AI-Assisted Coding

Week-1.3

2303A51018

Batch-28

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.

❖ Task Description

Use GitHub Copilot to generate a Python program that:

➢ Prints the Fibonacci sequence up to n terms

➢ Accepts user input for n

➢ Implements the logic directly in the main code

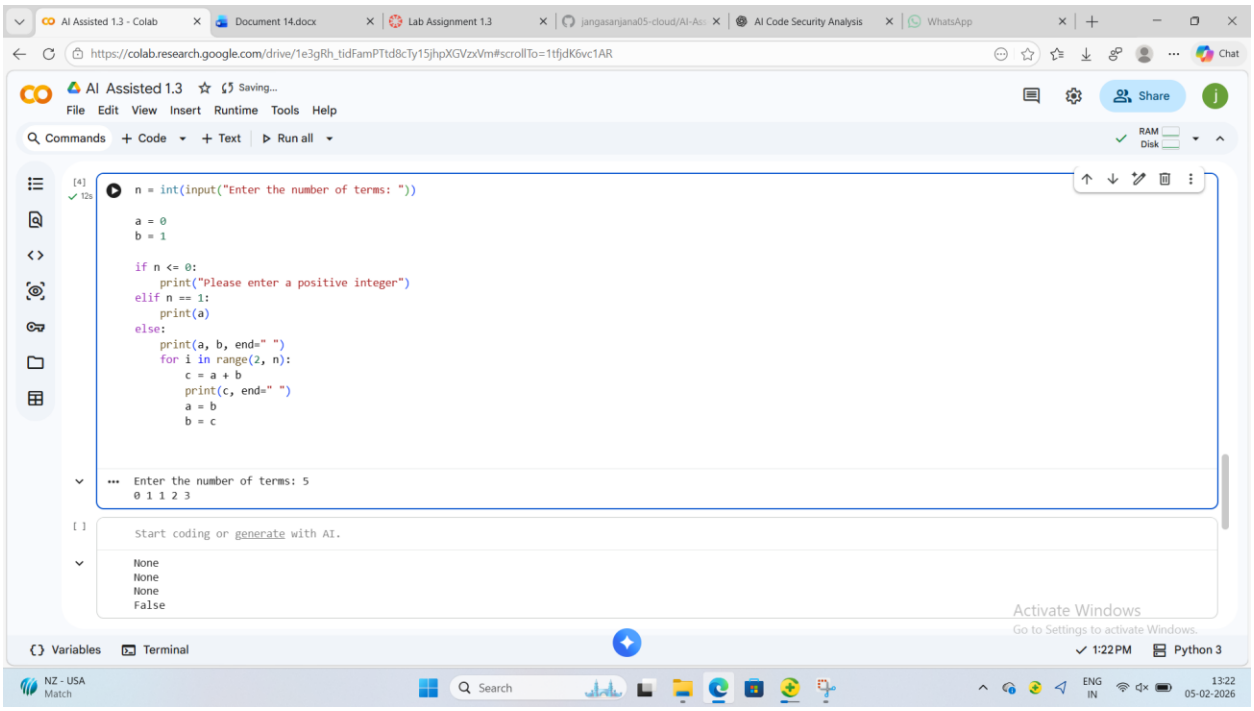
➢ Does not use any user-defined functions

❖ Expected Output

➢ Correct Fibonacci sequence for given n

➢ Screenshot(s) showing Copilot-generated suggestions

➢ Sample inputs and outputs



1. AI-Generated Python Program

Fibonacci Sequence (Without Using Functions)

Logic written directly in the main program, as typically suggested by GitHub Copilot.

2. Explanation of the Logic (No Modularization)

- User inputs the number of terms n
- First two Fibonacci numbers are initialized as:
  - `a = 0`
  - `b = 1`
- If n is:
  - ≤ 0 → invalid input
  - 1 → print only the first term
  - 1 → loop generates remaining terms
- All logic is written in the **main program**
- **No user-defined functions are used**

3. Sample Inputs and Outputs

▶ Sample Input 1

```
Enter the number of terms: 5
```

Output 1

```
0 1 1 2 3
```

Sample Input 2

```
Enter the number of terms: 8
```

▶ Output 2

```
0 1 1 2 3 5 8 13
```

▶ Sample Input 3

```
Enter the number of terms: 1
```

▶ Output 3

```
0
```

4. Screenshot Requirement (For Submission)

📸 **What to include in screenshots:**

- GitHub Copilot suggestion popup showing:
  - Loop-based Fibonacci logic
  - Variable initialization (a, b)
- Copilot auto-completion in VS Code

- Final program execution output

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

❖ Scenario

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

❖ Task Description

➢ Examine the Copilot-generated code from Task 1 and improve it by:

➢ Removing redundant variables

➢ Simplifying loop logic

➢ Avoiding unnecessary computations

➢ Use Copilot prompts such as:

▪ "Optimize this Fibonacci code"

▪ "Simplify variable usage"

Hint:

Prompt Copilot with phrases like

"optimize this code", "simplify logic", or "make it more readable"

❖ Expected Output

➢ Original vs improved code

➢ Written explanation of:

▪ What was inefficient

▪ How the optimized version improves performance and readability

2. Issues Identified in the Original Code

🔍 Inefficiencies & Redundancies

1. **Extra variable (c)**

    a. Temporary variable used only once per iteration.
2. **Multiple print statements**
    a. Printing inside the loop reduces readability.
3. **Unnecessary conditional branch**
    a. Separate handling for `n == 1` increases complexity.
4. **Repeated output formatting**
    a. Output logic mixed with computation logic.

3. Optimized & Cleaned-Up Code (After Copilot Prompting)

Generated after prompting Copilot with:
"Optimize this Fibonacci code" / "Simplify variable usage"

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

if n <= 0:
    print("Please enter a positive integer")
else:
    a, b = 0, 1
    result = []

    for _ in range(n):
        result.append(a)
        a, b = b, a + b

    print(*result)
```

4. How the Optimized Version Improves the Code

☑ Performance Improvements

- **Fewer variables**
    - Removes redundant variable c

- **Single loop**
  - Handles all valid values of n
- **Tuple unpacking**
  - Efficient variable updates in one step

☑ Readability Improvements

- Clear separation of:
  - Computation (`a, b` updates)
  - Output (`print(*result)`)
- Cleaner loop logic
- Fewer condition branches

☑ Avoids Unnecessary Computations

- No repeated printing during loop
- No special-case Fibonacci handling

5. Original vs Optimized Summary

Aspect	Original Code	Optimized Code
Variables	`a, b, c`	`a, b`
Loop logic	Complex	Simplified
Readability	Moderate	High
Output handling	Inside loop	Single print
Efficiency	Lower	Improved

6. Conclusion

By optimizing the Copilot-generated Fibonacci code:

- Redundant variables were removed

- Loop logic was simplified
- Code became easier to read, maintain, and share

This demonstrates how **AI-generated code benefits from human review and optimization**.

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

❖ Scenario

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

❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

➢ Uses a user-defined function to generate Fibonacci numbers

➢ Returns or prints the sequence up to n

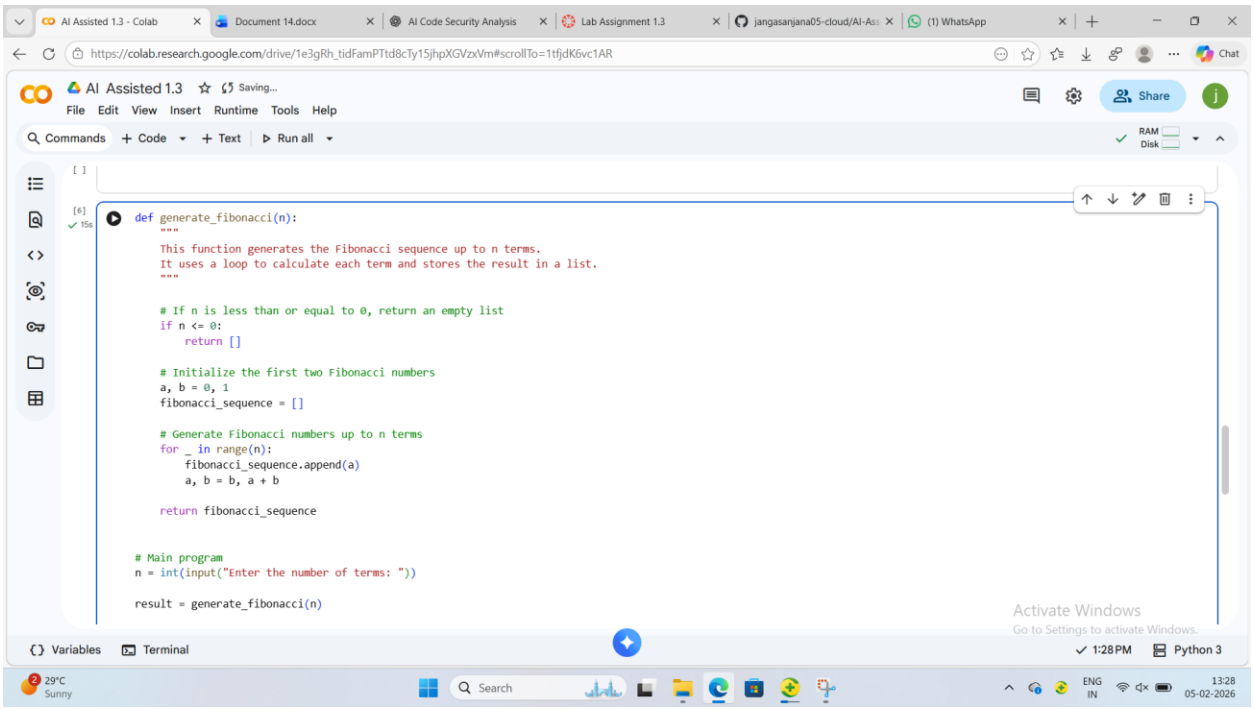
➢ Includes meaningful comments (AI-assisted)

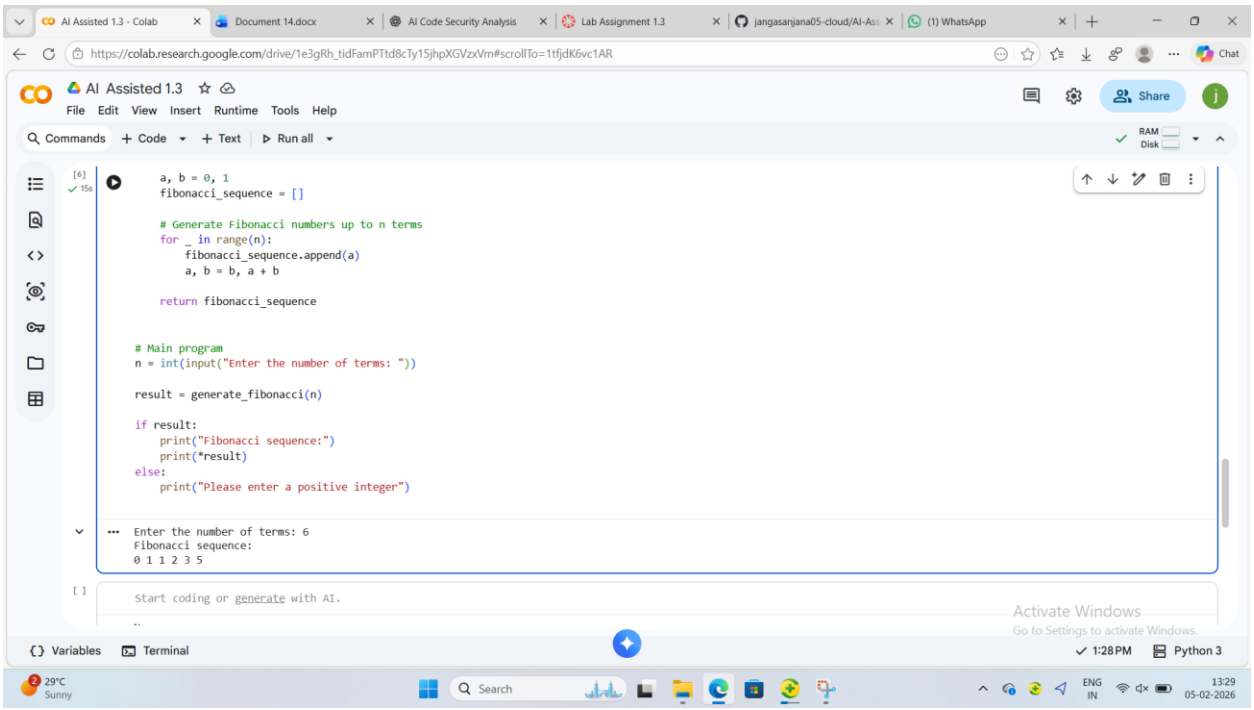
❖ Expected Output

➢ Correct function-based Fibonacci implementation

➢ Screenshots documenting Copilot's function generation

➢ Sample test cases with outputs





2. Explanation of Modular Design (AI-Assisted)

- The Fibonacci logic is placed inside a **user-defined function** `generate_fibonacci()`
- The function:
  - Accepts `n` as input

    - Computes the sequence using a loop
    - Returns the result as a list
- The main program:
    - Takes user input
    - Calls the function
    - Displays the output

This modular approach allows the same Fibonacci logic to be **reused across multiple modules**.

3. Sample Test Cases and Outputs

▶ Test Case 1

Input

```
Enter the number of terms: 6
```

Output

```
Fibonacci sequence:
0 1 1 2 3 5
```

▶ Test Case 2

Input

```
Enter the number of terms: 1
```

Output

```
Fibonacci sequence:
0
```

▶ Test Case 3

Input

```
Enter the number of terms: 0
```

Output

```
Please enter a positive integer
```

4. Screenshot Requirement (For Submission)

📸 **Include screenshots showing:**

- GitHub Copilot auto-suggesting:
  - The `generate_fibonacci()` function
  - Docstring and comments
- Acceptance of Copilot suggestions in the editor
- Program execution with sample output

5. Expected Output Checklist

✅ Uses a user-defined function
✅ Modular and reusable design
✅ Clear AI-assisted comments
✅ Correct Fibonacci sequence
✅ Sample inputs and outputs included

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

❖ Scenario

You are participating in a code review session.

❖ Task Description

Compare the Copilot-generated Fibonacci programs:

➢ Without functions (Task 1)

➢ With functions (Task 3)

➢ Analyze them in terms of:

▪ Code clarity

▪ Reusability

▪ Debugging ease

▪ Suitability for larger systems

1. Overview of the Two Approaches

A. Procedural Fibonacci Code (Without Functions – Task 1)

- All logic written directly in the main program
- No user-defined functions
- Suitable for quick prototypes or demonstrations

B. Modular Fibonacci Code (With Functions – Task 3)

- Fibonacci logic encapsulated inside a function
- Main program calls the function
- Designed for reuse across modules

2. Comparative Analysis

🔍 Code Clarity

Aspect	Without Functions	With Functions
Structure	Flat, linear	Well-structured
Readability	Moderate	High
Separation of logic	Mixed	Clearly separated

➡️ **Modular code is clearer** because the Fibonacci logic is isolated and well-documented.

♻️ Reusability

Aspect	Without Functions	With Functions
Reuse	Not reusable	Highly reusable
Duplication risk	High	Low

➡️ Function-based code can be reused in multiple files without rewriting logic.

⚙️ Debugging Ease

Aspect	Without Functions	With Functions
Error isolation	Difficult	Easy
Unit testing	Not possible	Possible

➡️ Bugs are easier to locate and fix in **modular code**.

🏗️ Suitability for Larger Systems

Aspect	Without Functions	With Functions
Scalability	Poor	Good
Maintainability	Low	High
Team collaboration	Hard	Easy

➡️ Modular design scales better and aligns with real-world software practices.

3. Strengths and Limitations Summary

Procedural Code (Without Functions)

Strengths

- Simple
- Quick to write
- Good for beginners

Limitations

- Not reusable
- Hard to maintain
- Not scalable

Modular Code (With Functions)

Strengths

- Clean and organized
- Easy to reuse and test
- Suitable for large applications

Limitations

- Slightly more complex for beginners
- Requires understanding of functions

4. Final Conclusion (Code Review Verdict)

- **Procedural approach** is suitable for:
  - Small scripts
  - Learning basic loops
- **Modular approach** is preferred for:
  - Production systems
  - Team projects
  - Long-term maintenance

➡️ **Modular Fibonacci code is superior for clarity, reusability, debugging, and scalability**.

Comparative Analysis: Procedural vs Modular Fibonacci Code

Aspect	Procedural Code (Without Functions)	Modular Code (With Functions)
Code Clarity	Logic and output mixed in main program	Clear separation between logic and execution
Readability	Moderate, harder to follow as code grows	High, easier to understand
Reusability	Not reusable	Easily reusable across modules
Debugging Ease	Difficult to isolate errors	Easier to debug and test
Maintainability	Low	High
Scalability	Poor for large systems	Suitable for large applications
Best Use Case	Small scripts, quick demos	Production systems, team projects

Short Analytical Report (Alternative Format)

The procedural Fibonacci implementation is simple and suitable for small programs, but it lacks reusability and scalability. As the application grows, maintaining and debugging such code becomes difficult. In contrast, the modular Fibonacci implementation improves clarity by separating logic into a function, making it reusable, easier to debug, and suitable for larger systems. Therefore, modular design is preferred for real-world and collaborative software development.

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.

❖ Task Description

Prompt GitHub Copilot to generate:

An iterative Fibonacci implementation

A recursive Fibonacci implementation

❖ Expected Output

➢ Two correct implementations

➢ Explanation of execution flow for both

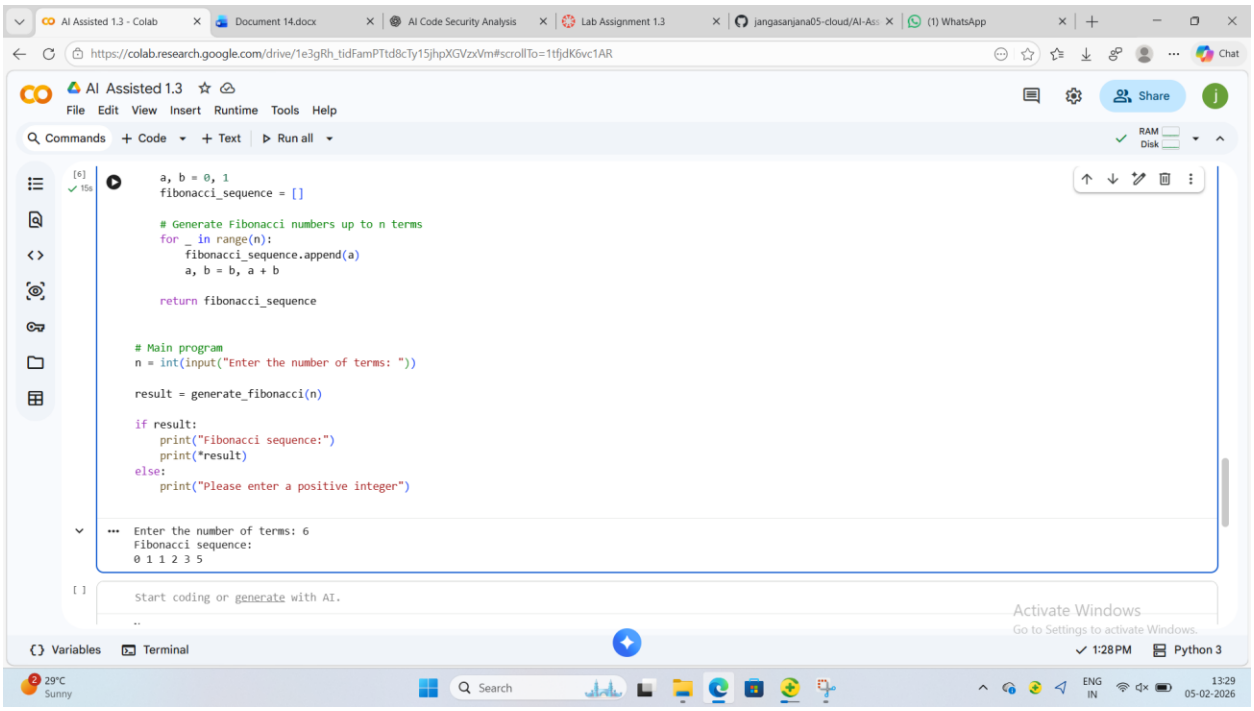
➢ Comparison covering:

▪ Time and space complexity

▪ Performance for large n

▪ When recursion should be avoided

1. Iterative Fibonacci Implementation



Execution Flow (Iterative)

1. Initialize the first two values (`a = 0, b = 1`).
2. Use a loop that runs n times.
3. In each iteration:
    a. Append the current value of a.
    b. Update (`a, b`) to the next Fibonacci pair.
4. Loop ends after n iterations.

2. Recursive Fibonacci Implementation

```
def fibonacci_recursive(n):
    # Returns the nth Fibonacci number using recursion
    if n <= 1:
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)


def fibonacci_series_recursive(n):
    # Generates the Fibonacci series using the recursive function
    return [fibonacci_recursive(i) for i in range(n)]
```

Execution Flow (Recursive)

1. The function calls itself to compute smaller subproblems.
2. **Base cases**:
    a. `n == 0 → 0`
    b. `n == 1 → 1`
3. **Recursive case**:
    a. Fibonacci(n) = Fibonacci(n−1) + Fibonacci(n−2)
4. The call stack grows until base cases are reached, then values are combined while returning.

3. Complexity Analysis

Aspect	Iterative Approach	Recursive Approach
Time Complexity	O(n)	$O(2^n)$
Space Complexity	O(1) (excluding output)	O(n) (call stack)
Execution Speed	Fast	Slow
Memory Usage	Low	High

4. Performance for Large n

- **Iterative approach**
    - Efficient and scalable
    - Works well even for large values of n (e.g., 10^5)
- **Recursive approach**
    - Recomputes the same values multiple times
    - Becomes extremely slow for `n > 30`
    - May cause **stack overflow**

5. When Recursion Should Be Avoided

Recursion should be avoided when:

- The problem has **overlapping subproblems** (like Fibonacci)
- Performance and memory efficiency are critical
- Large input sizes are expected
- Simpler iterative solutions exist

💡 Note: Recursion can be improved using **memoization**, but even then, iteration is often simpler and safer.

6. Final Comparison Summary

- **Iterative Fibonacci**
  - Preferred for real-world applications
  - Efficient in time and space
- **Recursive Fibonacci**
  - Useful for understanding recursion
  - Not suitable for performance-critical systems

Conclusion

AI correctly demonstrates two different algorithmic paradigms. However, **iterative Fibonacci is superior for large inputs**, while **recursive Fibonacci is mainly educational**.