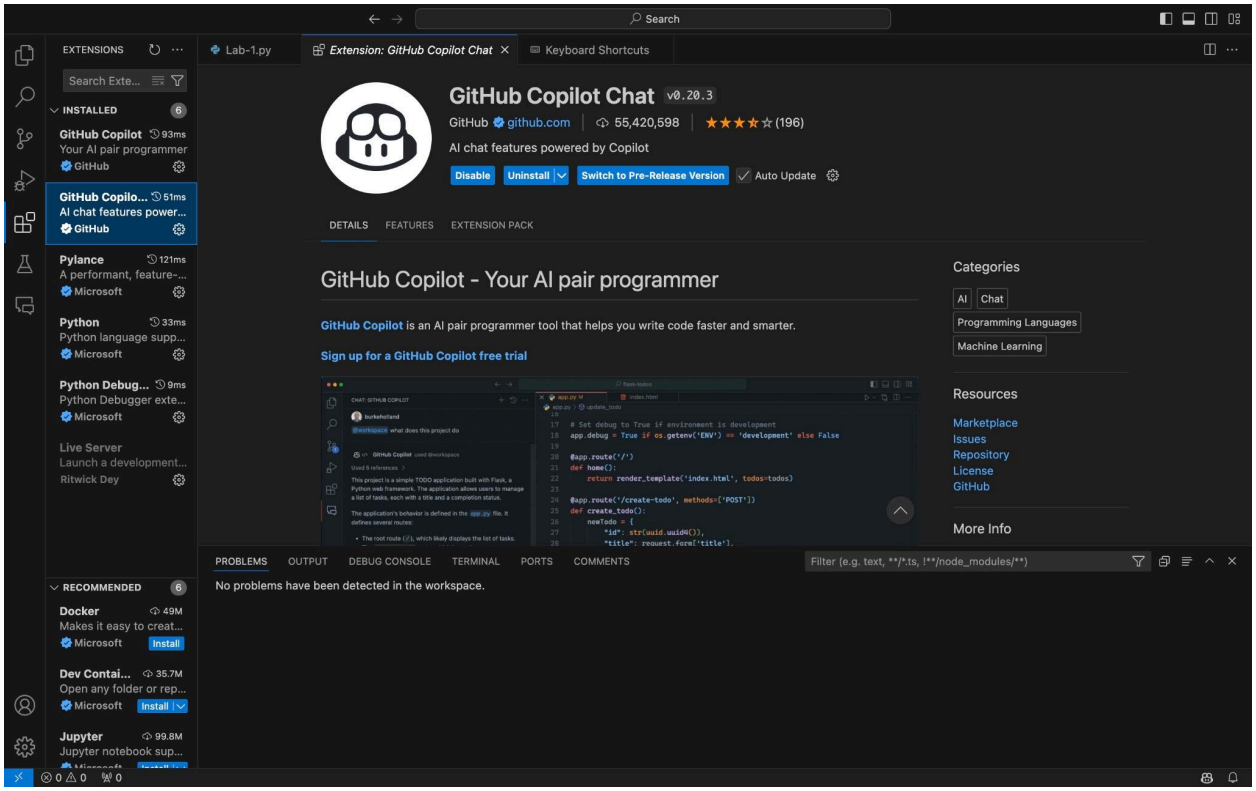
# AI Assisted Coding
# Assignment -1.3
# Roll no : 2303A510C8

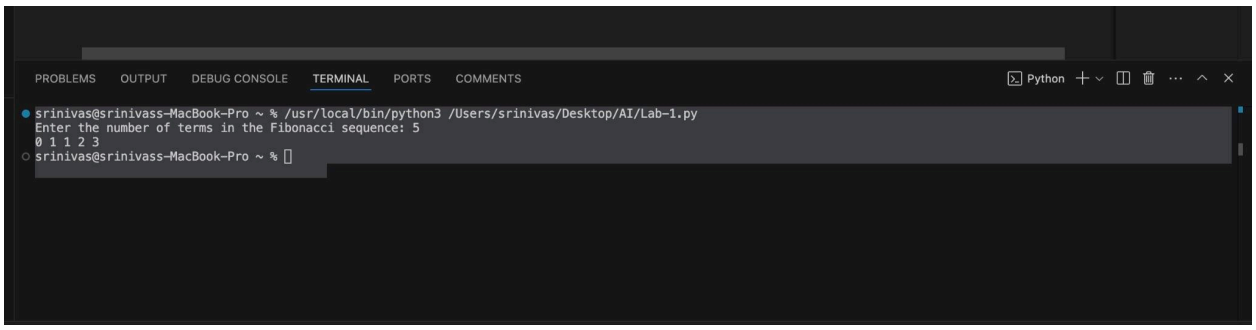
## Task 0 : Install and configure GitHub Copilot in VS Code. Take screenshots of each step.




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

```python
# Write a Python program that accepts user input for 'n' and prints the Fibonacci sequence up to n terms using a loop. Do not use any functions.
# Accept user input for 'n'
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
# Initialize the first two terms of the Fibonacci sequence
a, b = 0, 1
# Print the Fibonacci sequence up to n terms
for _ in range(n):
    print(a, end=' ')
    a, b = b, a + b
print()  # For a new line after the sequence
```

# Output:

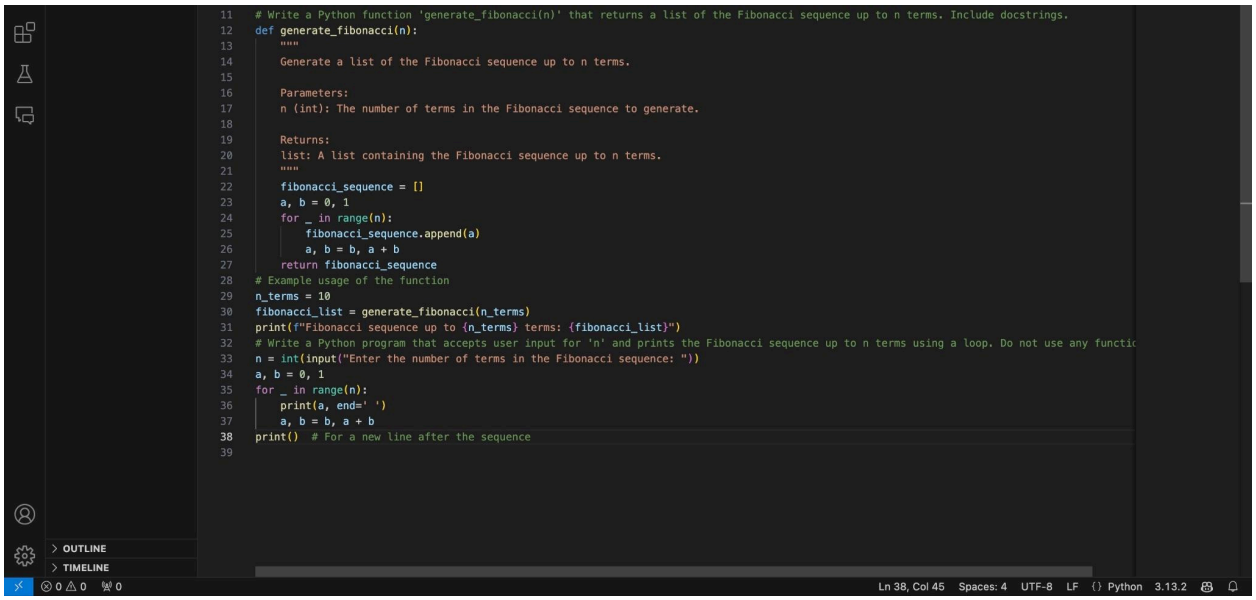



**Explanation**

1. **Initialization:**
    - The program begins by initializing the first two terms of the Fibonacci sequence: `t1 = 0` and `t2 = 1`.
    - A counter variable `count` is set to `0` to track how many numbers have been printed.
2. **Input Handling:**
    - The user inputs an integer `n` (the number of terms required).
3. **Conditional Logic (`if-elif-else`):**

- The code first validates the input. If `n <= 0`, it prompts for a positive integer.
- If `n == 1`, it handles the edge case by printing only the first term.

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



```
# Write a Python function 'generate_fibonacci(n)' that returns a list of the Fibonacci sequence up to n terms. Include docstrings.
def generate_fibonacci(n):
    """
    Generate a list of the Fibonacci sequence up to n terms.

    Parameters:
    n (int): The number of terms in the Fibonacci sequence to generate.

    Returns:
    list: A list containing the Fibonacci sequence up to n terms.
    """
    fibonacci_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fibonacci_sequence.append(a)
        a, b = b, a + b
    return fibonacci_sequence
# Example usage of the function
n_terms = 10
fibonacci_list = generate_fibonacci(n_terms)
print(f"Fibonacci sequence up to {n_terms} terms: {fibonacci_list}")
# Write a Python program that accepts user input for 'n' and prints the Fibonacci sequence up to n terms using a loop. Do not use any functic
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
a, b = 0, 1
for _ in range(n):
    print(a, end=' ')
    a, b = b, a + b
print()  # For a new line after the sequence
```

Output:

```
srinivas@srinivass-MacBook-Pro AI % /usr/local/bin/python3 /Users/srinivas/Desktop/AI/Lab-1.py
Enter the number of terms in the Fibonacci sequence: 10
0 1 1 2 3 5 8 13 21 34
Fibonacci sequence up to 10 terms: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Enter the number of terms in the Fibonacci sequence:
```

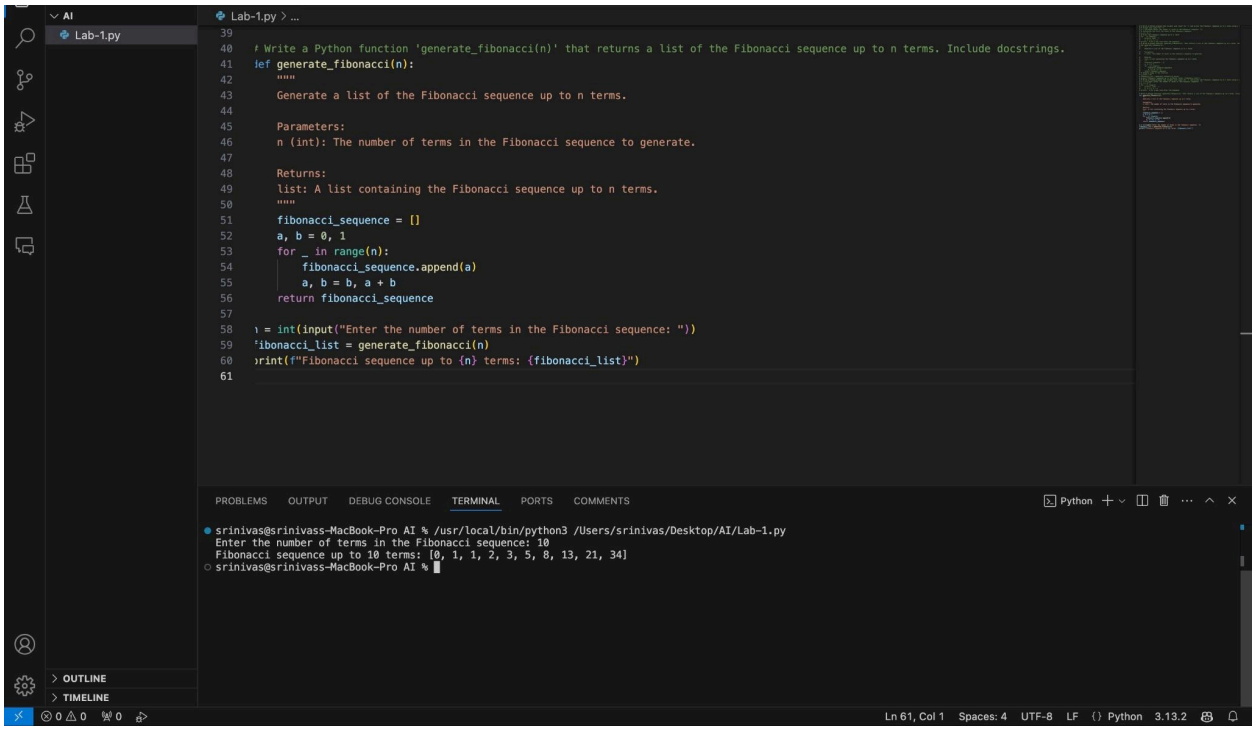
Explanation:

**Tuple Unpacking:** We replaced `nth = t1 + t2`, `t1 = t2`, `t2 = nth` with `a, b = b, a + b`. This removes the need for a temporary variable and reduces memory usage.

**For Loop vs While Loop:** A `for` loop using `range(n)` is generally more readable and less prone to infinite loop errors than a manually incremented `while` loop.

**Variable Naming:** Changed `t1/t2` to standard `a/b` for mathematical clarity.

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




# Explanation:

### Function Definition (`def`):

- We define a function named `generate_fibonacci(n)`. It takes one **parameter** (`n`), which represents the number of terms needed.

### Docstring (AI-Assisted):

- The text inside the triple quotes (`""" ... """`) is the docstring. It explains what the function does, the arguments it accepts, and what it returns. This is standard practice for readable code.

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

| Feature | Task 1: No Functions (Procedural) | Task 3: With Functions (Modular) |
|---|---|---|
| **Code Clarity** | Logic is mixed with input/output. Harder to read as code grows. | Logic is isolated. The main block is clean and easy to read. |
| **Reusability** | **Low.** You must copy-paste the loop to use it elsewhere. | **High.** You can import this function into any other Python file. |
| **Debugging** | **Harder.** Variables are global; changes affect the whole script. | **Easier.** You can test the function in isolation with different inputs. |
| **Suitability** | Suitable only for quick, one-off scripts. | Essential for large systems and professional development. |

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

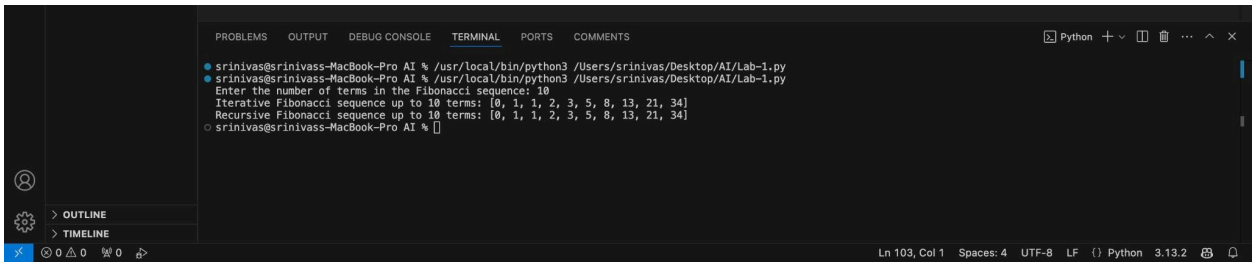
```
# Generate two Python functions: one for Iterative Fibonacci and one for Recursive
Fibonacci.
def iterative_fibonacci(n):
    """
    Generate a list of the Fibonacci sequence up to n terms using an iterative
approach.

    Parameters:
    n (int): The number of terms in the Fibonacci sequence to generate.

    Returns:
    list: A list containing the Fibonacci sequence up to n terms.
    """
    fibonacci_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fibonacci_sequence.append(a)
        a, b = b, a + b
    return fibonacci_sequence
def recursive_fibonacci(n):
    """
    Generate a list of the Fibonacci sequence up to n terms using a recursive approach.
    Parameters:
    n (int): The number of terms in the Fibonacci sequence to generate.
    Returns:
    list: A list containing the Fibonacci sequence up to n terms.
    """
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        seq = recursive_fibonacci(n - 1)
        seq.append(seq[-1] + seq[-2])
        return seq
```

```
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
iterative_result = iterative_fibonacci(n)
recursive_result = recursive_fibonacci(n)
print(f"Iterative Fibonacci sequence up to {n} terms: {iterative_result}")
print(f"Recursive Fibonacci sequence up to {n} terms: {recursive_result}")
```

Output:




Explanation:

**Iterative Approach:**

- **Time Complexity:** O(n) (Linear). It calculates each number exactly once.
- **Space Complexity:** O(1) (Constant). It only stores two variables (a and b).
- **Performance:** Very fast, even for large numbers (e.g., n=1000).

**Recursive Approach:**

- **Time Complexity:** O(2n) (Exponential). It recalculates the same values multiple times (e.g., to calculate F(5), it calculates F(3) twice).
- **Space Complexity:** O(n) due to the "Call Stack" memory used by function calls.
- **Performance:** Very slow for large n. If n>40, the program may freeze or crash due to "Stack Overflow."