[AI Assisted Coding]

ANNAPU VASANTHA SHOBA RANI

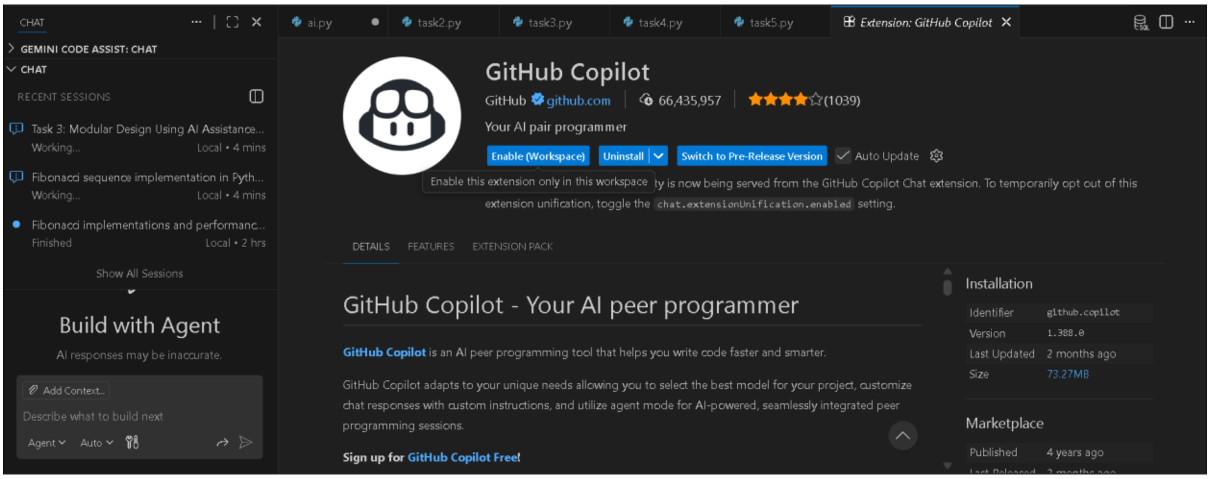
2303A51395

BATCH - 06

Lab 1: Environment Setup – GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow

Lab Objectives:

Week1 -



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

PROMPT:

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

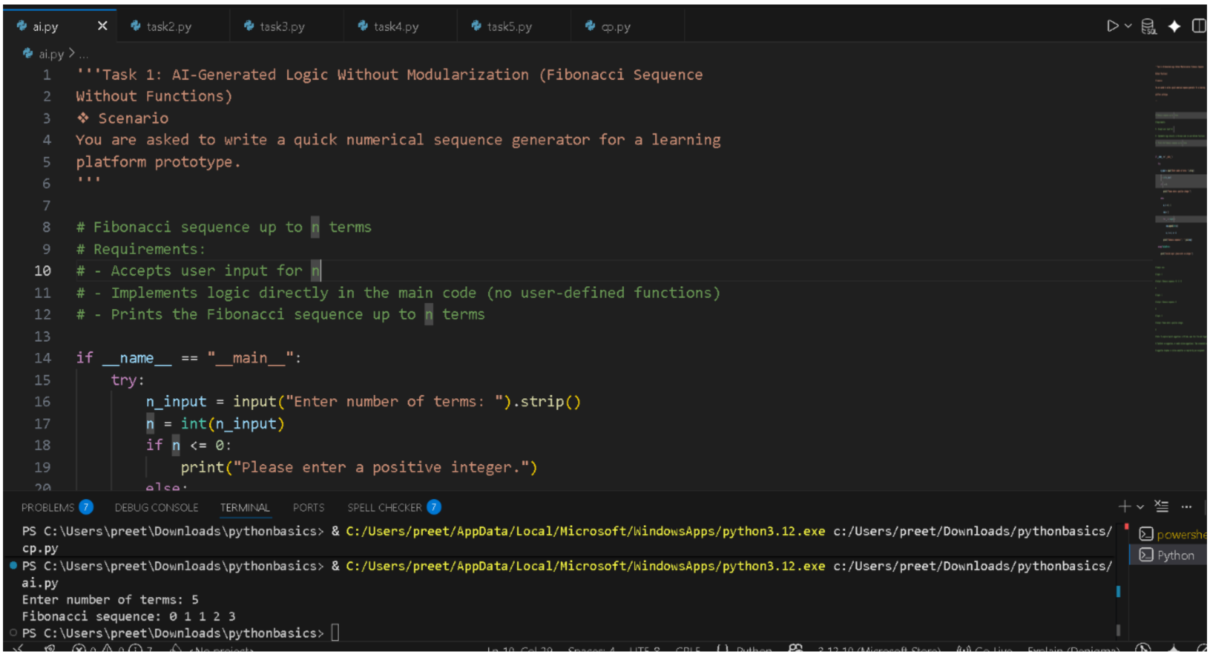
➢ Does not use any user-defined functions

❖ Expected Output

➢ Correct Fibonacci sequence for given n

➢ Screenshot(s) showing Copilot-generated suggestions

➢ Sample inputs and outputs



OUTPUT: Enter number of terms: 5

Fibonacci sequence: 0 1 1 2 3

Code Explanation (Fibonacci without functions)

- input() → takes number of terms from user
- int() → converts input to integer
- if n <= 0 → checks for invalid (non-positive) input

- a, b = 0, 1 → starting Fibonacci numbers
- for _ in range(n) → runs loop n times
- seq.append(str(a)) → stores current Fibonacci number
- a, b = b, a + b → updates values for next term
- " ".join(seq) → prints numbers in one line
- try / except → handles invalid (non-integer) input safely

Output: Prints Fibonacci sequence up to n terms.

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

PROMPT: 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”

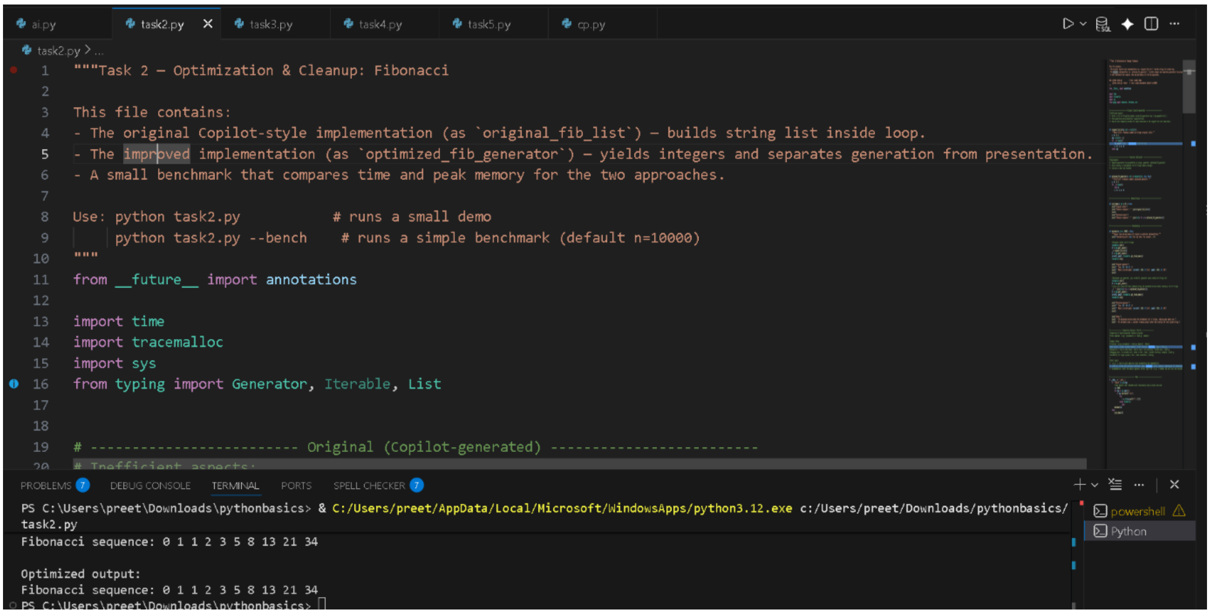
❖ Expected Output

➢ Original vs improved code

➢ Written explanation of:

▪ What was inefficient

▪ How the optimized version improves performance and readability



OUTPUT:

Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34

Optimized output:

Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34

CODE EXPLANATION

Optimization & Cleanup (Fibonacci)

Original version (original_fib_list)

- Generates Fibonacci numbers
- Converts each number to string inside the loop
- Stores all values in a list
  → **More memory usage**, mixed logic (generation + formatting)

Optimized version (optimized_fib_generator)

- Uses a **generator (yield)**
- Produces numbers one-by-one
- Conversion to string happens only when printing
  → **Less memory**, cleaner design, reusable

Demo

- Prints Fibonacci using both methods
- Output is the same

Benchmark

- Measures **time** and **memory**
- Optimized version:
    - Uses less peak memory
    - Scales better for large n

Key takeaway

- Generator = better performance + cleaner code
- Separate **logic (generation)** from **presentation (printing)**

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

PROMPT:

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

Code Explanation:

Optimization & Cleanup (Fibonacci)

Original version (original_fib_list)

- Generates Fibonacci numbers
- Converts each number to string inside the loop
- Stores all values in a list
  → **More memory usage**, mixed logic (generation + formatting)

Optimized version (optimized_fib_generator)

- Uses a **generator (yield)**
- Produces numbers one-by-one
- Conversion to string happens only when printing
  → **Less memory**, cleaner design, reusable

Demo

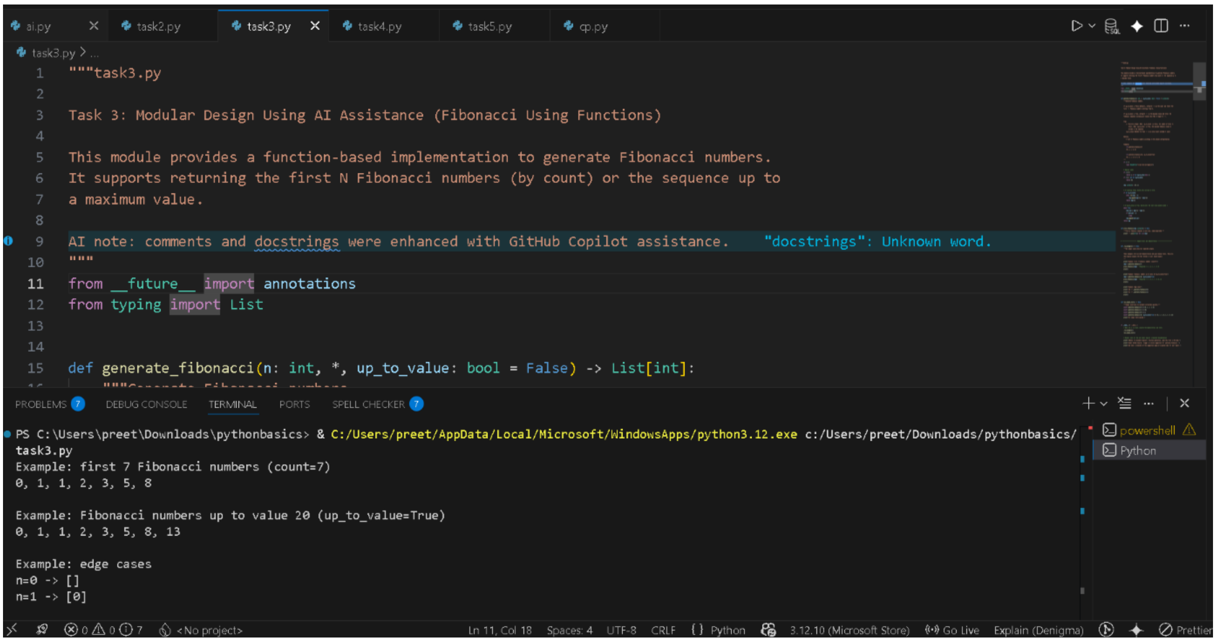
- Prints Fibonacci using both methods
- Output is the same

Benchmark

- Measures **time** and **memory**
- Optimized version:
  - Uses less peak memory
  - Scales better for large n

Key takeaway

- Generator = better performance + cleaner code
- Separate **logic (generation)** from **presentation (printing)**



OUTPUT:

Example: first 7 Fibonacci numbers (count=7)

0, 1, 1, 2, 3, 5, 8

Example: Fibonacci numbers up to value 20 (up_to_value=True)

0, 1, 1, 2, 3, 5, 8, 13

Example: edge cases

n=0 -> []

n=1 -> [0]

All simple tests passed.

Notes: To document Copilot's function generation, open this file in VS Code,

enable GitHub Copilot, trigger an inline suggestion for `generate_fibonacci`,

and take a screenshot of the suggestion popup or accepted code for your report.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

PROMPT:

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

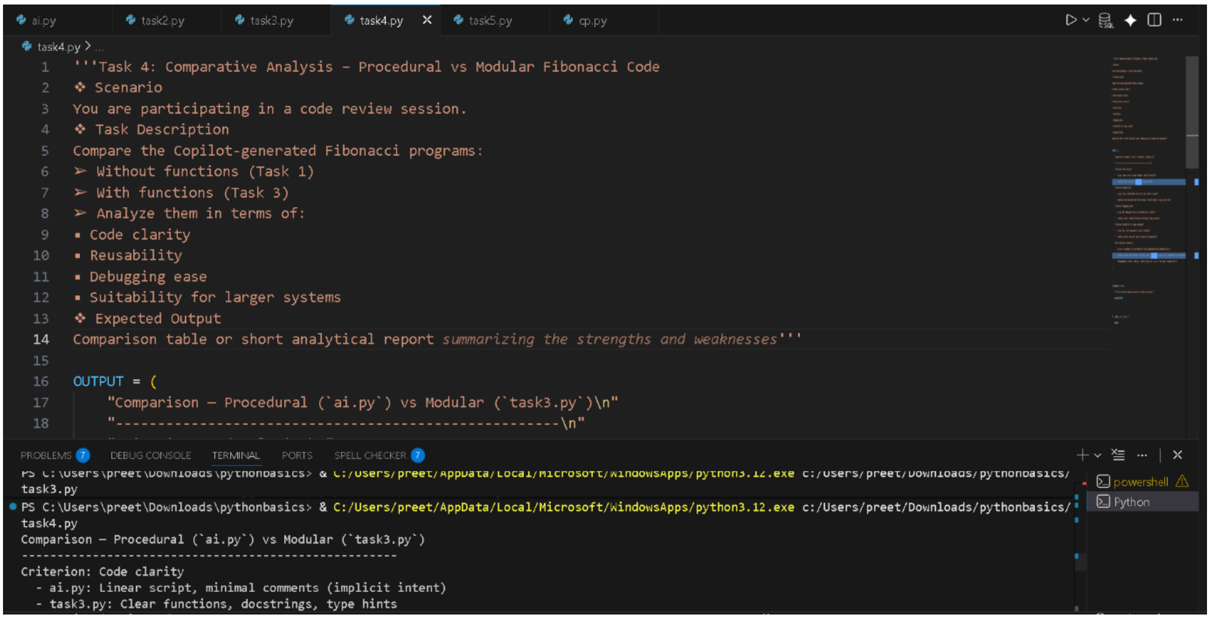
▪ Reusability

▪ Debugging ease

▪ Suitability for larger systems

❖ Expected Output

Comparison table or short analytical report



OUTPUT:

Comparison — Procedural (`ai.py`) vs Modular (`task3.py`)

Criterion: Code clarity

- ai.py: Linear script, minimal comments (implicit intent)

- task3.py: Clear functions, docstrings, type hints

Criterion: Reusability

- ai.py: Tied to stdin/stdout and script flow — hard to reuse
- task3.py: Pure functions that return values, flexible options (`up_to_value`)

Criterion: Debugging ease

- ai.py: Must debug main flow; no isolated units or tests
- task3.py: Small, testable functions with explicit edge cases

Criterion: Suitability for larger systems

- ai.py: Poor — not composable or easily testable
- task3.py: Good — clear API, easy to extend and integrate

Short analytical summary:

- ai.py is acceptable for quick demos but lacks documentation and composability.
- task3.py follows best practices (functions, types, docstrings, tests) and is preferable for maintainability and production.
- Recommendation: prefer `task3.py`. Extract logic from `ai.py` if you need a reusable API.

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

PROMPT :

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

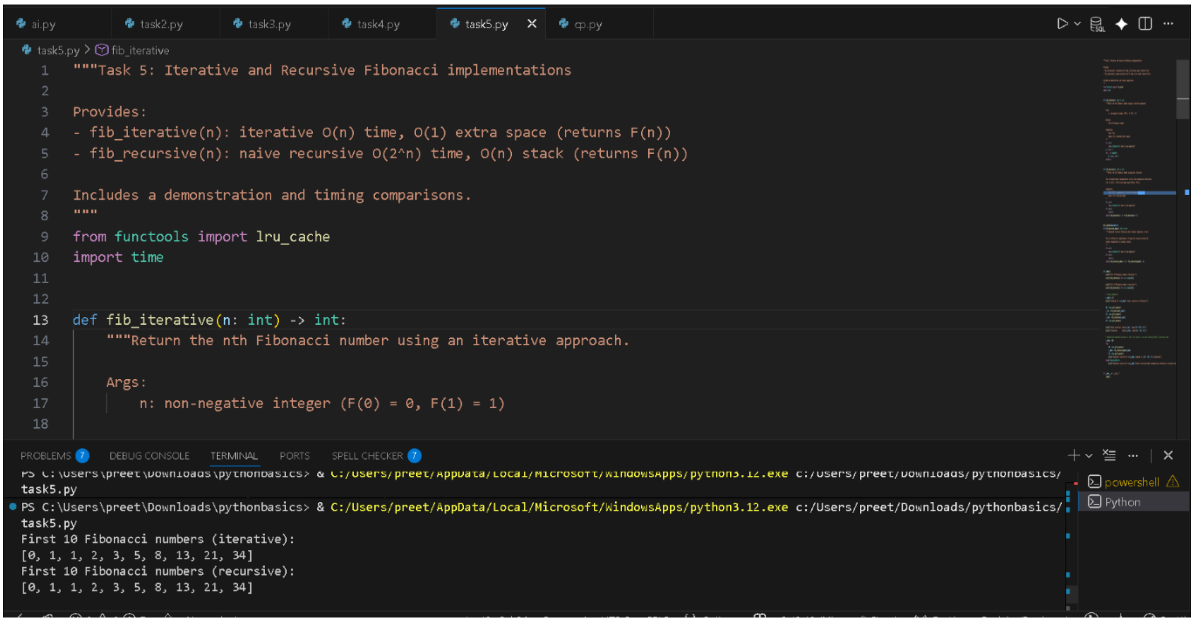
➢ Explanation of execution flow for both

➢ Comparison covering:

▪ Time and space complexity

▪ Performance for large n

▪ When recursion should be avoided



OUTPUT:

First 10 Fibonacci numbers (iterative):

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

First 10 Fibonacci numbers (recursive):

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Timing for n=30 (naive recursive vs iterative):

naive recursive: value=832040, time=0.185535s

iterative: value=832040, time=0.000009s

memoized recursive for n=500 computed in 0.001015s (memoized)

CODE EXPLANATION

Task 5: Iterative vs Recursive Fibonacci

fib_iterative(n)

- **Uses a loop to calculate Fibonacci**
- **Updates values using a, b = b, a + b**
- **Time: O(n)**
- **Space: O(1)**
  ✔ Fast and memory-efficient

fib_recursive(n)

- **Follows the formula F(n) = F(n-1) + F(n-2)**
- **Repeats many calculations**
- **Time: $O(2^n)$**
- **Space: O(n) (call stack)**
  ✖ Very slow for large n

fib_recursive_memo(n)

- **Uses @lru_cache to store previous results**
- **Avoids repeated work**
- **Time: O(n)**
- **Space: O(n)**
  ✔ Fast but still uses recursion

Demo & Timing

- **Prints first 10 Fibonacci numbers**
- **Compares execution time of iterative vs recursive**
- **Shows memoized recursion is much faster than naive recursion**

Conclusion

- **Iterative → best for performance**

- **Naive recursive → only for learning**
- **Memoized recursive → good balance, but iterative is safest for large n**