

# ASSIGNMENT-01

**Name:** Chitti Udaya Sathwika

**HT.No:** 2303A51322

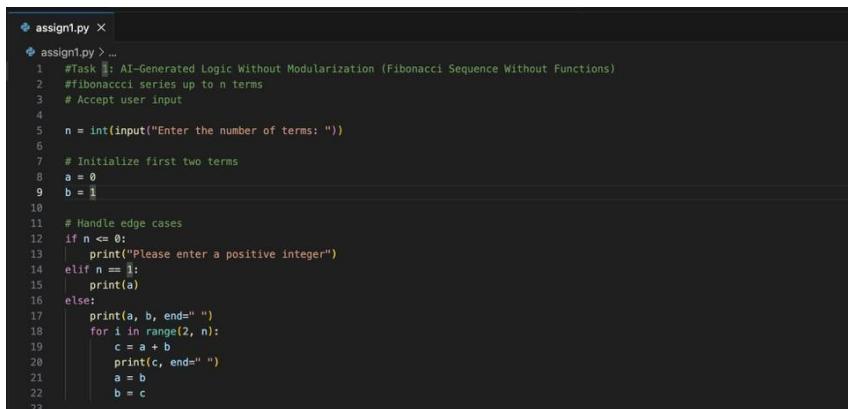
**Batch:** 19

## 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**



```
assign1.py < ...
assign1.py > ...
#Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)
#fibonacci 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
23
```

### Result:

```
Enter the number of terms: 35
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

## 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:

```
assign1.py < ...  
24 #Task 2: AI Code Optimization & Cleanup (Improving Efficiency)  
25 n = int(input("Enter the number of terms: "))  
26  
27 prev, curr = 0, 1  
28  
29 for i in range(n):  
30     print(prev, end=" ")  
31     prev, curr = curr, prev + curr  
32
```

### Result:

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

## 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:

```
assign1.py X
assign1.py > ...
33 # Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)
34 # Function to generate Fibonacci sequence up to n terms
35 def fibonacci(n):
36     sequence = []
37     prev, curr = 0, 1
38
39     for i in range(n):
40         sequence.append(prev)
41         prev, curr = curr, prev + curr
42
43     return sequence
44
45
46 # Take user input
47 n = int(input("Enter the number of terms: "))
48
49 # Call the function and print the result
50 result = fibonacci(n)
51 print("Fibonacci sequence:", result)
52
```

### Result:

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

### 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 approach

## Code:

```
(assign1.py) [1] assign1.py > ...
54 # Task 5: AI-generated iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)
55
56 # Iterative approach
57 def fibonacci_iterative(n):
58     prev, curr = 0, 1
59     result = []
60
61     for i in range(n):
62         result.append(prev)
63         prev, curr = curr, prev + curr
64
65     return result
66
67 # User input
68 n = int(input("Enter number of terms: "))
69 print("Iterative Fibonacci:", fibonacci_iterative(n))
70
71
72 # Recursive approach
73 def fibonacci_recursive(n):
74     if n <= 1:
75         return n
76     return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
77
78
79 # User input
80 n = int(input("Enter number of terms: "))
81
82 print("Recursive Fibonacci:")
83 for i in range(n):
84     print(fibonacci_recursive(i), end=" ")
```

### **Result:**

```
84     print(ribonacci_recursive(i), end= " ")
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
vaishnavibraigon@vaishnavis-Laptop:~/Desktop$ python3 assign1.py
vaishnavibraigon@vaishnavis-Laptop:~/Desktop$ python3 assign1.py
vaishnavibraigon@vaishnavis-Laptop:~/Desktop$ python3 assign1.py

Enter 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]
vaishnavibraigon@vaishnavis-Laptop:~/Desktop$
```

## 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.