Ai Assisted Coding

Assignment-1

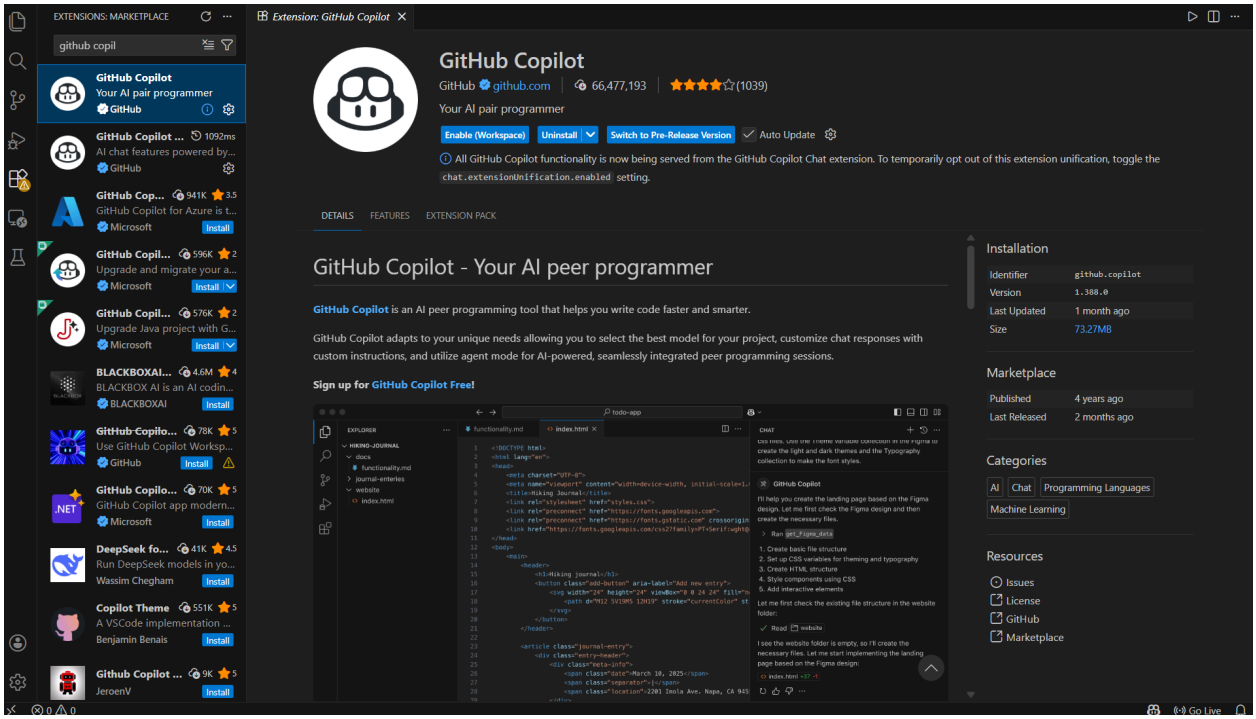
Name:Shaik Sabina

Batch-05

2303A51303

TASK-0

- Install and configure GitHub Copilot in VS Code.

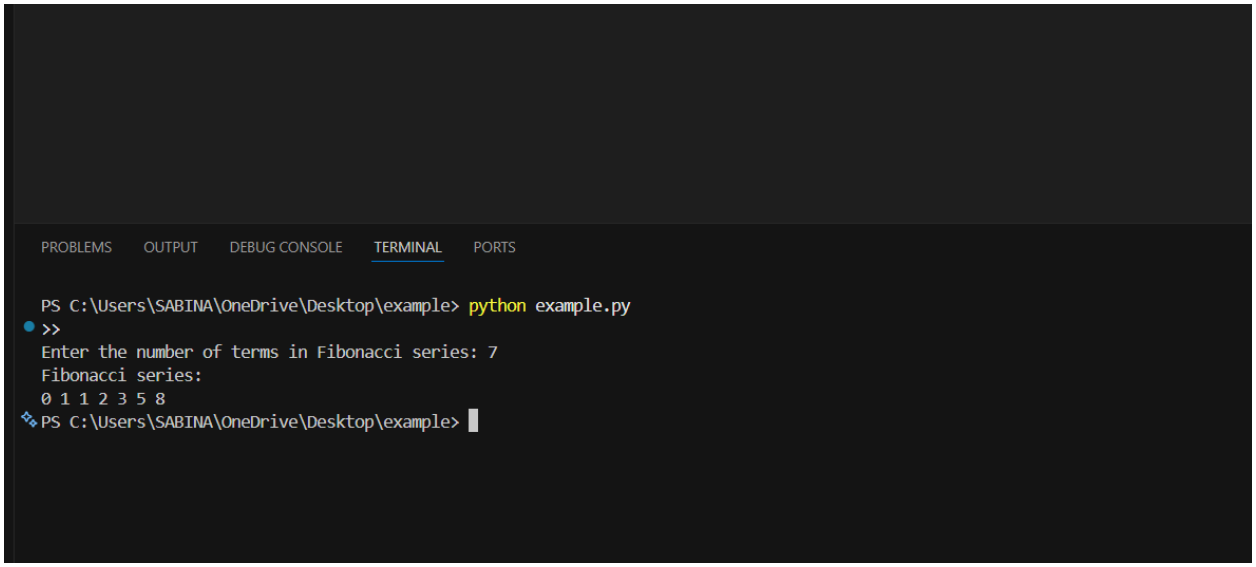


EXPLANATION:In this task, GitHub Copilot is installed and configured in Visual Studio Code to enable AI-assisted coding. Copilot provides real-time code suggestions based on comments and existing code context. This setup helps developers write code faster and reduces manual effort. Screenshots are taken to document each installation and configuration step.

TASK-1

PROMPT-Fibonacci Sequence Without Functions
CODE:

OUTPUT:

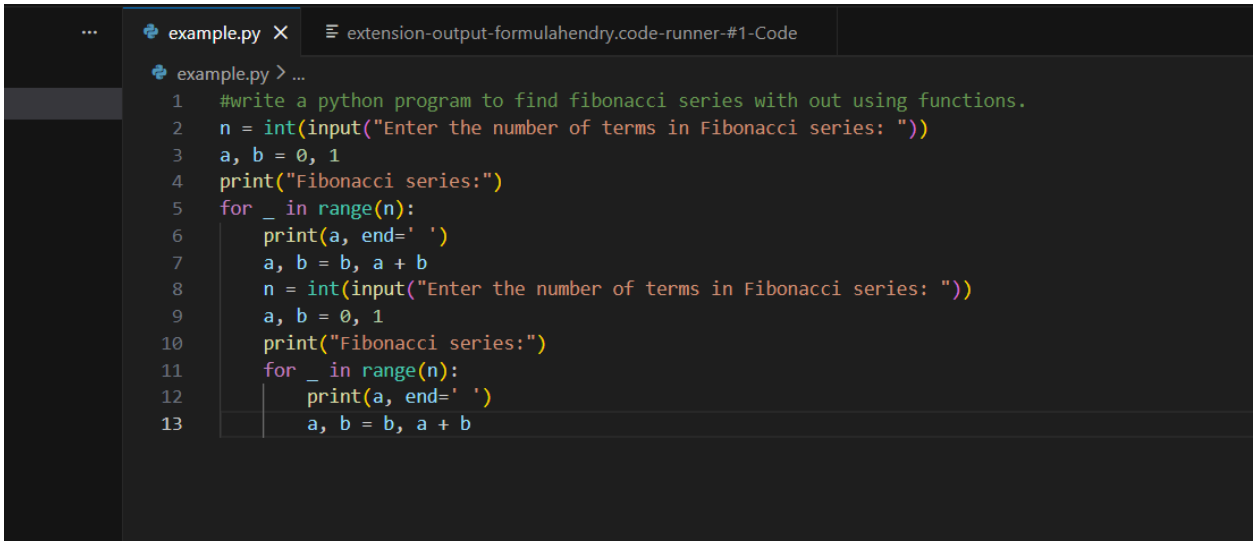


EXPLANATION:This task demonstrates how GitHub Copilot can generate a Python program for printing the Fibonacci sequence without using user-defined functions. The program accepts user input for the number of terms and implements the logic directly in the main code. This shows Copilot's ability to quickly generate correct procedural code using simple prompts and comments.

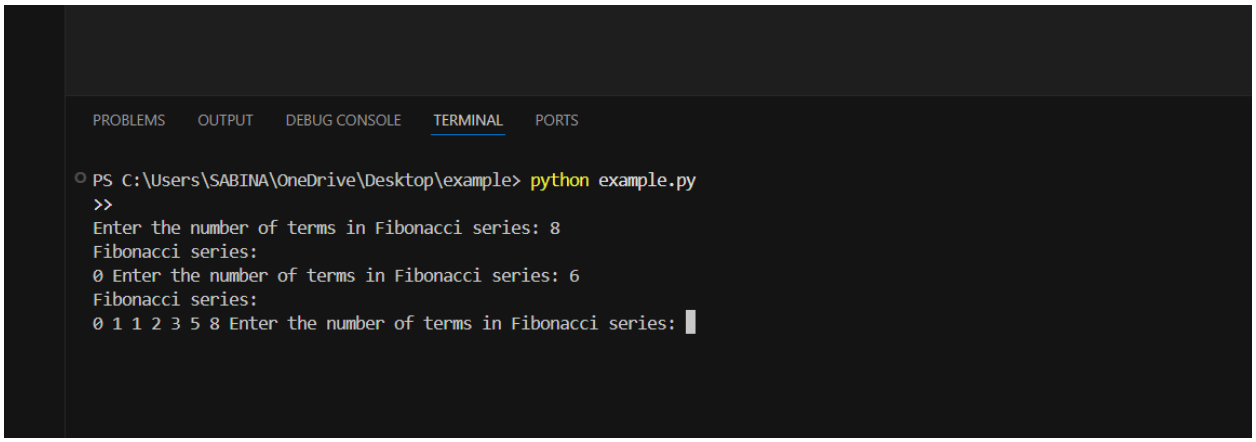
TASK-2

PROMPT-optimize the code using Ai code optimization and simply variable usage

CODE:



OUTPUT:

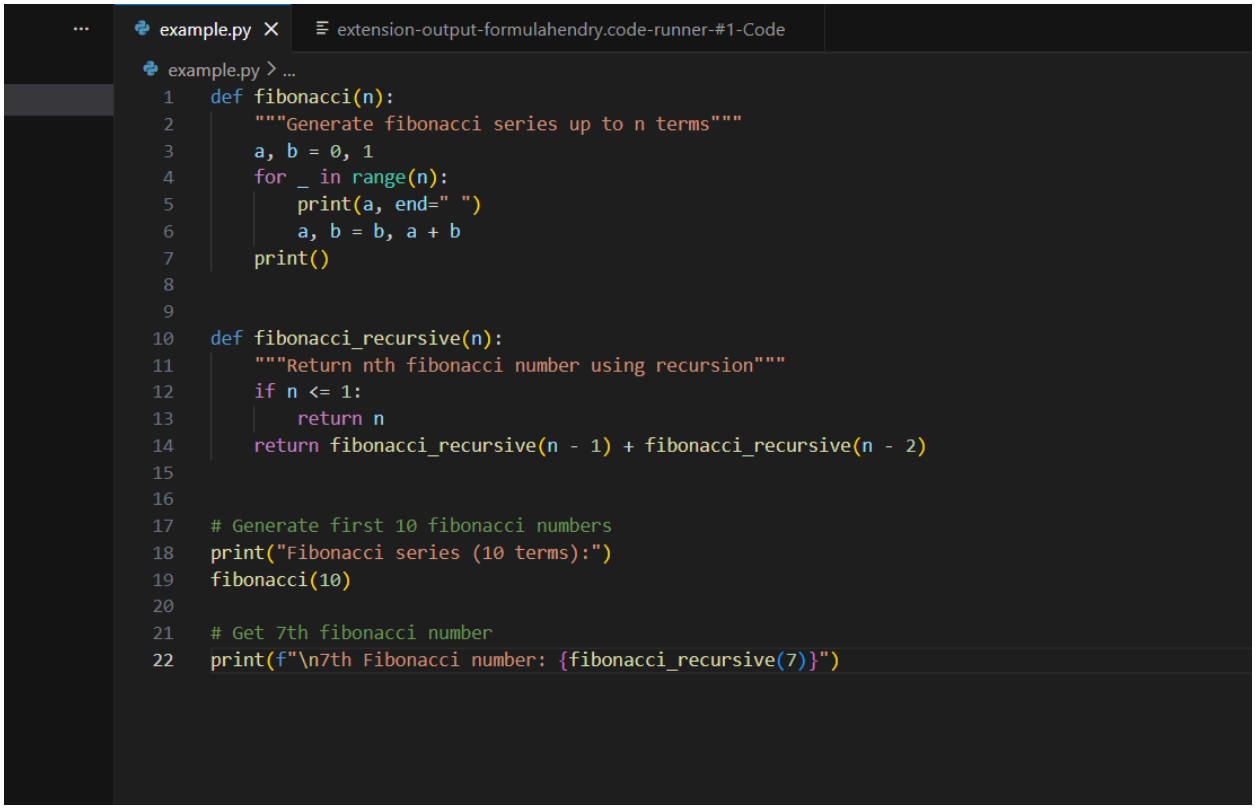


EXPLANATION:In this task, the Fibonacci code generated in Task 1 is reviewed and optimized using GitHub Copilot. Redundant variables are removed, loop logic is simplified, and unnecessary computations are avoided. The optimized code improves readability, efficiency, and maintainability while producing the same output as the original version.

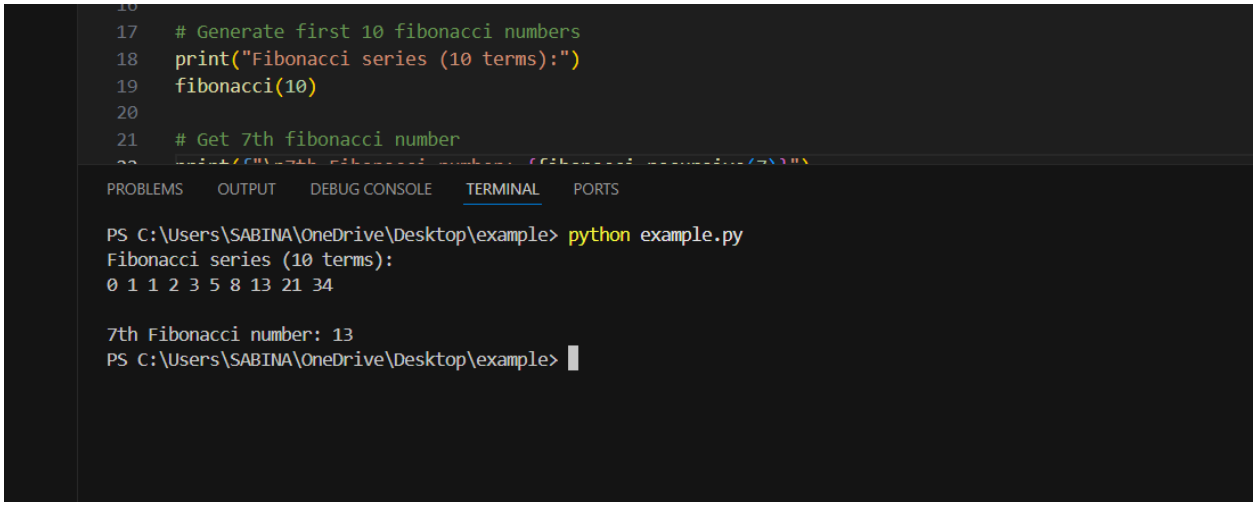
TASK-3

PROMPT-Generate a python code to print fibonacci series using functions

CODE:



OUTPUT:



EXPLANATION:This task focuses on creating a modular Fibonacci program using a user-defined function with the help of GitHub Copilot. The Fibonacci logic is encapsulated inside a function, making the code reusable and well-structured. Meaningful comments generated by Copilot improve code clarity and make it easier to understand and maintain.

TASK-4

PROMPT-generate a python code to compare Procedural vs Modular Fibonacci Code

CODE:

example.py extension-output-formulahendry.code-runner-#1-Code

example.py > ...

```python
# PROCEDURAL APPROACH
def fibonacci_procedural(n):
    """Simple procedural implementation"""
    a, b = 0, 1
    for _ in range(n):
        print(a, end=" ")
        a, b = b, a + b
    print()

# MODULAR APPROACH
def fib_generator(n):
    """Generator function for cleaner logic"""
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

def fibonacci_modular(n):
    """Modular implementation using helper function"""
    sequence = list(fib_generator(n))
    print(" ".join(map(str, sequence)))

def get_fibonacci_list(n):
    """Reusable function to get fibonacci sequence"""
    return list(fib_generator(n))

# USAGE EXAMPLES
if __name__ == "__main__":
    print("Procedural Fibonacci (n=10):")
    fibonacci_procedural(10)

    print("\nModular Fibonacci (n=10):")
    fibonacci_modular(10)

    print("\nModular - Get as list:")
    fib_list = get_fibonacci_list(10)
    print(fib_list)

    print("\nModular - Get specific number:")
    print(f"5th Fibonacci: {fib_list[5]}")
```

OUTPUT:

```python
    sequence = list(fib_generator(n))
    print(" ".join(map(str, sequence)))

def get_fibonacci_list(n):
    """Reusable function to get fibonacci sequence"""
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\SABINA\OneDrive\Desktop\example> python example.py
Procedural Fibonacci (n=10):
0 1 1 2 3 5 8 13 21 34

Modular Fibonacci (n=10):
0 1 1 2 3 5 8 13 21 34

Modular - Get as list:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Modular - Get specific number:
5th Fibonacci: 5
PS C:\Users\SABINA\OneDrive\Desktop\example>
```

EXPLANATION:This task compares the Fibonacci program written without functions and the function-based implementation. The comparison is based on code clarity, reusability, ease of debugging, and suitability for larger systems. The modular approach is found to be more reusable and maintainable, while the procedural approach is simpler for small programs.

COMPARISION TABLE:

Aspect	Procedural Approach (Without Functions)	Modular Approach (Using Functions)
Code Structure	All logic is written directly in the main program	Logic is separated into a user-defined function
Code Clarity	Simple but becomes lengthy as logic grows	Clear and well-organized
Reusability	Code cannot be reused easily	Function can be reused in multiple modules
Debugging	Difficult to debug in large programs	Easier to debug and test functions individually
Maintainability	Harder to maintain for larger applications	Easier to maintain and update
Suitability	Suitable only for small, simple programs	Suitable for large and scalable systems
Readability	Acceptable for short code	High readability due to modular design

TASK-5

PROMPT-write a python code to generate An iterative Fibonacci implementation And A recursive Fibonacci implementation

CODE:

example.py

extension-output-formulahendry.code-runner-#1-Code

example.py > ...

```python
def fibonacci_iterative(n):
    fib_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fib_sequence.append(a)
        a, b = b, a + b
    return fib_sequence

def fibonacci_recursive(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        seq = fibonacci_recursive(n - 1)
        seq.append(seq[-1] + seq[-2])
        return seq

# Example usage
n = 10
print("Iterative Fibonacci:", fibonacci_iterative(n))
print("Recursive Fibonacci:", fibonacci_recursive(n))
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\SABINA\OneDrive\Desktop\example> python example.py
Iterative Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Recursive Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\SABINA\OneDrive\Desktop\example>
```

EXPLANATION:In this task, GitHub Copilot is used to generate both iterative and recursive implementations of the Fibonacci series. The iterative approach uses

loops and is more efficient for large inputs, while the recursive approach uses function calls and is easier to understand conceptually. However, recursion has higher time and space complexity and should be avoided for large values of n.

COMPARISION TABLE:

Aspect	Iterative Fibonacci	Recursive Fibonacci
Implementation Method	Uses loops (for/while)	Uses function calls
Execution Flow	Executes sequentially	Function calls itself repeatedly
Time Complexity	O(n)	$O(2^n)$
Space Complexity	O(1)	O(n) due to call stack
Performance	Faster and efficient for large n	Slow for large n
Memory Usage	Low memory usage	High memory usage
Ease of Understanding	Slightly more logical thinking required	Conceptually simple
Suitability	Best for real-world and large inputs	Suitable only for small inputs
When to Avoid	Rarely needs to be avoided	Should be avoided for large n