AI Assisted Coding

M. Sprusheeth Rao

2303A51206

Batch - 04

07-01-2026

Lab 1: Environment Setup – GitHub Copilot and VS Code Integration +

Understanding AI-assisted Coding Workflow

Lab Objectives:
● To install and configure GitHub Copilot in Visual Studio Code.

● To explore AI-assisted code generation using GitHub Copilot.

● To analyze the accuracy and effectiveness of Copilot's code

suggestions.

● To understand prompt-based programming using comments and

code context

Lab Outcomes (LOs):

After completing this lab, students will be able to:

● Set up GitHub Copilot in VS Code successfully.

● Use inline comments and context to generate code with Copilot.

● Evaluate AI-generated code for correctness and readability.

● Compare code suggestions based on different prompts and
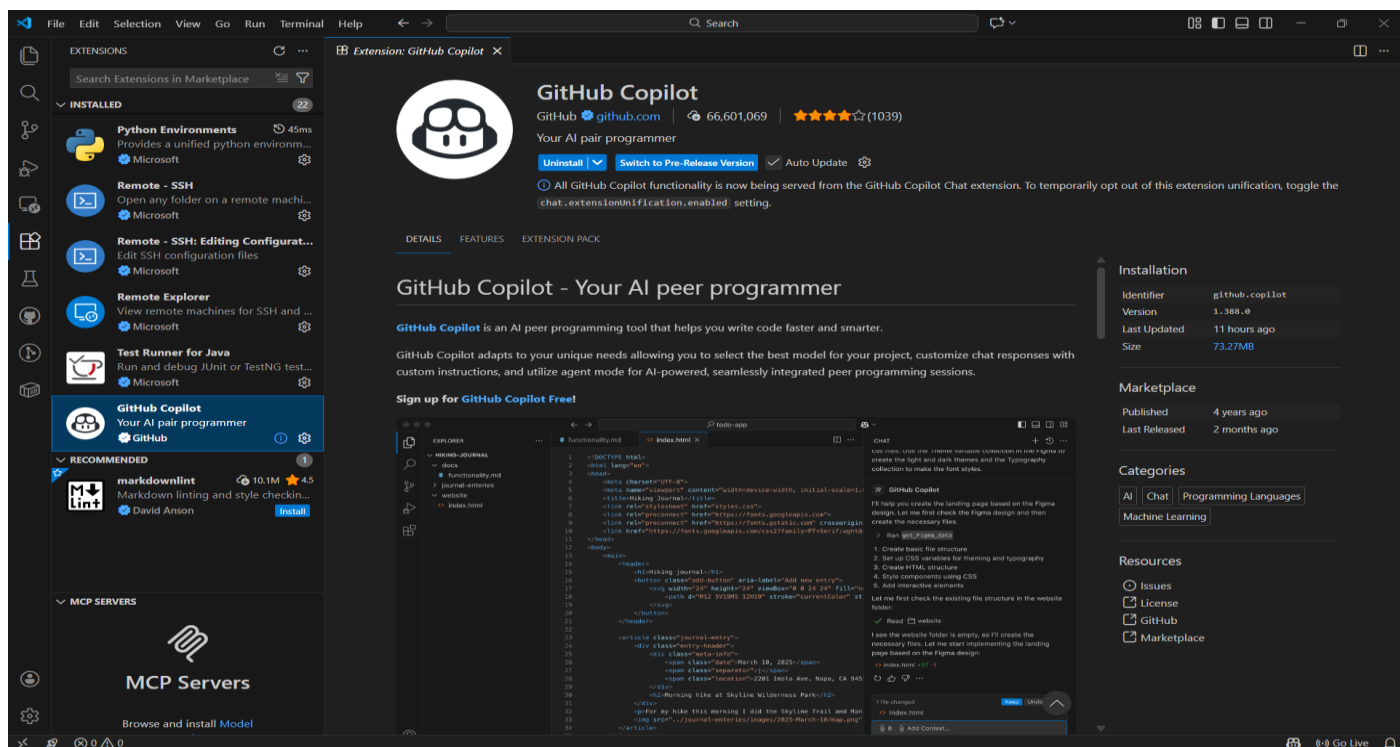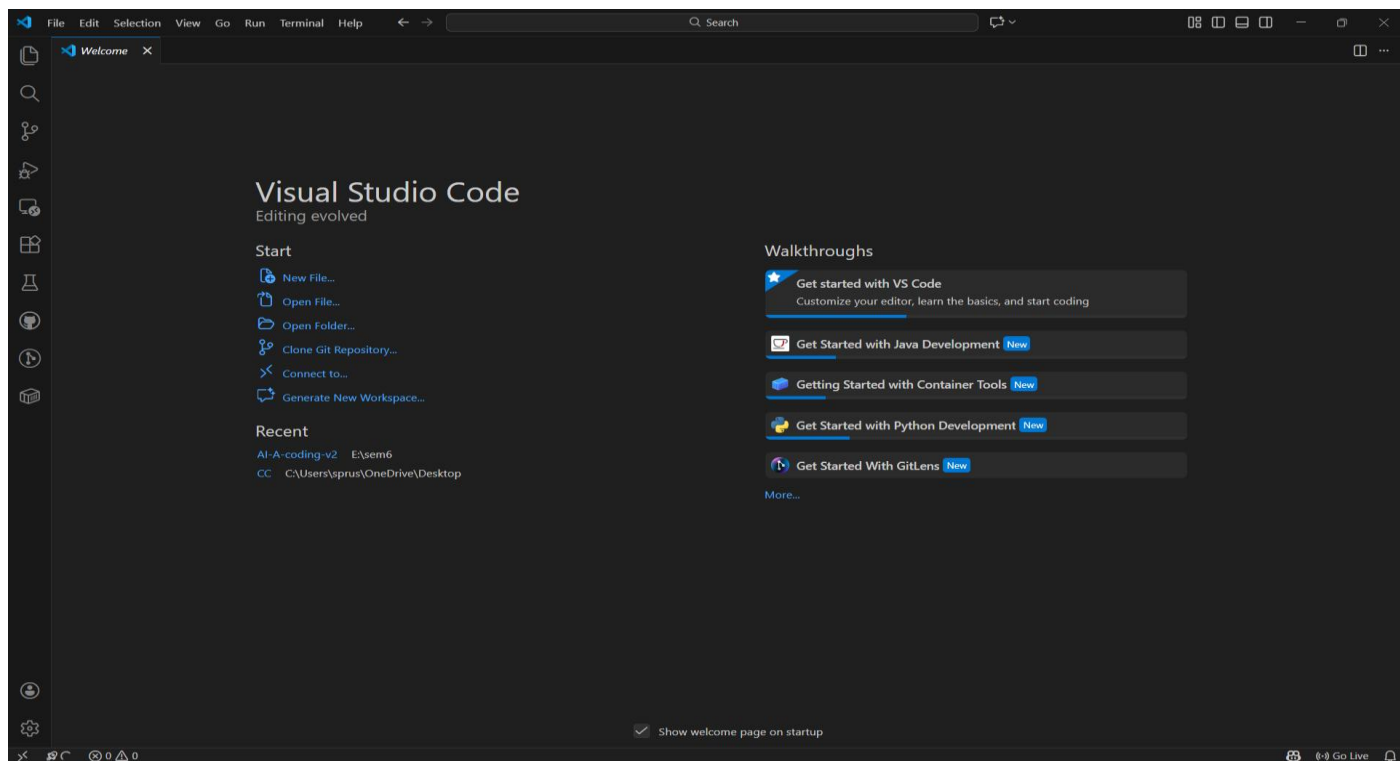
programming styles.

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 0: Environment Setup:-

Task 1: AI-Generated Logic Without Modularization (Factorial without Functions)

• Scenario

You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

• Task Description

Use GitHub Copilot to generate a Python program that computes a
mathematical product-based value (factorial-like logic) directly in the main
execution flow, without using any user-defined functions.

• Constraint:

➢ Do not define any custom function
➢ Logic must be implemented using loops and variables only

• Expected Deliverables

➢ A working Python program generated with Copilot assistance

➢ Screenshot(s) showing:

➢ The prompt you typed

➢ Copilot's suggestions

➢ Sample input/output screenshots

➢ Brief reflection (5–6 lines):

➢ How helpful was Copilot for a beginner?

➢ Did it follow best practices automatically?

Task 1: AI-Generated Logic Without Modularization (Factorial without
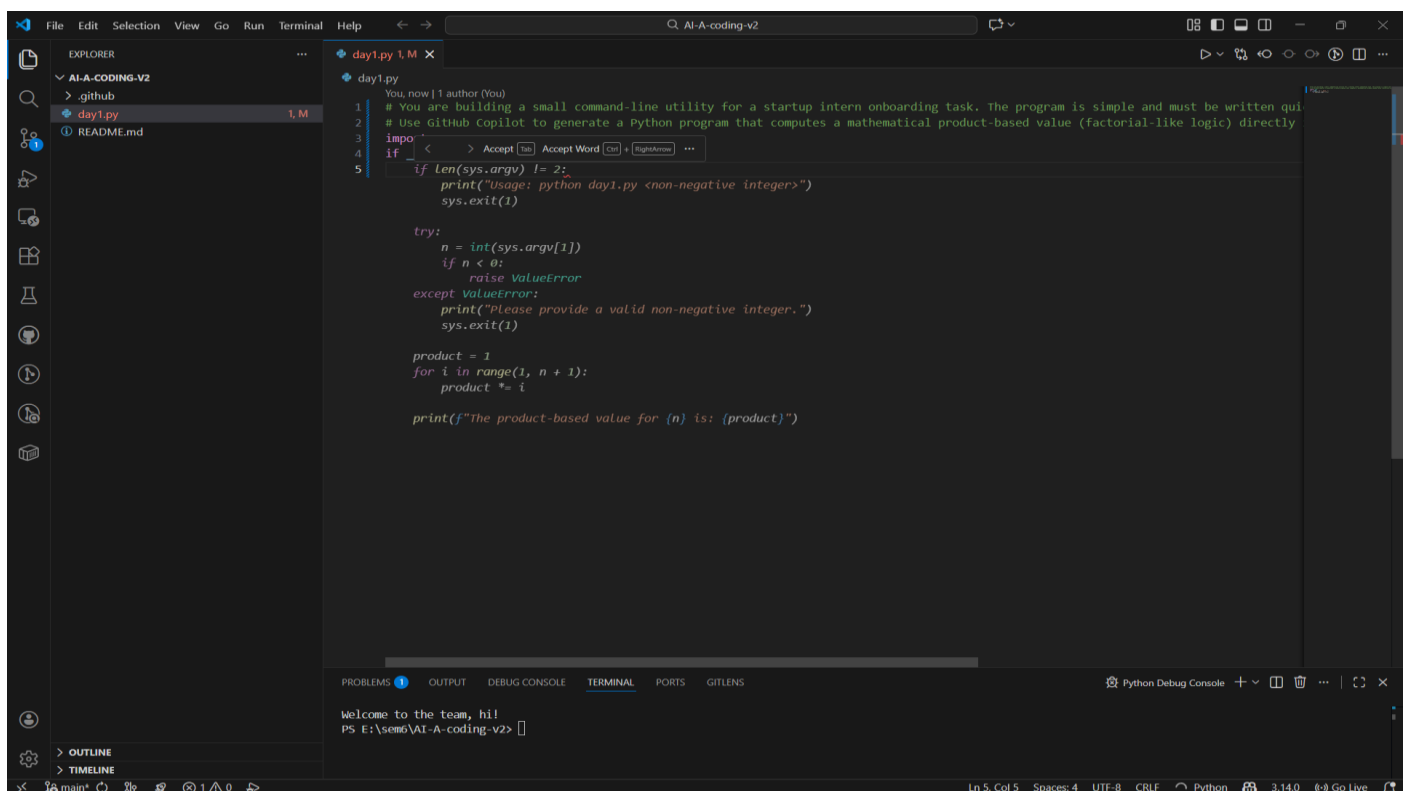
Functions)

```python
# You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.
# Use GitHub Copilot to generate a Python program that computes a mathematical product-based value (factorial-like logic) directly in the main execution flow, without usi
import sys
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python day1.py <non-negative integer>")
        sys.exit(1)

    try:
        n = int(sys.argv[1])
        if n < 0:
            raise ValueError
    except ValueError:
        print("Please provide a valid non-negative integer.")
        sys.exit(1)

    product = 1
    for i in range(1, n + 1):
        product *= i
    print(f"The product-based value for {n} is: {product}")
```

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

❖ Scenario

Your team lead asks you to review AI-generated code before committing it to

a shared repository.

❖ Task Description

Analyze the code generated in Task 1 and use Copilot again to:

➢ Reduce unnecessary variables

➢ Improve loop clarity

➢ Enhance readability and efficiency

Hint:

Prompt Copilot with phrases like

"optimize this code", "simplify logic", or "make it more readable"
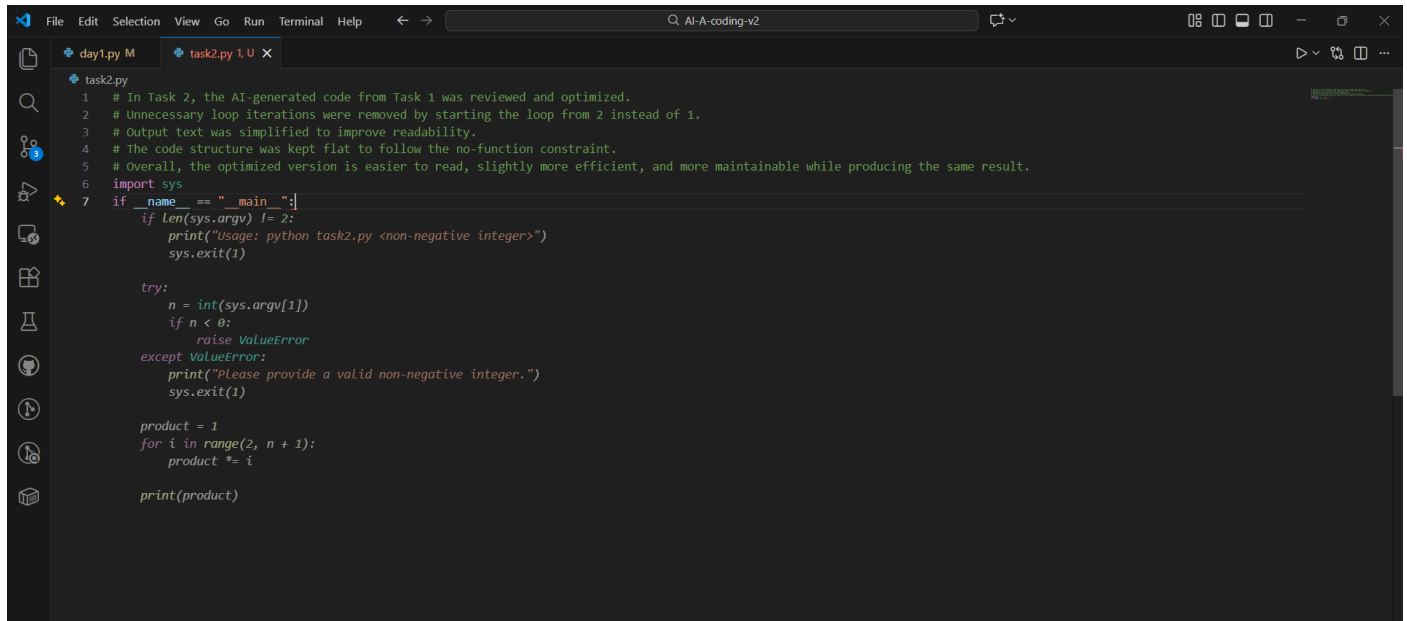
❖ Expected Deliverables

➢ Original AI-generated code

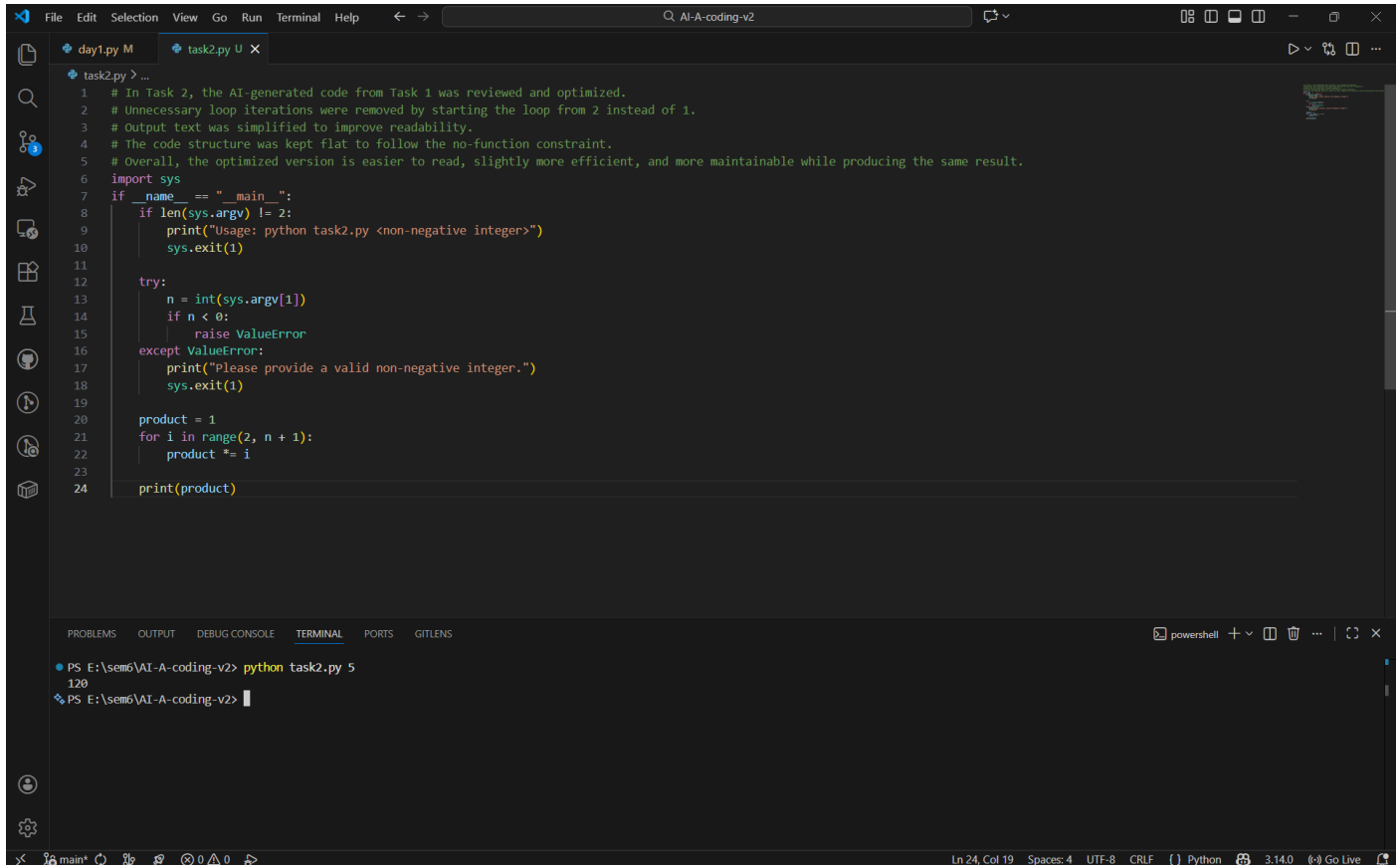➢ Optimized version of the same code

➢ Side-by-side comparison

➢ Written explanation:

▪ What was improved?

▪ Why the new version is better (readability, performance,

maintainability.

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


```python
# In Task 2, the AI-generated code from Task 1 was reviewed and optimized.
# Unnecessary loop iterations were removed by starting the loop from 2 instead of 1.
# Output text was simplified to improve readability.
# The code structure was kept flat to follow the no-function constraint.
# Overall, the optimized version is easier to read, slightly more efficient, and more maintainable while producing the same result.
import sys
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python task2.py <non-negative integer>")
        sys.exit(1)

    try:
        n = int(sys.argv[1])
        if n < 0:
            raise ValueError
    except ValueError:
        print("Please provide a valid non-negative integer.")
        sys.exit(1)

    product = 1
    for i in range(2, n + 1):
        product *= i

    print(product)
```



```
PS E:\sem6\AI-A-coding-v2> python task2.py 5
120
PS E:\sem6\AI-A-coding-v2>
```

Task 3: Modular Design Using AI Assistance (Factorial with Functions)

❖ **Scenario**

The same logic now needs to be reused in multiple scripts.

❖ **Task Description**

Use GitHub Copilot to generate a modular version of the program by:

➢ **Creating a user-defined function**

➢ **Calling the function from the main block**

❖ **Constraints**

Task 3: Modular Design Using AI Assistance (Factorial with Functions)

➢ **Use meaningful function and variable names**

➢ **Include inline comments (preferably suggested by Copilot)**

❖ **Expected Deliverables**

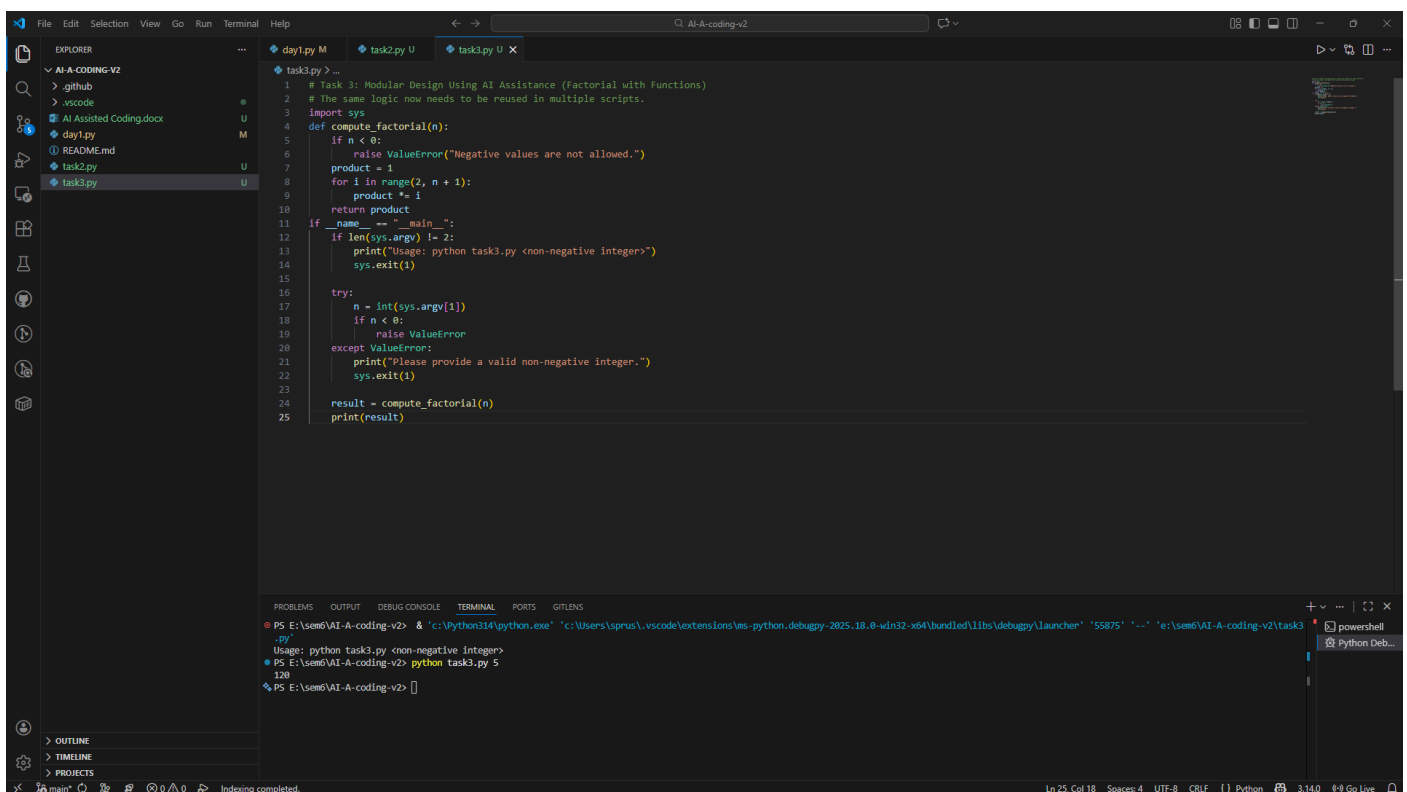➢ **AI-assisted function-based program**

➢ **Screenshots showing:**

o **Prompt evolution**

o **Copilot-generated function logic**

➢ **Sample inputs/outputs**

➢ **Short note:**

o **How modularity improves reusability.**


```python
# Task 3: Modular Design Using AI Assistance (Factorial with Functions)
# The same logic now needs to be reused in multiple scripts.
import sys
def compute_factorial(n):
    if n < 0:
        raise ValueError("Negative values are not allowed.")
    product = 1
    for i in range(2, n + 1):
        product *= i
    return product
if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python task3.py <non-negative integer>")
        sys.exit(1)

    try:
        n = int(sys.argv[1])
        if n < 0:
            raise ValueError
    except ValueError:
        print("Please provide a valid non-negative integer.")
        sys.exit(1)

    result = compute_factorial(n)
    print(result)
```

Short Note: How Modularity Improves Reusability

Modularity helps in reusability by helping separate logic in terms of different functions which may be reused in multiple programs. The factorial computation is put in a function which makes the code easier to maintain and test. If the logic has to be changed, changes can be made at one place without having any impact on the whole program. Modular code is also more readable and easier to work on in a team environment.

Task 4: Comparative Analysis – Procedural vs Modular AI Code (With vs

Without Functions)

❖ **Scenario**

As part of a code review meeting, you are asked to justify design choices.

❖ **Task Description**

Compare the non-function and function-based Copilot-generated

programs on the following criteria:

➢ **Logic clarity**

➢ **Reusability**

➢ **Debugging ease**

➢ **Suitability for large projects**

➢ **AI dependency risk**

❖ **Expected Deliverables**

Choose one:

➢ **A comparison table**

OR

➢ **A short technical report (300–400 words).**

Task 4: Comparative Analysis – Procedural vs Modular AI Code (With vs Without Functions)

Criteria	Procedural Code (Without Functions)	Modular Code (With Functions)
Logic Clarity	Logic is written in one continuous flow, which is easy to follow for very small programs but becomes cluttered as code grows.	Logic is clearly separated into functions, making the purpose of each part easier to understand.
Reusability	Code cannot be reused easily because the logic is tightly coupled to the main execution block.	Function-based logic can be reused across multiple scripts by importing or calling the function.
Debugging Ease	Debugging is harder since all logic exists in one block, making it difficult to isolate issues.	Debugging is easier because errors can be traced to specific functions.
Suitability for Large Projects	Not suitable for large projects as it leads to poor organization and low maintainability.	Well suited for large projects due to better structure, scalability, and teamwork support.
AI Dependency Risk	High risk, as beginners may copy AI-generated code without understanding the full flow.	Lower risk, as modular structure encourages understanding of individual components.

Task 5: AI-Generated Iterative vs Recursive Thinking

❖ Scenario

Your mentor wants to test how well AI understands different

computational paradigms.

❖ Task Description

Prompt Copilot to generate:

An iterative version of the logic

A recursive version of the same logic

❖ Constraints

Both implementations must produce identical outputs

Students must not manually write the code first

❖ Expected Deliverables

Two AI-generated implementations

Execution flow explanation (in your own words)

Comparison covering:

➢ Readability

➢ Stack usage

➢ Performance implications

➢ When recursion is not recommended.

Task 5: AI-Generated Iterative vs Recursive Thinking

Iterative Thinking -

Recursive thinking –

Aspect	Iterative Approach	Recursive Approach
Readability	Easy to understand for beginners	More mathematical and elegant
Stack Usage	Uses constant memory	Uses additional stack memory
Performance	Faster and memory efficient	Slower for large inputs
Error Risk	Low	Risk of stack overflow
When Not Recommended	—	Not recommended for large input values