs.