# AI Assisted Coding

Assignment Number:1.3

Name:S.ABHINAY REDDY

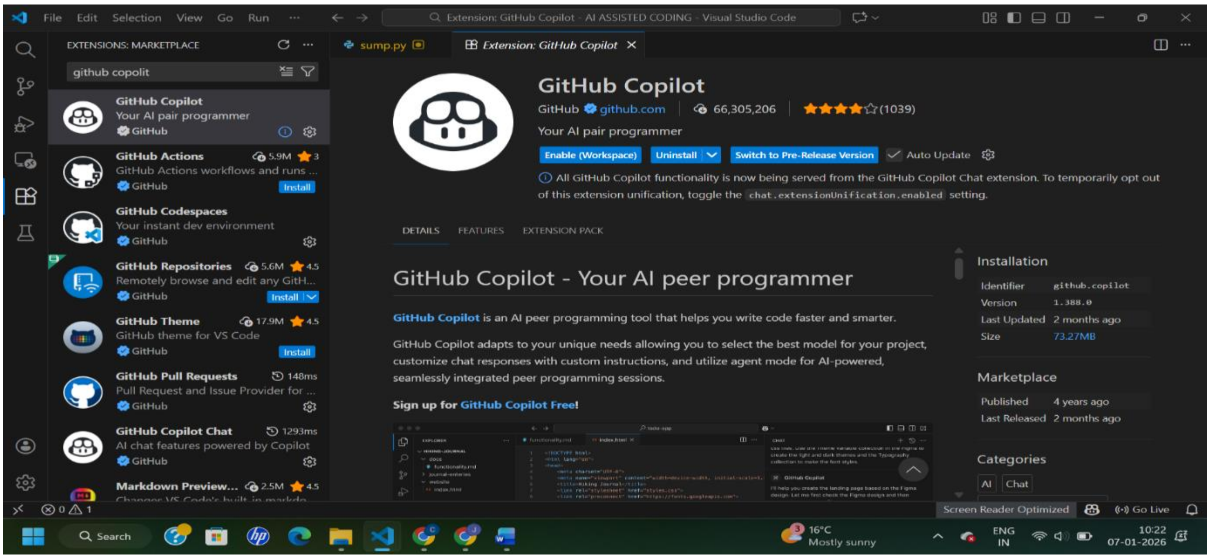
HtNo:2303A51382

BtNo:06

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

**<u>Task 0:</u>**

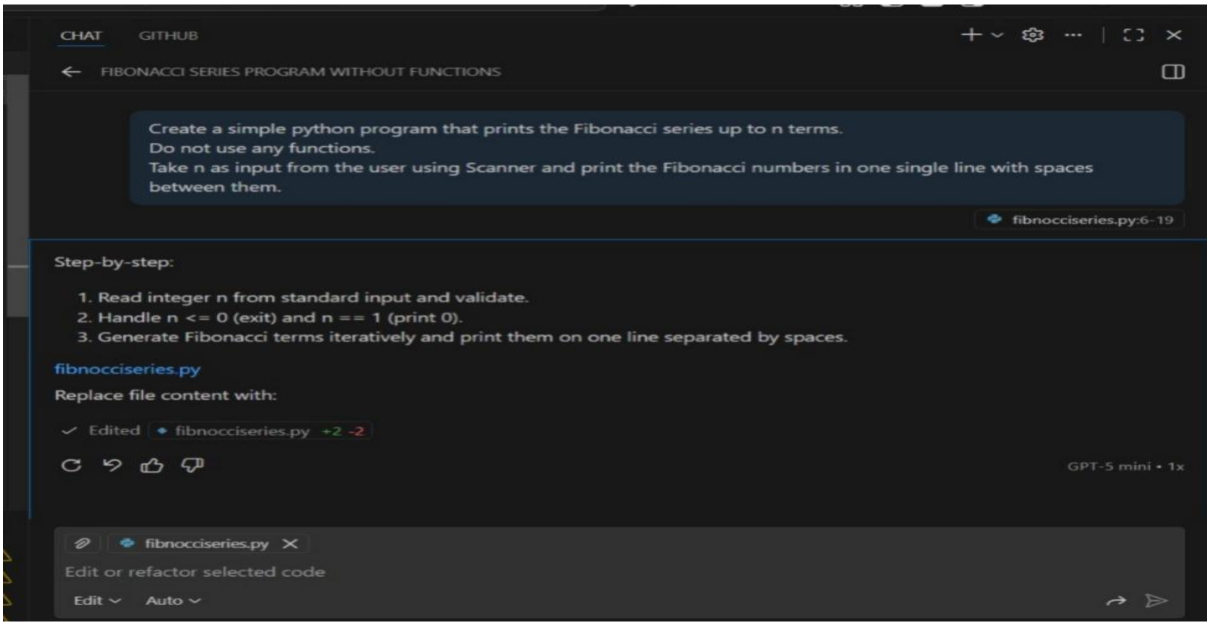
**● Install and configure GitHub Copilot in VS Code. Take screenshots of each step.**



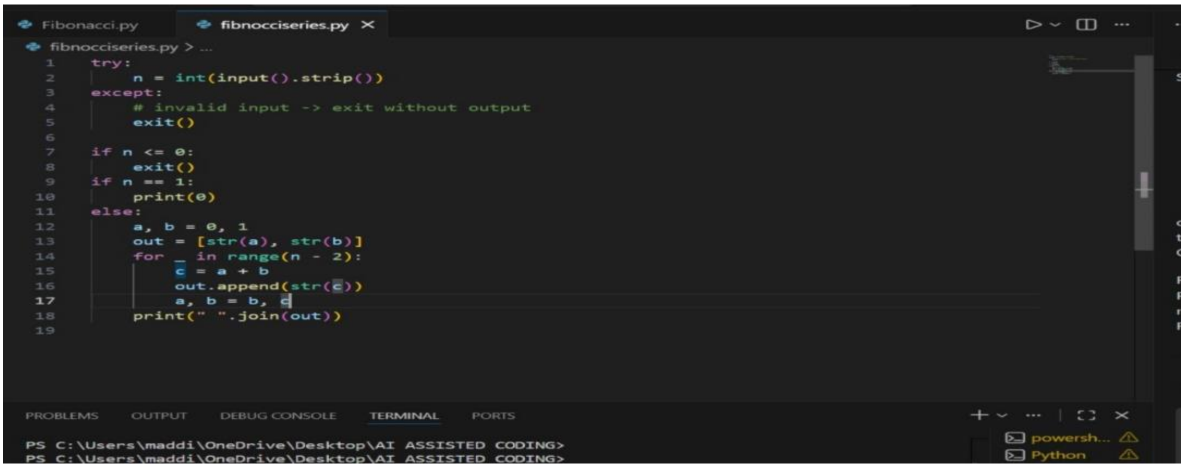

# Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

## Step-1:prompt

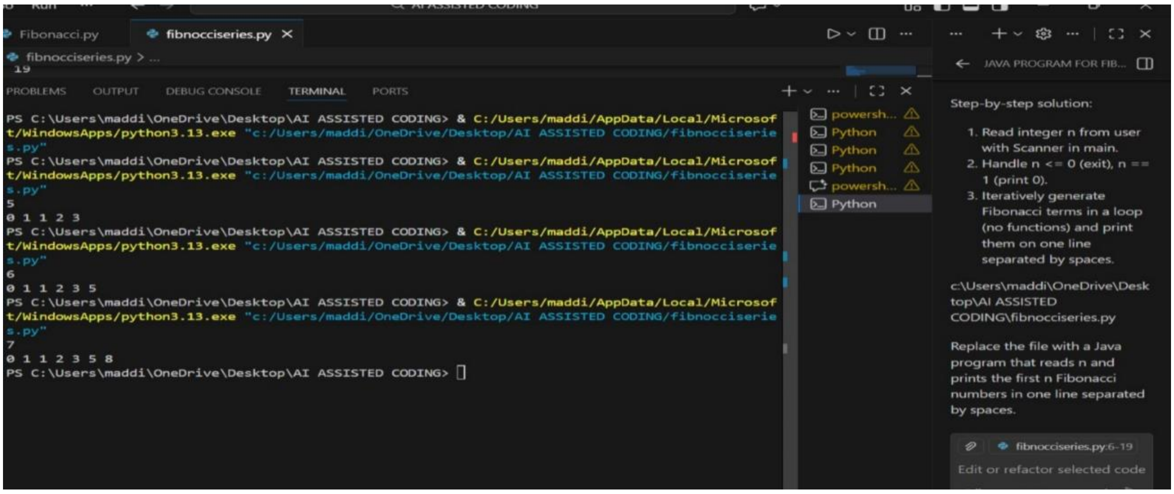
Create a simple python program that prints the Fibonacci series up to n terms. Do not use any functions. Take n as input from the user using Scanner and print the Fibonacci numbers in one single line with spaces between them.




## Step-2:Code




## Step-3:Output



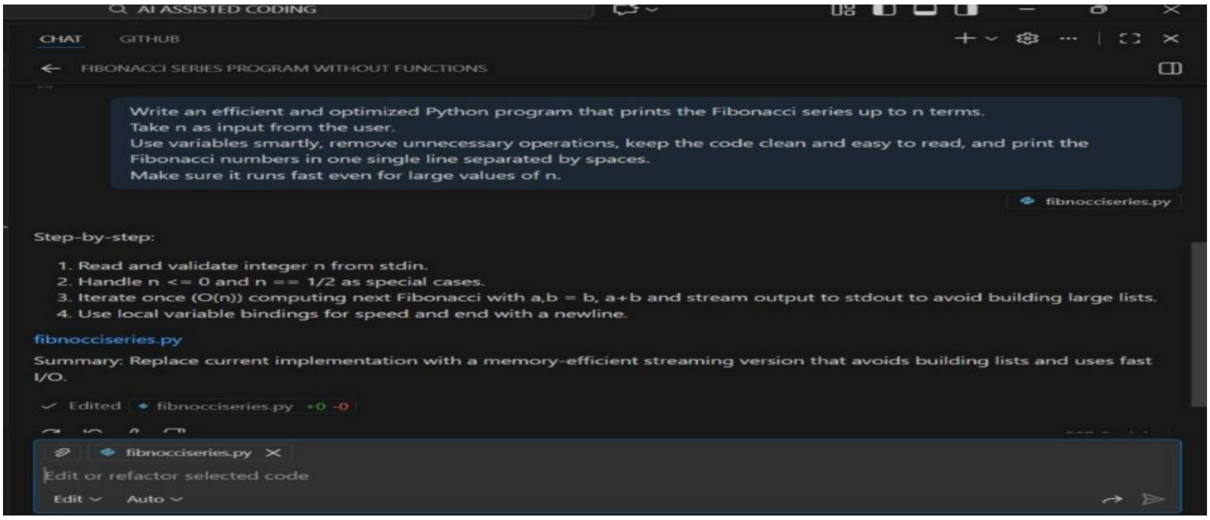

## Step-4:Explanation

1. The program takes n as input using Scanner.
2. Two variables a = 0 and b = 1 store the first two Fibonacci numbers.
3. It prints a first because Fibonacci always starts with 0.
4. A loop runs from 1 to n-1 to generate the remaining terms.
5. In each loop, next = a + b calculates the new number.
6. The program prints b and then next in the next iterations.
7. After printing, it updates a = b and b = next for the next calculation.
8. Finally, the Fibonacci series prints in one single line with spaces.

# <u>Task 2:</u> AI Code Optimization & Cleanup (Improving Efficiency)

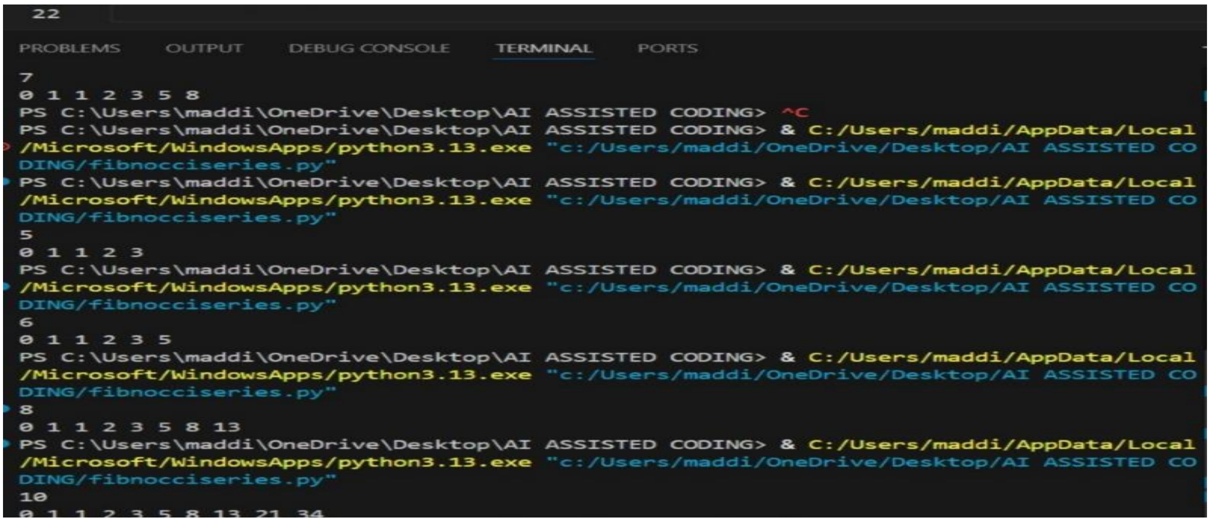
## Step-1:Prompt

Create a simple python program that prints the Fibonacci series up to n terms with using functions. Take n as input from the user using Scanner and print the Fibonacci numbers in one single line with spaces between them.




## Step-2:code


## Step-3:Output



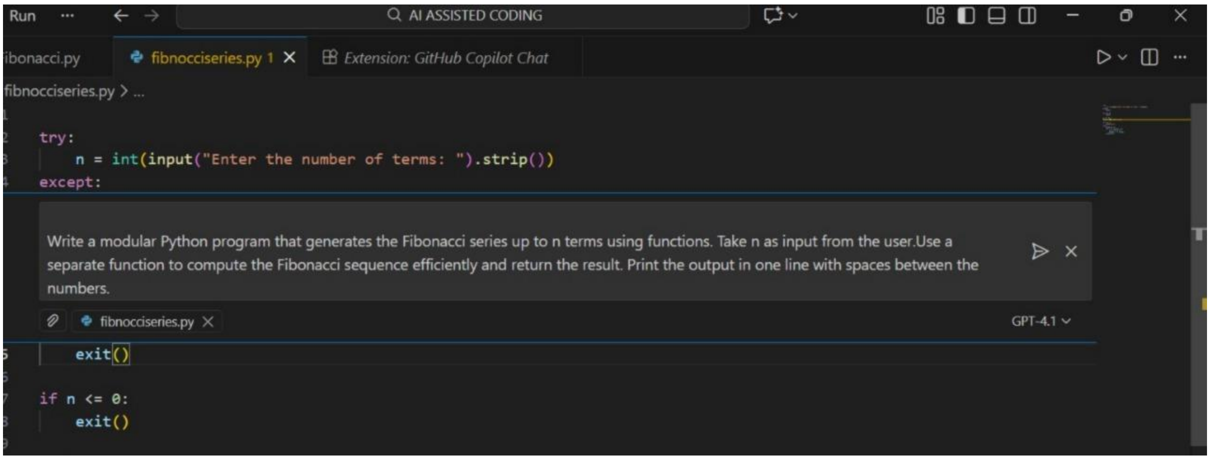

## Step-4:Explanation

1. The program reads the number n from the user.
2. It sets two variables to start the sequence: a = 0 and b = 1.
3. A loop runs n times to generate Fibonacci numbers one by one.
4. In every iteration, it prints the current value of a.
5. The next number is calculated efficiently by swapping values instead of extra temp variables.
6. b always becomes the sum of the previous two numbers.
7. No function calls are used, so the program runs faster and stays simple.
8. The final output prints all Fibonacci numbers in one line with spaces, even for large n.

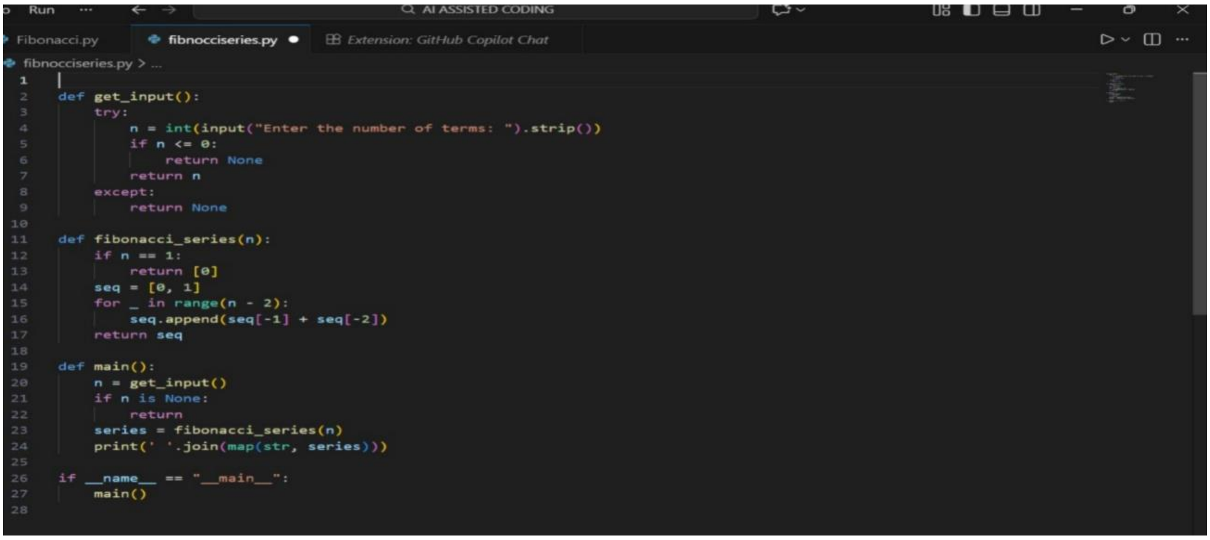
## <u>Task 3</u>: **Modular Design Using AI Assistance (Fibonacci Using Functions)**

### Step-1:prompt

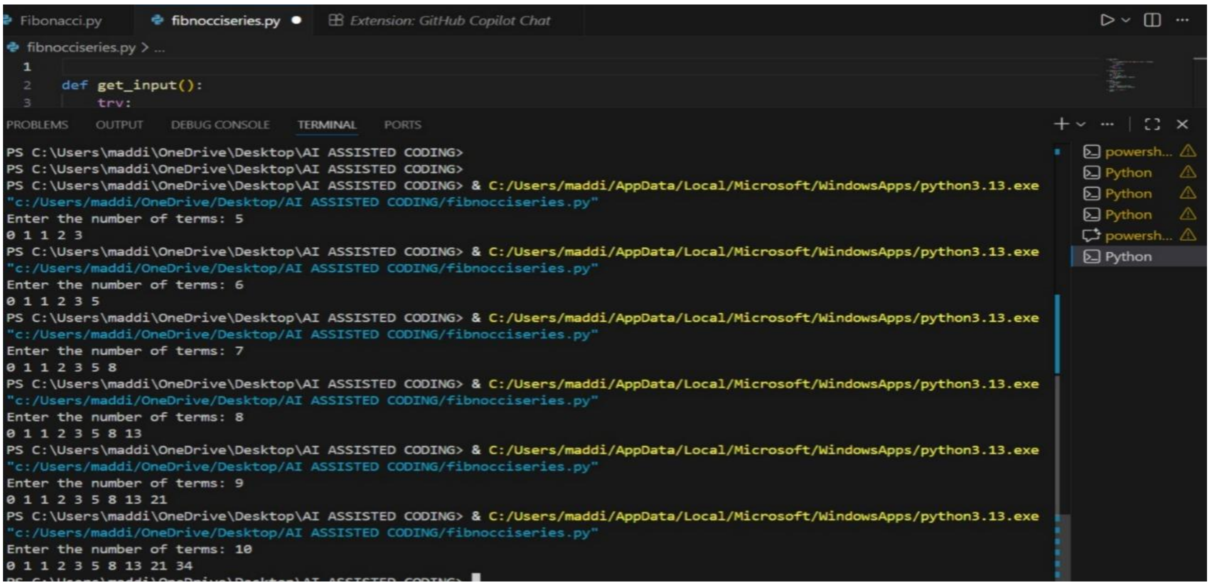
Write a modular Python program that generates the Fibonacci series up to n terms using functions. Take n as input from the user.Use a separate function to compute the Fibonacci sequence efficiently and return the result. Print the output in one line with spaces between the numbers.




Step-2:Code




Step-03:Output




Step-04:Explanation

1. The program asks the user to enter a number n.
2. A function is created to generate Fibonacci numbers instead of writing everything in one place.
3. Inside the function, the series starts with 0 and 1.
4. A loop calculates the next numbers by adding the previous two.
5. All the Fibonacci numbers are stored in a list and returned by the function.
6. The main part of the program calls the function and gets the list.
7. Then it prints the numbers in one single line with spaces between them.
8. Using functions makes the code clean, reusable, and easier to improve later.

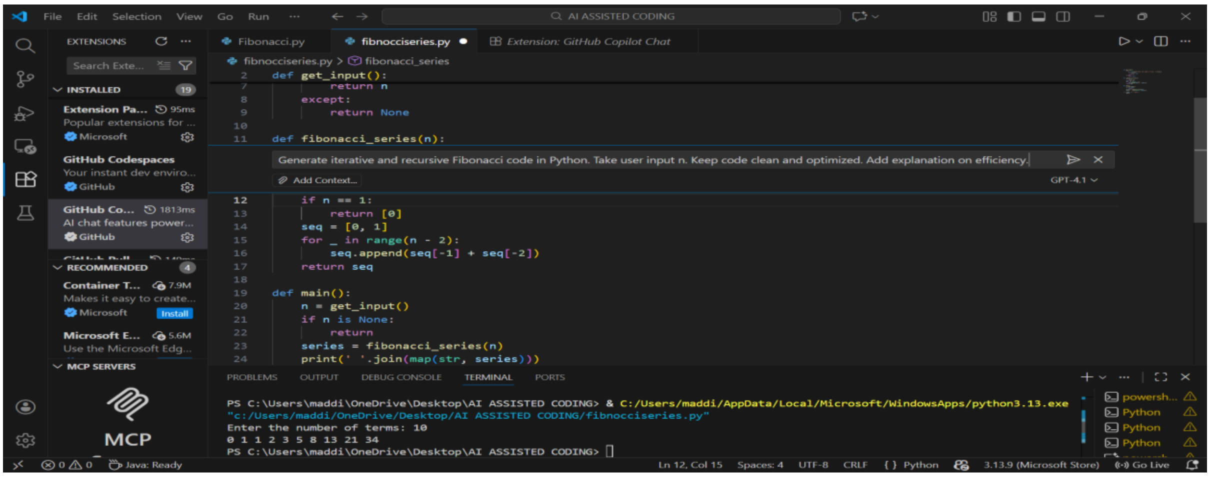
## Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code.

| **Feature** | **Without Functions** | **With Functions** |
|---|---|---|
| Code Clarity | Logic is written in one block; harder to read when long | Logic is separated into a named function → easier to understand |
| Reusability | Cannot reuse Fibonacci logic without rewriting | Can call the function anywhere in program |
| Debugging Ease | Bugs must be traced in main logic, mixed with other code | Errors isolated in function → easier to test & fix |
| Suitability for Larger Systems | Poor; not scalable, becomes messy with added features | Good; fits into bigger systems, easier to maintain |
| Testing | Hard to unit test a part of code independently | Function can be tested separately with multiple inputs |
| Maintainability | Low; changes affect entire code block | High; changes only in function, no impact on main flow |
| Performance Impact | No function call overhead (very small benefit) | Minimal overhead but worth it for structure & scaling |

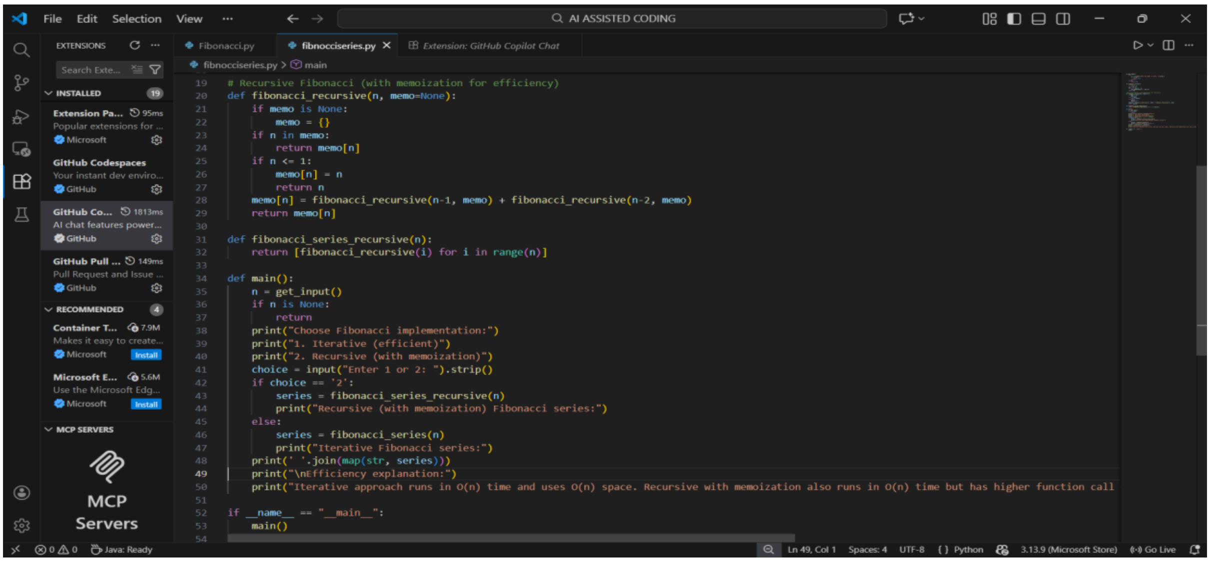
## Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

## Step-01:Prompt

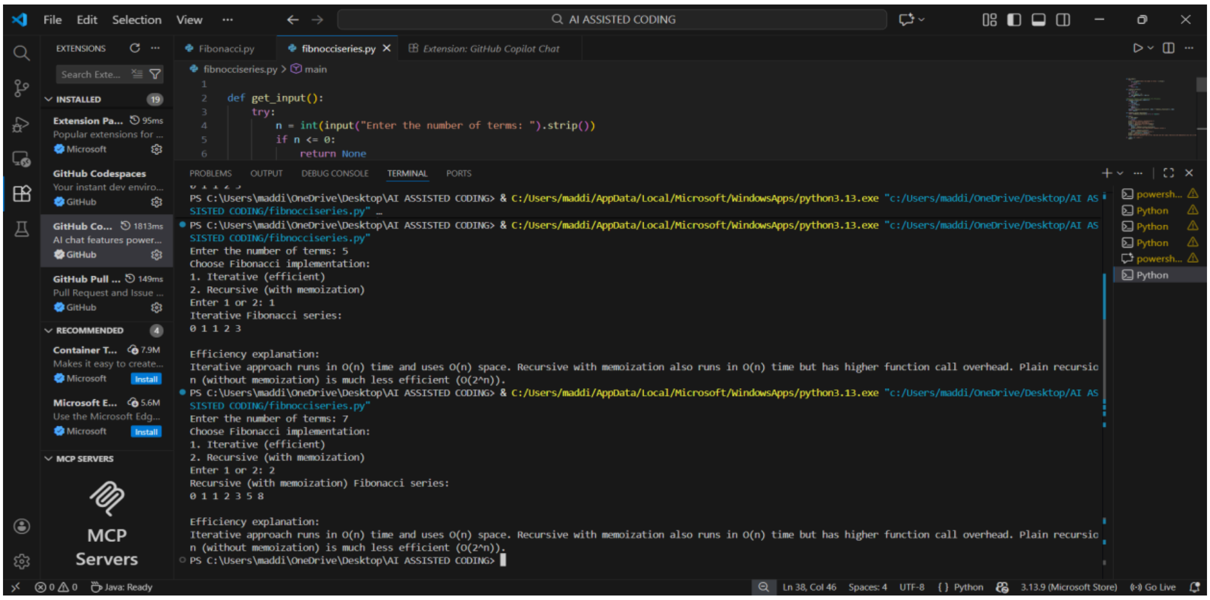
Generate iterative and recursive Fibonacci code in Python. Take user input n. Keep code clean and optimized. Add explanation on efficiency.



Step-02:Code



Step-03:Output




Step-04:Explanation

**Iterative Approach**

1. The program takes n from the user.
2. It starts Fibonacci with two variables: a = 0, b = 1.
3. A loop runs n times to create the sequence step by step.
4. In each round, it prints a, then updates the values (a becomes b, b becomes a + b).
5. This method is fast, uses very little memory, and works well even for big n.

**Efficiency:**

- Time: O(n) → runs in a straight line with loop
- Space: O(1) → only 2 variables used

---

**Recursive Approach**

1. The program takes n from the user.
2. A recursive function calls itself to find Fibonacci numbers.
3. It breaks the problem into smaller parts: fib(n) = fib(n-1) + fib(n-2).
4. It keeps calling itself until it reaches the base values (0 and 1).
5. This method is slow for large n because it repeats the same work many times.

**Efficiency:**

- Time: $O(2^n)$ → grows very fast, not good for big n

- Space: O(n) → stack memory is used for every call

**Comparison:**

| Aspect | Iterative | Recursive |
|---|---|---|
| Time Complexity | O(n) | O($2^n$) (very slow due to repeated calls) |
| Space Complexity | O(1) | O(n) (stack memory for calls) |
| Performance for Large n | Excellent (can handle $10^7$+ if needed) | Poor (fib(50) may take seconds/minutes) |
| Memory Usage | Very low | High because of recursion stack |
| Scalability | Best for real systems | Not scalable without optimization |
| Risk | No crash risk | StackOverflow for large n |