

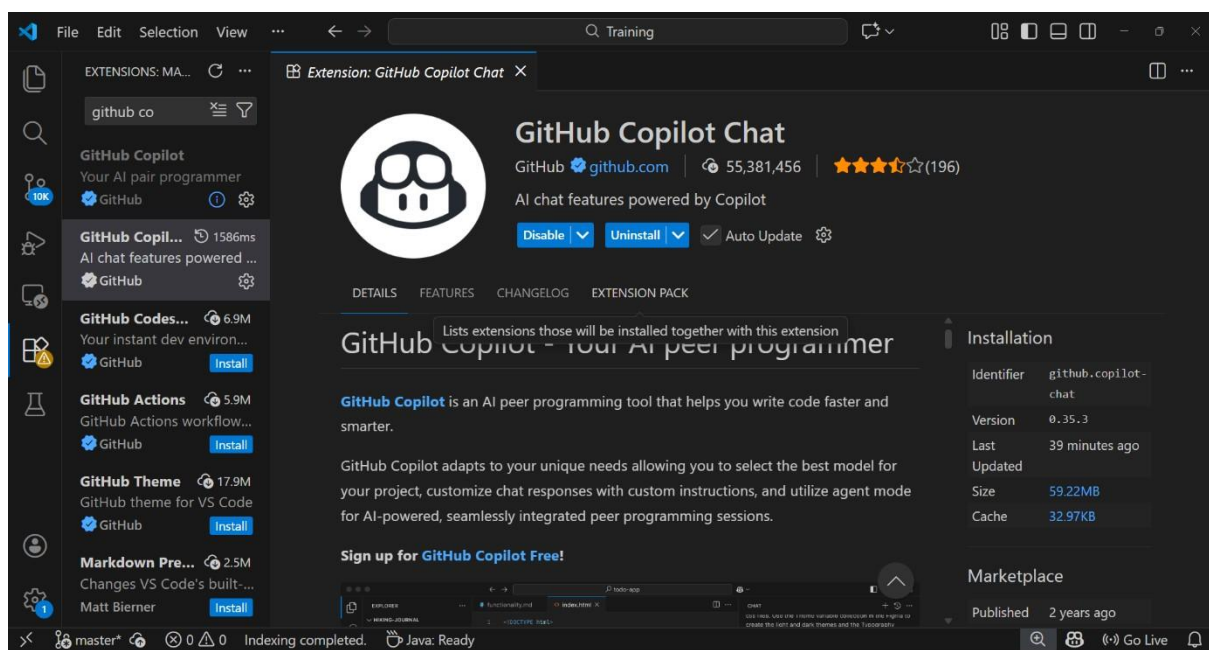
AI Assistant Coding Assignment-1

Name : Saketh Birru

Roll no : 2303A51403

Batch : 06

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



Visual Studio Code interface showing the Python extension page. The left sidebar displays the Extensions view with a search for 'python'. The main panel shows the 'Python' extension by Microsoft, with a description: 'Python language support with extension access points for IntelliSense (Pylance), Debugging...'. The extension is installed and updated. The right sidebar shows the 'Installation' details, including the identifier 'ms-python.python', version '2025.20.1', and size '29.61MB'. The bottom status bar indicates 'Indexing completed' and 'Java: Ready'.

Python extension for Visual Studio Code

A Visual Studio Code extension with rich support for the Python language (for all actively supported Python versions), providing access points for extensions to seamlessly integrate and offer support for IntelliSense (Pylance), debugging (Python Debugger), formatting, linting, code navigation, refactoring, variable explorer, test explorer, environment management (NEW Python Environments Extension).

Support for vscod.dev

The Python extension does offer some support when running on vscod.dev (which

Installation	
Identifier	ms-python.python
Version	2025.20.1
Last Updated	42 minutes ago
Size	29.61MB
Cache	651B

Marketplace	
Published	9 years ago
Last	17 hours ago

Visual Studio Code interface showing the GitHub Copilot Chat extension page. The left sidebar displays the Explorer view with a search for 'chat.py'. The main panel shows the 'GitHub Copilot Chat' extension by GitHub, with a description: 'AI chat features powered by Copilot'. The extension is installed and updated. The right sidebar shows the 'Installation' details, including the identifier 'github.copilot-chat', version '0.35.3', and size '59.22MB'. The bottom status bar indicates 'Indexing completed' and 'Java: Ready'.

GitHub Copilot Chat

AI chat features powered by Copilot

GitHub Copilot - Your AI peer programmer

GitHub Copilot is an AI peer programming tool that helps you write code faster and smarter.

GitHub Copilot adapts to your unique needs allowing you to select the best model for your project, customize chat responses with custom instructions, and utilize agent mode for AI-powered, seamlessly integrated peer programming sessions.

Installation	
Identifier	github.copilot-chat
Version	0.35.3
Last Updated	37 minutes ago
Size	59.22MB
Cache	32.97KB

Marketplace	
Published	2 years ago

for above program also include the subtraction function

Proceeding to open `chat.py` to inspect where to add the subtraction function; I'll check the current functions and behavior.

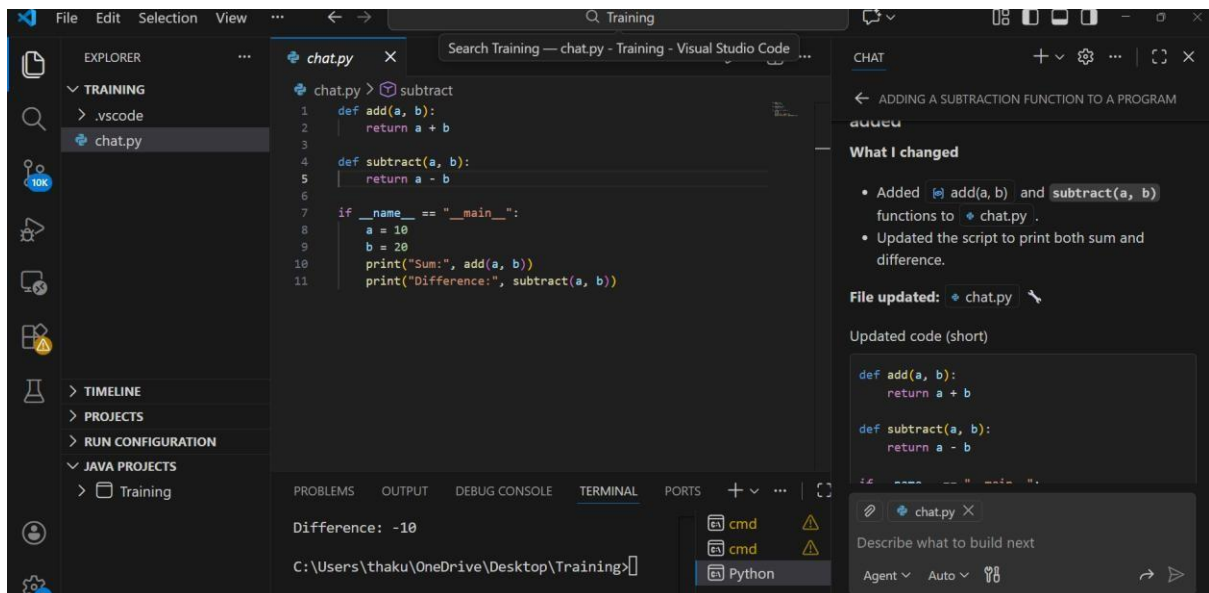
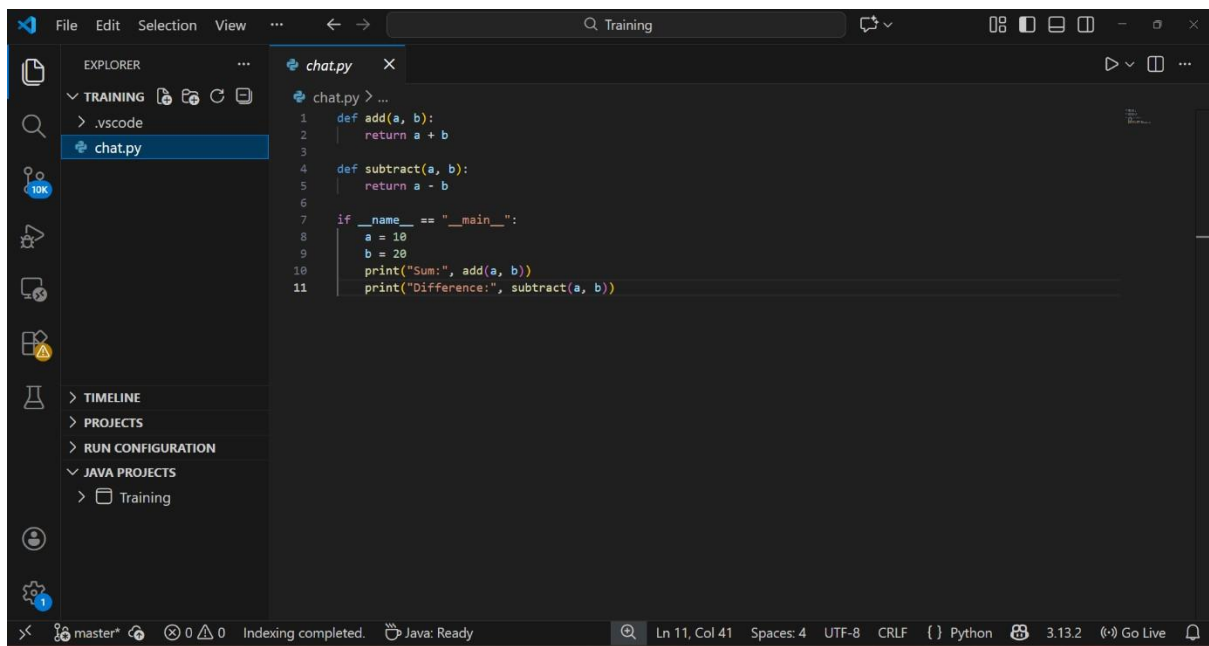
Modified code structure and implemented subtraction function

Great! I'll add `add` and `subtract` functions and update `chat.py`

Add Context...

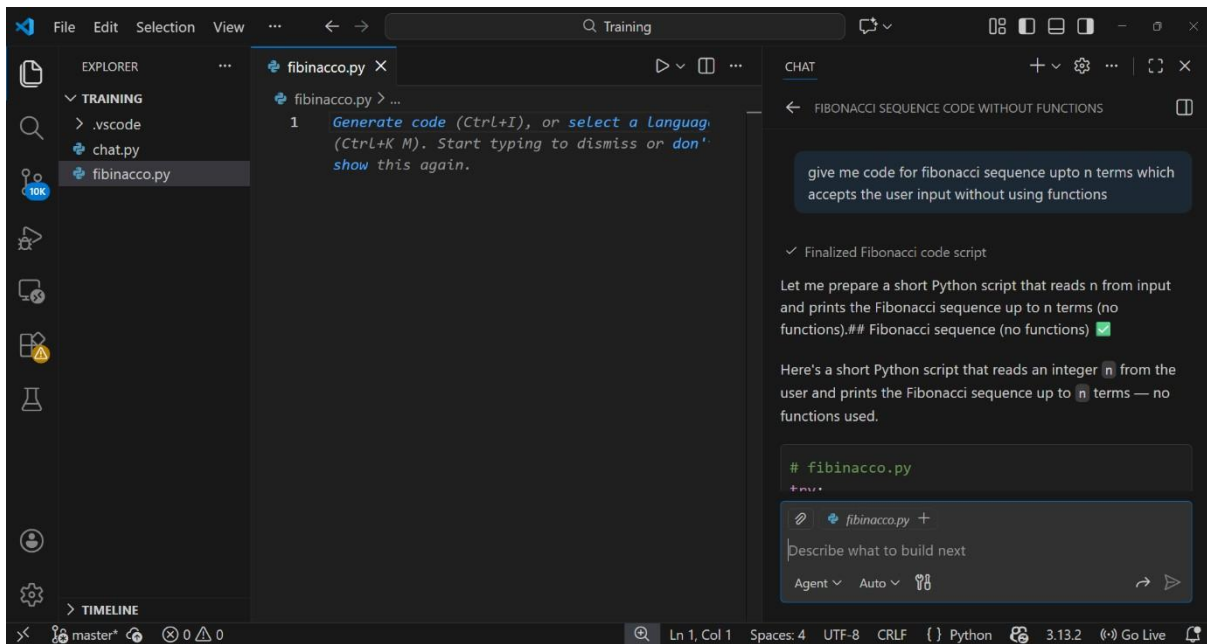
Describe what to build next

Agent Auto

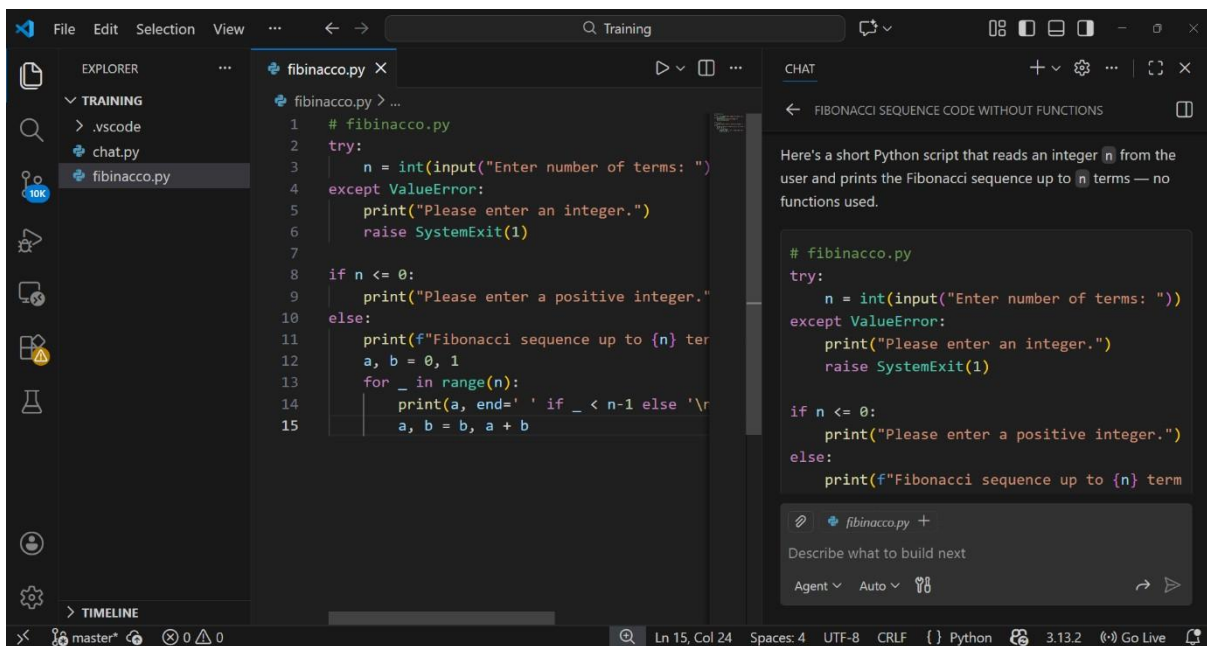


Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

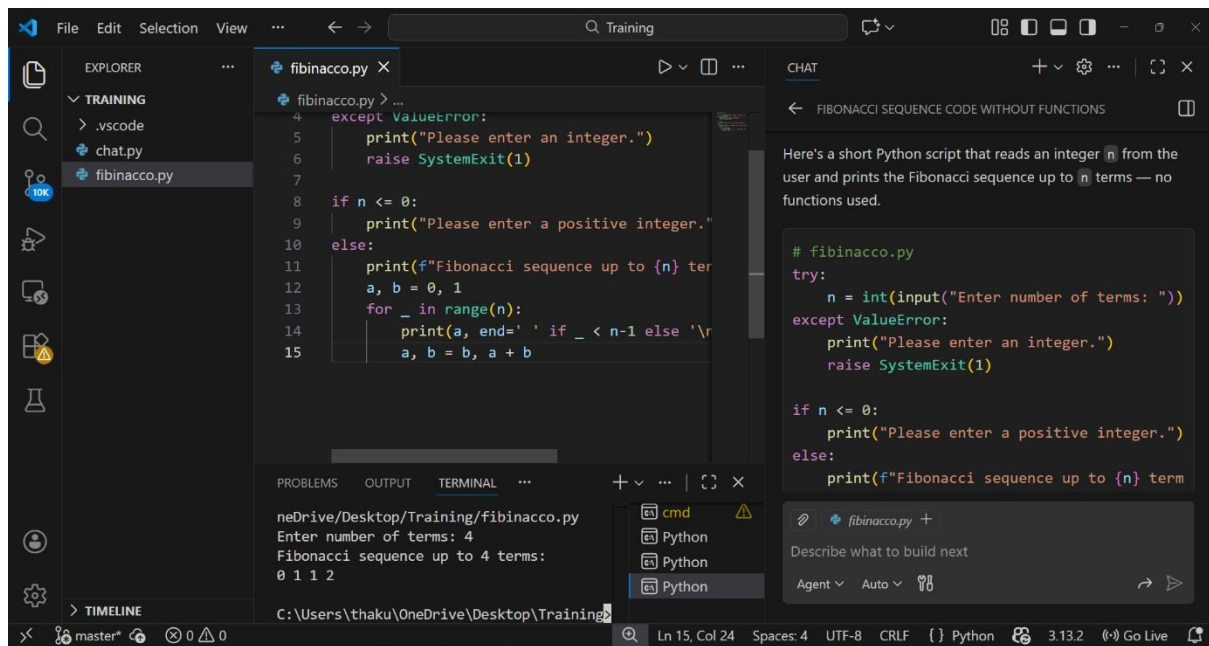
Step 1: Giving prompt: Give me code for fibonacci sequence up to n terms which accepts the user input without using functions



Step 2: Code



Step 3: Output



The screenshot shows the Visual Studio Code editor with a file named `fibinacco.py` open. The code in the file is as follows:

```
4 except ValueError:
5     print("Please enter an integer.")
6     raise SystemExit(1)
7
8 if n <= 0:
9     print("Please enter a positive integer.")
10 else:
11     print(f"Fibonacci sequence up to {n} terms")
12     a, b = 0, 1
13     for _ in range(n):
14         print(a, end=' ' if _ < n-1 else '\n')
15         a, b = b, a + b
```

The terminal output shows the execution of the script:

```
neDrive/Desktop/Training/fibinacco.py
Enter number of terms: 4
Fibonacci sequence up to 4 terms:
0 1 1 2
```

The chat window on the right provides a summary of the script's functionality:

Here's a short Python script that reads an integer `n` from the user and prints the Fibonacci sequence up to `n` terms — no functions used.

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} terms")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ' if _ < n-1 else '\n')
        a, b = b, a + b
```

Step 4: Explanation of Code

Input Handling:

- Uses try-except to ensure the user enters an integer.
- Exits gracefully if input is invalid.

Validation:

- Checks if `n <= 0`.
- Prompts user to enter a positive integer if invalid.

Output Message:

- Prints a header with correct pluralization (term vs terms).

Initialization:

- Starts with `a = 0, b = 1` (first two Fibonacci numbers).

Loop Logic:

- Runs for `_ in range(n)` to generate `n` terms.
- Prints current value of `a`.
- Updates values with `a, b = b, a + b`.

Printing Style:

- Numbers are printed on the same line separated by spaces.
- Last number ends with a newline.

Result:

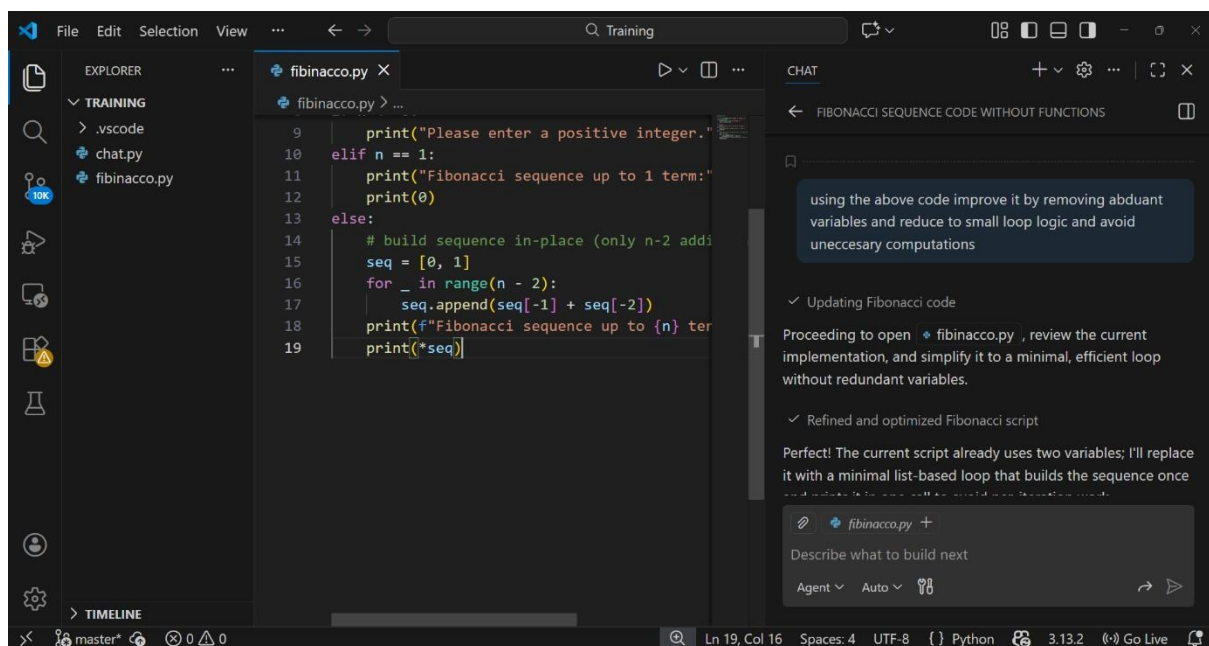
- Produces the Fibonacci sequence up to n terms.

Example:

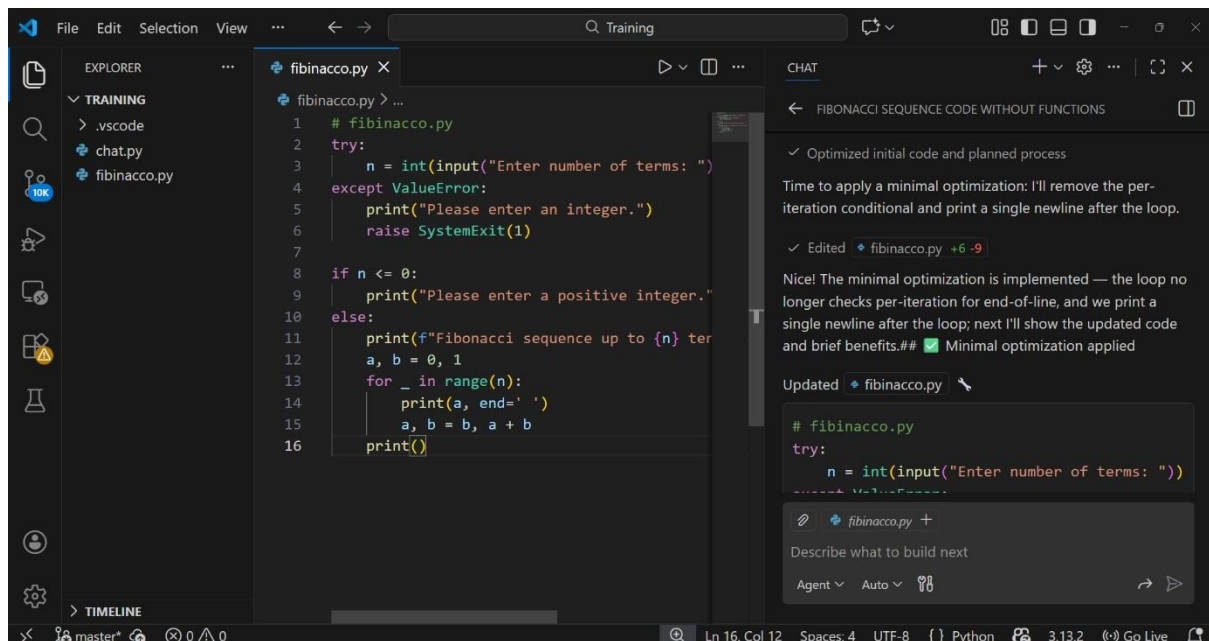
Input: 4 → Output: 0 1 1 2

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

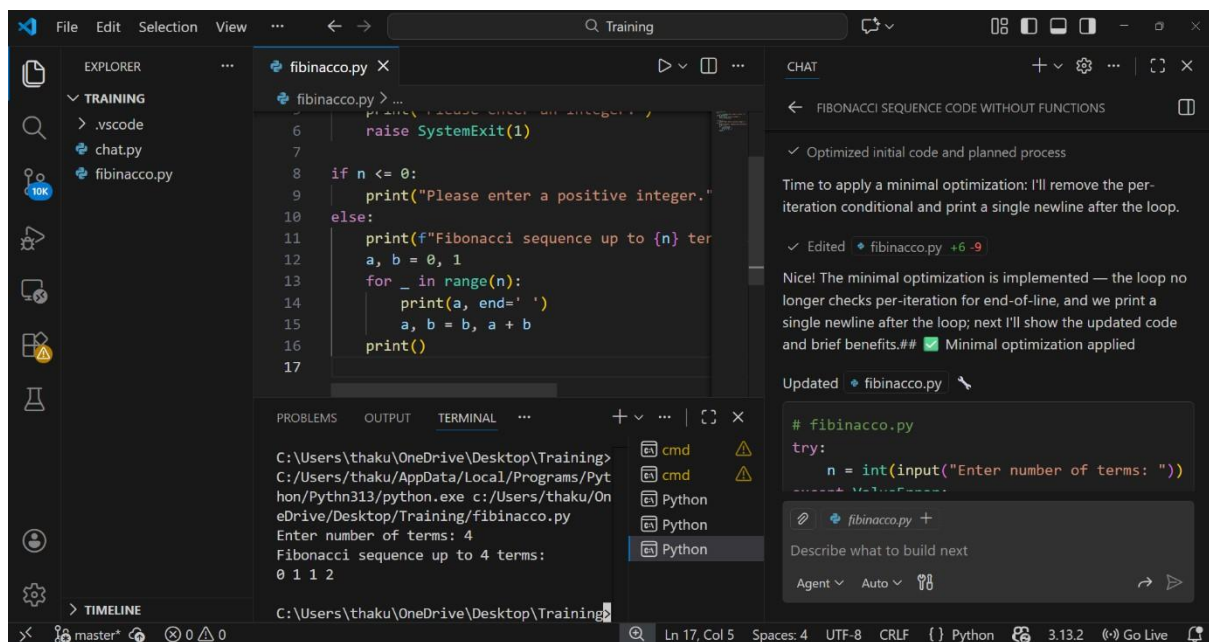
Step 1: Giving Prompt: Using the above code improve it by removing abundant variables and reduce to small loop logic and avoid unnecessary computations / optimize the code



Step 2: Code



Step 3: Output



Step 4: Code Explanation

Input Handling

- Uses `try-except` to ensure the user enters an integer.
- Exits safely if input is invalid.

Validation

- Checks if `n <= 0`.

- Prompts user to enter a positive integer if invalid.

Output Message

- Prints a header with correct pluralization (term vs terms).

Initialization

- Starts with $a = 0$, $b = 1$ (first two Fibonacci numbers).

Loop Logic

- Runs for `_` in `range(n)` to generate n terms.
- Prints current value of a .
- Updates values with $a, b = b, a + b$.

What Was Inefficient in the Original Code

The original code had stuff that wasn't needed to generate the Fibonacci sequence.

A volatile memory variable was used to hold the sum and added extra length.

It's because you're putting the fibonacci sequence into a list as well a program like this will just output the result, but store it in memory somewhere.

The loop was constructed in more than one step, which made the code less readable

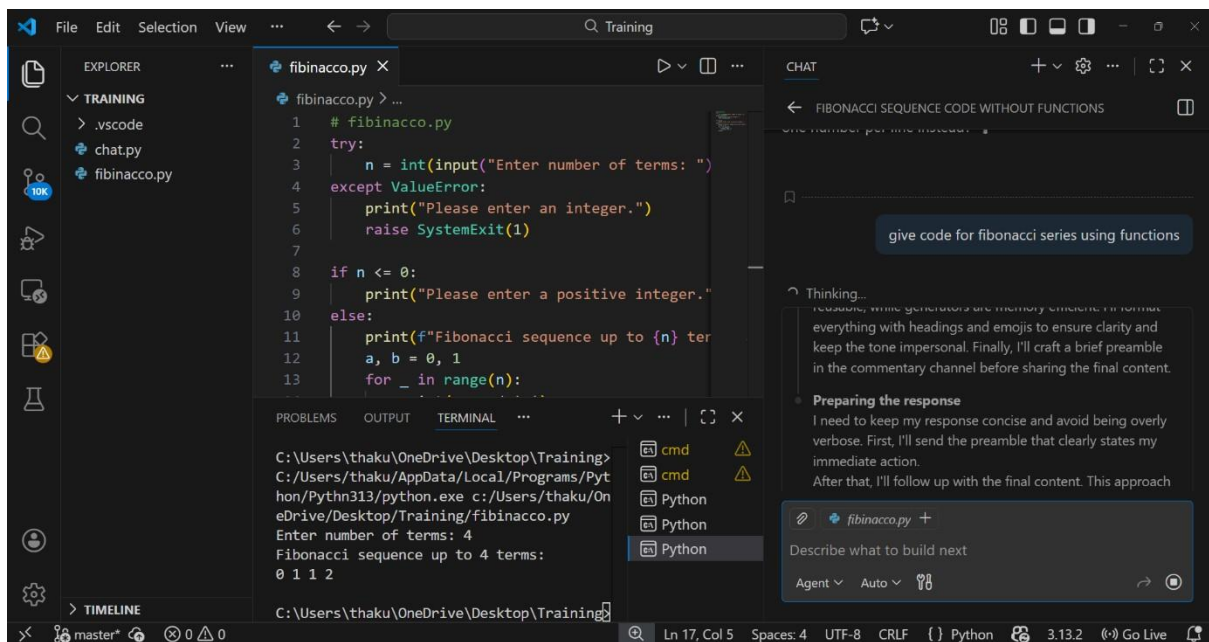
How the optimized/python version brings performance improvement and improved readability.

Unnecessary variables were discarded and the values were updated by tuple assignment in an efficient manner.

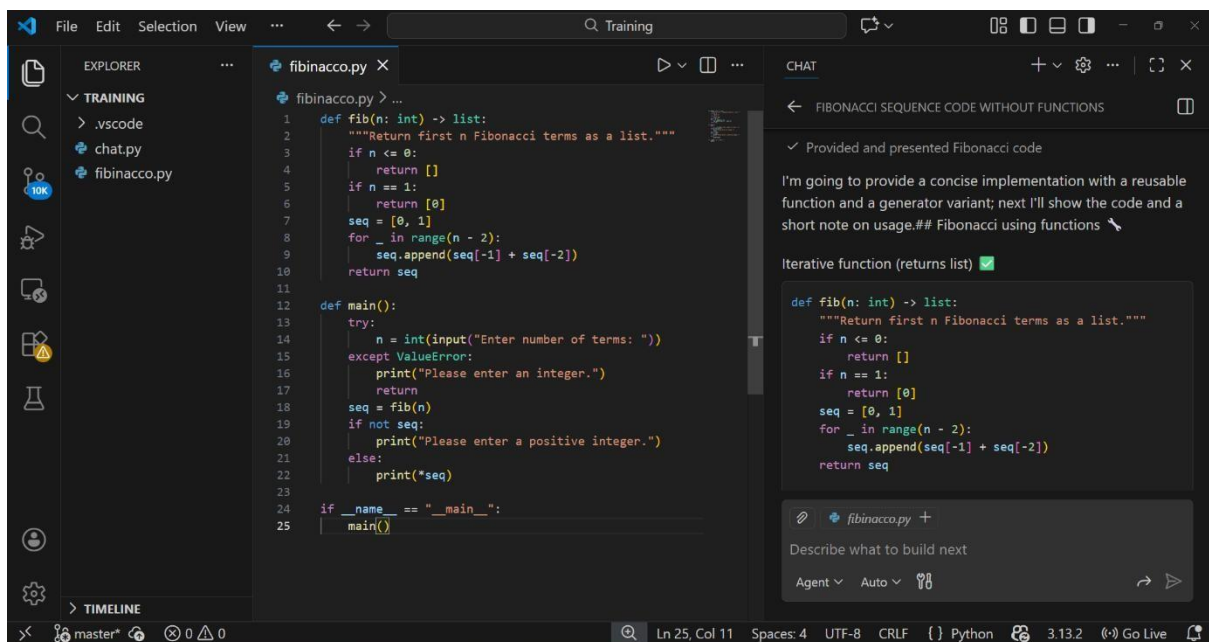
Your code also writes to disc, I assume you want to speed the codeup, so this is not good. Your optimized version that this saved memory.3) that it prints the number instead of storing a whole vector with all solutions.

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

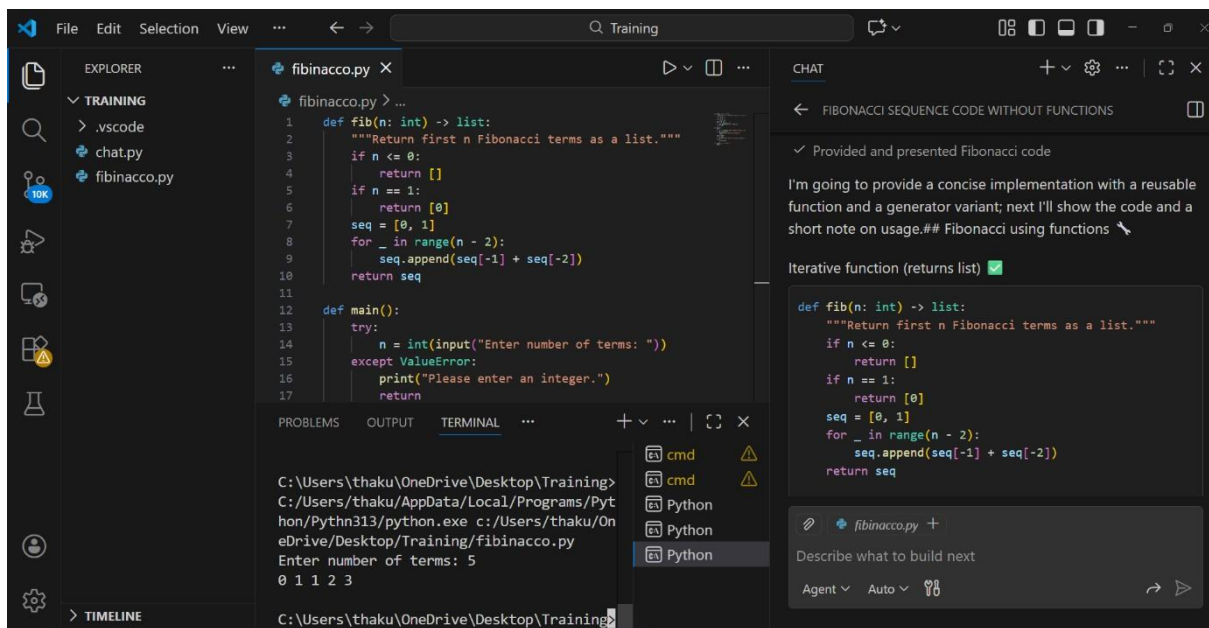
Step 1 : Prompt: Give code for Fibonacci sequence using functions.



Step 2: Code:



Step 3: Output



The screenshot shows the Visual Studio Code editor with a file named `fibonacci.py`. The code defines a `fib` function and a `main` function. The terminal shows the execution of the script, where the user enters 5, and the output is 0 1 1 2 3. The chat interface on the right shows a conversation about the Fibonacci code, with the AI providing a concise implementation and a short note on usage.

```
1 def fib(n: int) -> list:
2     """Return first n Fibonacci terms as a list."""
3     if n <= 0:
4         return []
5     if n == 1:
6         return [0]
7     seq = [0, 1]
8     for _ in range(n - 2):
9         seq.append(seq[-1] + seq[-2])
10    return seq
11
12 def main():
13     try:
14         n = int(input("Enter number of terms: "))
15     except ValueError:
16         print("Please enter an integer.")
17     return
```

Terminal Output:

```
C:\Users\thaku\OneDrive\Desktop\Training>
C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibonacci.py
Enter number of terms: 5
0 1 1 2 3
```

Chat Interface:

FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS

✓ Provided and presented Fibonacci code

I'm going to provide a concise implementation with a reusable function and a generator variant; next I'll show the code and a short note on usage.## Fibonacci using functions ↵

Iterative function (returns list) ✓

```
def fib(n: int) -> list:
    """Return first n Fibonacci terms as a list."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    for _ in range(n - 2):
        seq.append(seq[-1] + seq[-2])
    return seq
```

Describe what to build next

Agent Auto

Step 4: Explanation of code

Function Design

- o `fib(n)` cleanly separates Fibonacci logic from input/output.
- o Returns a list of the first `n` Fibonacci terms.

Validation

- o If `n <= 0`: returns an empty list.
- o If `n == 1`: returns `[0]`.
- o Handles edge cases gracefully.

Sequence Generation

- o Starts with `[0, 1]`. o Uses a loop for `_ in range(n- 2)` to generate remaining terms.
- o Each new term = sum of last two (`seq[-1] + seq[-2]`).

Main Function

- o Prompts user for input. o Uses try-except to catch invalid input (`ValueError`).
- o Calls `fib(n)` to generate sequence. o Prints a warning if input is invalid or non-positive.

Output

- Prints the sequence neatly with `print(*seq)` (space-separated values).

Practices

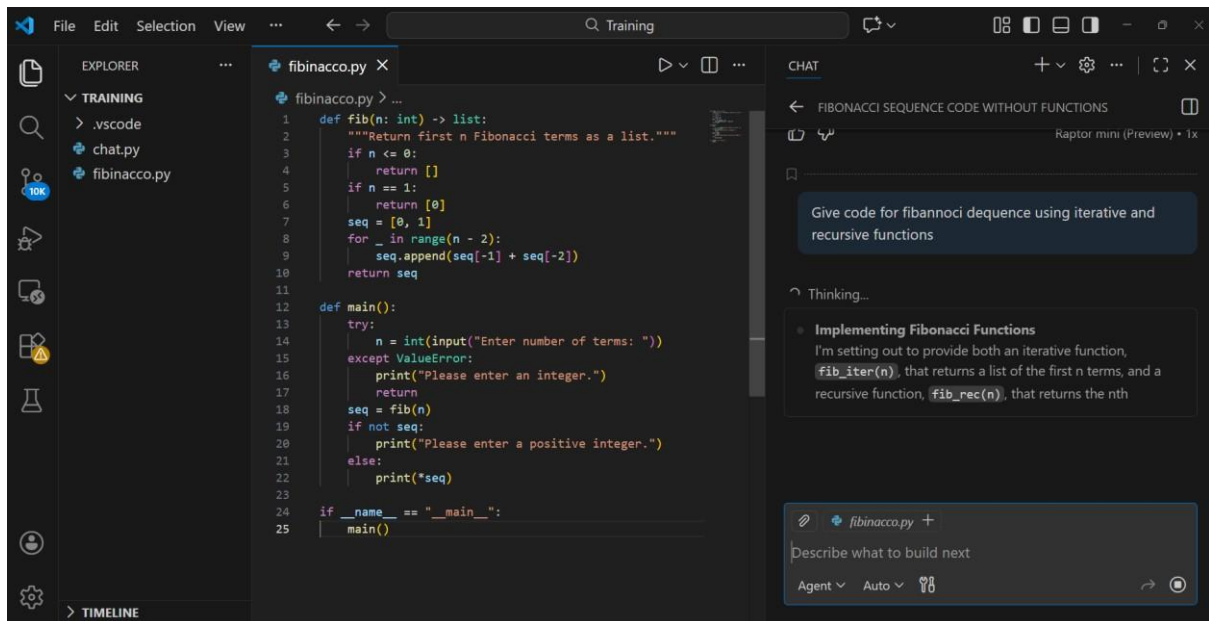
- Includes a docstring for clarity.
- Uses `__name__ == "__main__"` guard for modularity.
- Keeps logic reusable and testable.

Task 4 : Comparative Analysis – Procedural vs Modular Fibonacci Code

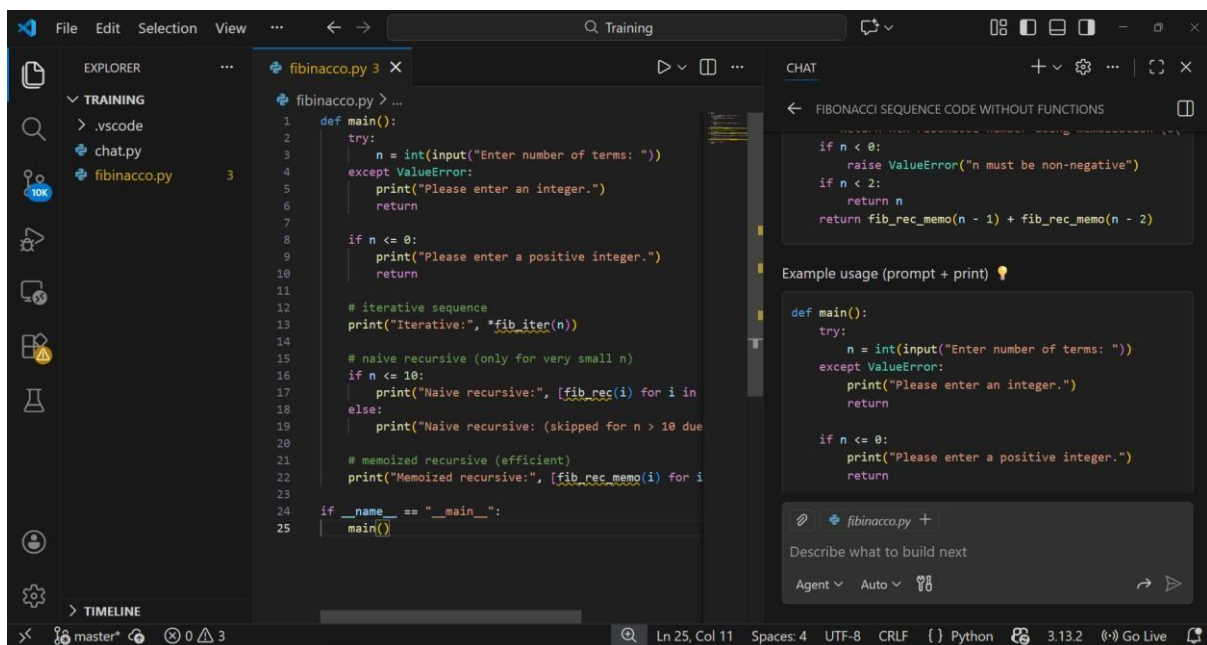
Feature	Without Functions	With Functions
Code Clarity	Logic is written in one block; harder to read when long	Logic is separated into a named function → easier to understand
Reusability	Cannot reuse Fibonacci logic without rewriting	Can call the function anywhere in program
Debugging Ease	Bugs must be 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 1 : Prompt: Give code for fibanocci series using iterative and recursive functions



Step 2: Code Iterative:



Step 3: Output

```

4 def fib_iter(n: int) -> list:
6     if n <= 0:
7         return []
8     if n == 1:
9         return [0]
10    seq = [0, 1]
11    for _ in range(n - 2):
12        seq.append(seq[-1] + seq[-2])
13    return seq
14
15
16 def fib_rec(n: int) -> int:
17     """Naive recursive: return the nth Fibonacci number (0-indexed)."""
18     if n < 0:
19         raise ValueError("n must be non-negative")
20     if n < 2:
21         return n

```

Enter number of terms: 4
Iterative: 0 1 1 2
Naive recursive: [0, 1, 1, 2]
Memoized recursive: [0, 1, 1, 2]

Step 4: Explanation of code

fib_iter(n) → Iterative list builder, efficient $O(n)$.

- fib_rec(n) → Naive recursion, exponential cost, only practical for very small n.
- fib_rec_memo(n) → Recursive with @lru_cache, efficient $O(n)$ via memoization.
- Input Handling → Validates integer input, rejects negatives/zero.
- Main Function → o Prints iterative sequence always. o Prints naive recursive only if $n \leq 10$. o Prints memoized recursive for any n.
- Good Practices → Clear separation of methods, input validation, modular design, efficient memoization.

Example (n=6):

Iterative: 0 1 1 2 3 5

Naive recursive: [0, 1, 1, 2, 3, 5]

Memoized recursive: [0, 1, 1, 2, 3, 5]

Comparison covering:

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	Stack Overflow for large n