School of Computer Science and Artificial Intelligence

Lab Assignment-01

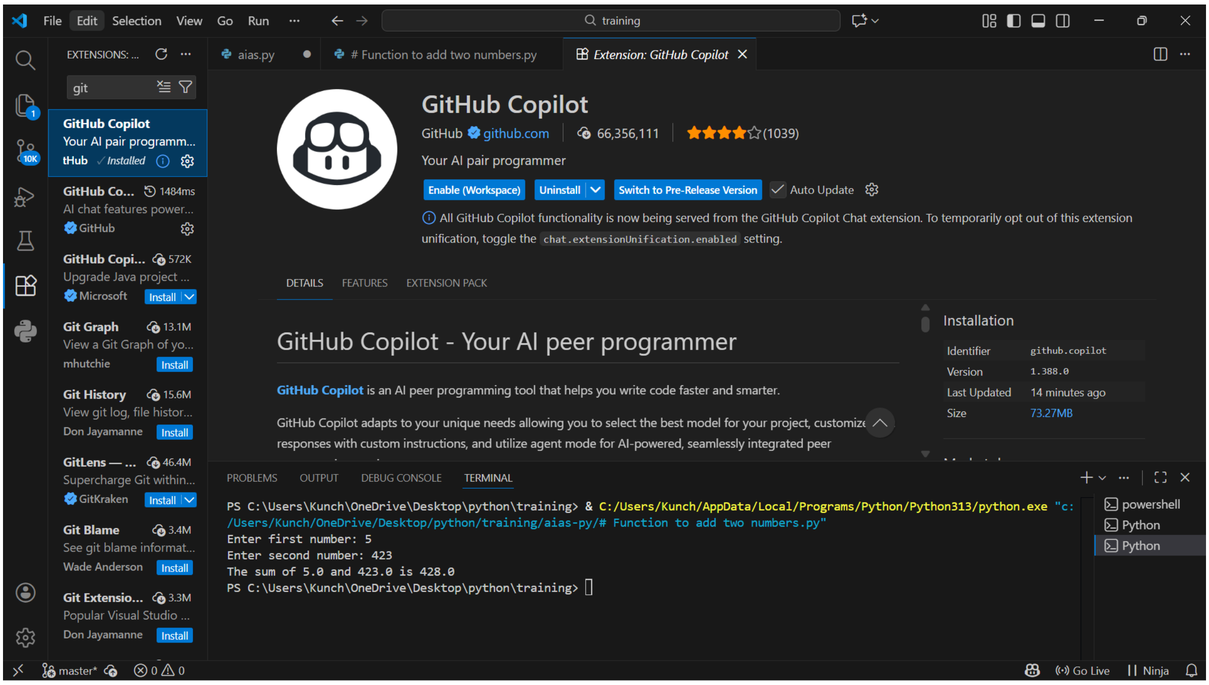
Course Name : AI Assistant Coding
Name of the Student : Shaik Naved Ahmed
Registration No : 2303A54053
Batch No : 47-B

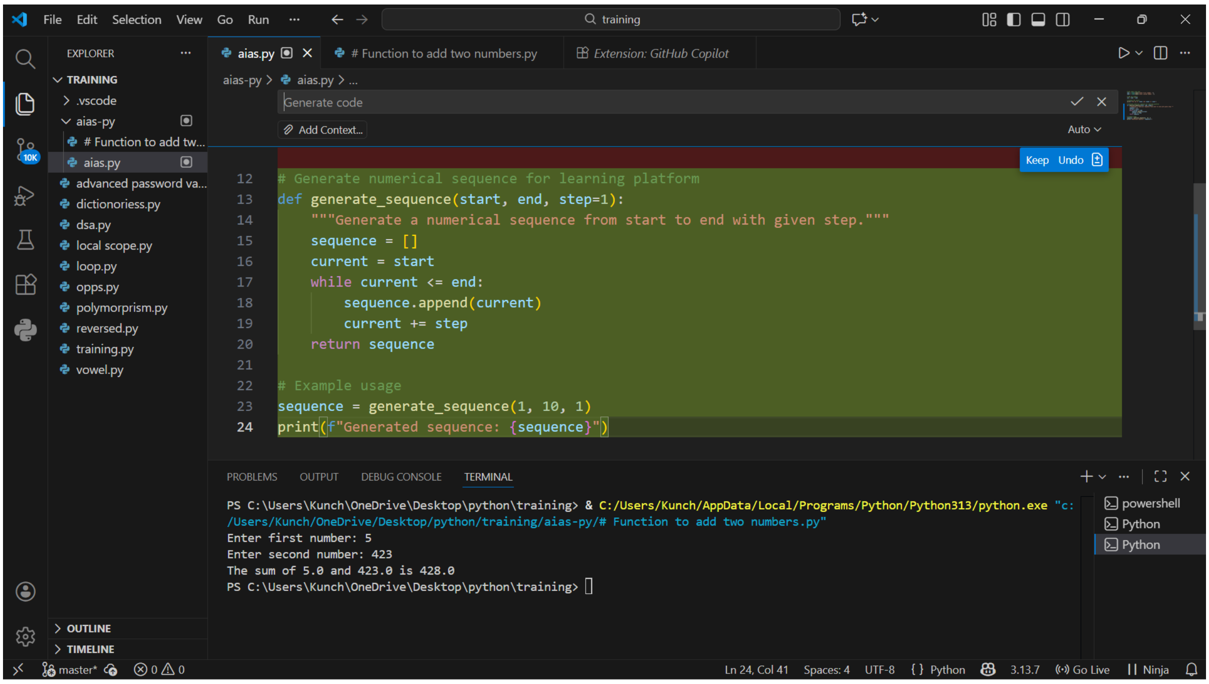
Task 0

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

Expected Output

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





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

Prompt:

Write 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, and does not use any user-defined functions

Code :

```
# 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
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
a, b = 0, 1
count = 0
print("Fibonacci sequence:")
while count < n:
print(a, end=' ')
a, b = b, a + b
count += 1
print() # for newline after the sequence
```


```
# 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
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
a, b = 0, 1
count = 0
print("Fibonacci sequence:")
while count < n:
print(a, end=' ')
a, b = b, a + b
count += 1
print() # for newline after the sequence
```


Output:

Enter the number of terms in the Fibonacci sequence: 10

Fibonacci sequence:

0 1 1 2 3 5 8 13 21 34

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

Improved code :

```
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
a, b = 0, 1
fib_sequence = []
for _ in range(n):
fib_sequence.append(str(a))
a, b = b, a + b
print("Fibonacci sequence: " + " ".join(fib_sequence))
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
fib_sequence = []
for _ in range(n):
fib_sequence.append(str(a))
a, b = b, a + b
print("Fibonacci sequence: " + " ".join(fib_sequence))
```

Output :

Enter the number of terms in the Fibonacci sequence: 7

Fibonacci sequence: 0 1 1 2 3 5 8

Explanation:

The original code was inefficient because it repeated the input and loop logic and failed to reinitialize variables, which could cause incorrect results. This duplication made the program harder to read and maintain. The optimized version uses a single input and loop, properly initializes variables, and removes redundant code. This improves both performance and readability by keeping the logic clear and concise.

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

Prompt :

```
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
```

Code :

```
def generate_fibonacci(n):
"""Generate Fibonacci sequence up to n terms."""
a, b = 0, 1
fib_sequence = []
for _ in range(n):
fib_sequence.append(str(a))
a, b = b, a + b
return fib_sequence
# Accept user input for number of terms
n = int(input("Enter the number of terms in the Fibonacci sequence: "))
# Generate and print the Fibonacci sequence
fib_sequence = generate_fibonacci(n)
print("Fibonacci sequence: " + " ".join(fib_sequence))
```

Output:

Enter the number of terms in the Fibonacci sequence: 6

Fibonacci sequence: 0 1 1 2 3 5

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

Report :

Code Clarity: Using user-defined functions makes the code more structured and easier to understand by clearly separating logic into meaningful blocks.

Reusability: Functions allow the same Fibonacci logic to be reused multiple times without rewriting code, saving time and reducing errors.

Debugging Ease: Errors are easier to locate and fix because each function can be tested and debugged independently.

Suitability for Larger Systems: Function-based code scales well in larger programs, as it improves organization, maintenance, and integration with other modules.

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:

Iterative Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Recursive Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Explanation of Execution Flow

Iterative Fibonacci Implementation

The iterative function initializes two variables (a and b) to store consecutive Fibonacci numbers. A loop runs n times, appending the current value of a to the list and updating the values using tuple assignment. The sequence is built step by step without function calls.

Recursive Fibonacci Implementation

The recursive function handles base cases ($n \leq 2$) directly. For larger values, it calls itself to compute the sequence up to n-1 and then appends the next Fibonacci number by adding the last two values. This process continues until the base case is reached.

Comparison

Time Complexity:
Both implementations run in **O(n)** time since each Fibonacci number is computed once.

Space Complexity:
The iterative approach uses **O(1)** extra space (excluding the output list), while the recursive approach uses **O(n)** space due to the call stack.

Performance for Large n:
The iterative approach performs better for large values of n because it avoids function call overhead and excessive memory usage.

When Recursion Should Be Avoided:
Recursion should be avoided for large inputs as it can cause stack overflow, increased memory consumption, and slower performance compared to iteration.