

AI ASSISTANT CODING ASSIGNMENT-1.2

Name: A.Sathwik

Enrollment no: 2303A5158

Batch: 34

Semester: VI

Branch: CSE(AIML)

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

Lab Objectives:

Week1 -

Monday

- 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.

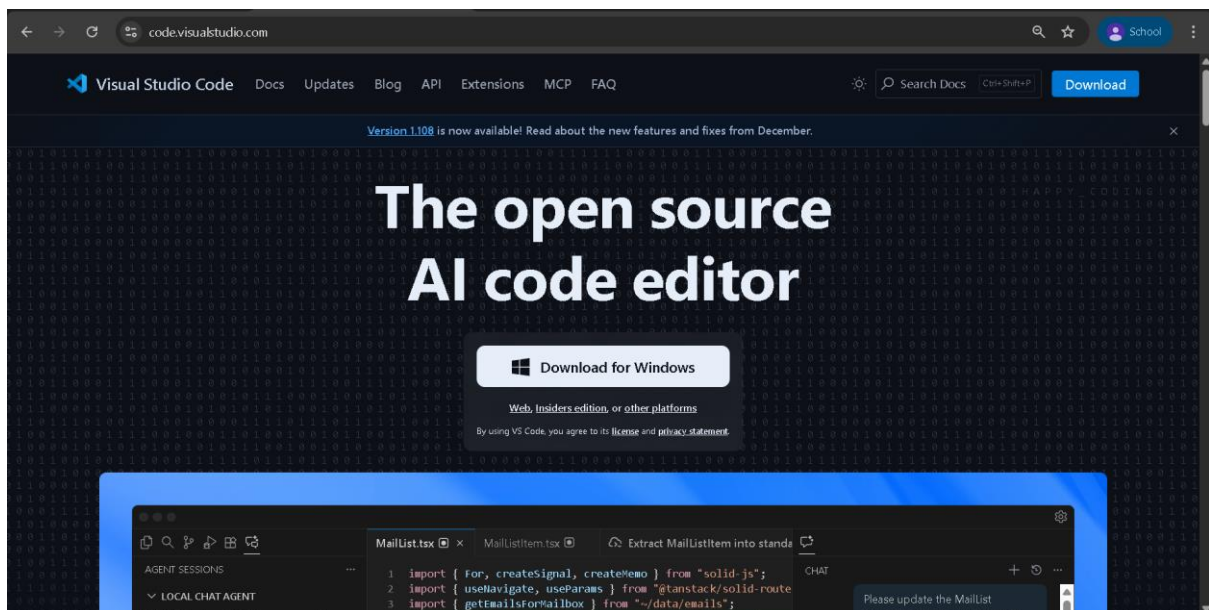
- Evaluate AI-generated code for correctness and readability.
- Compare code suggestions based on different prompts and programming styles.

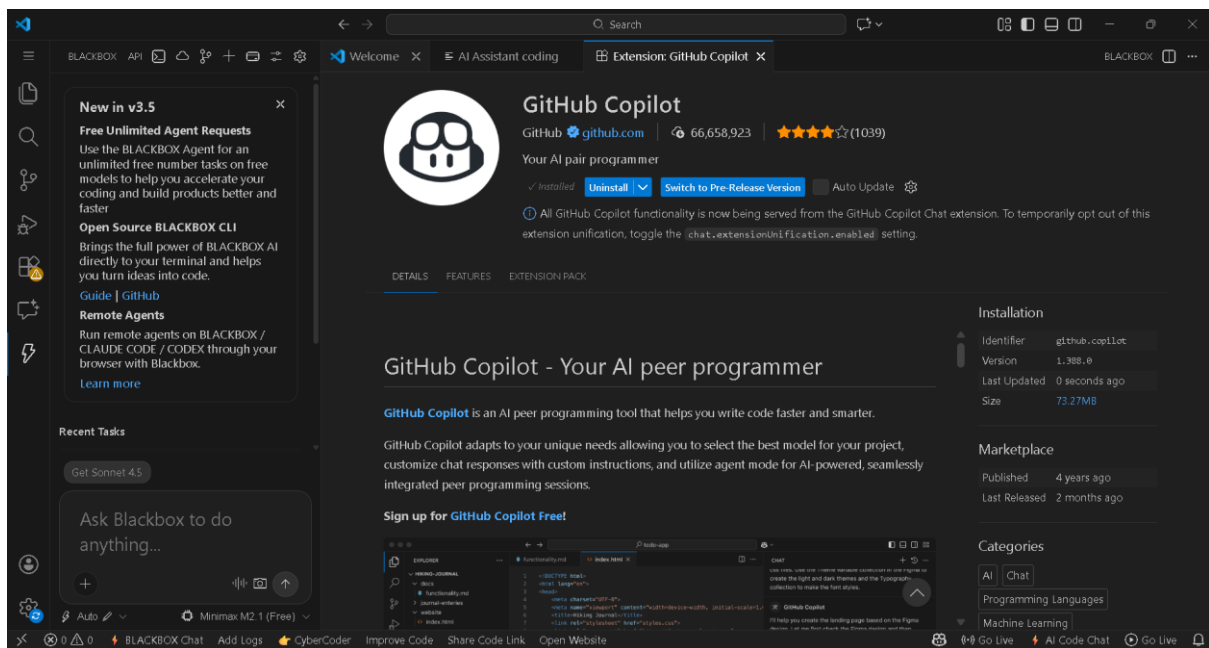
Task 0:

- 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 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.

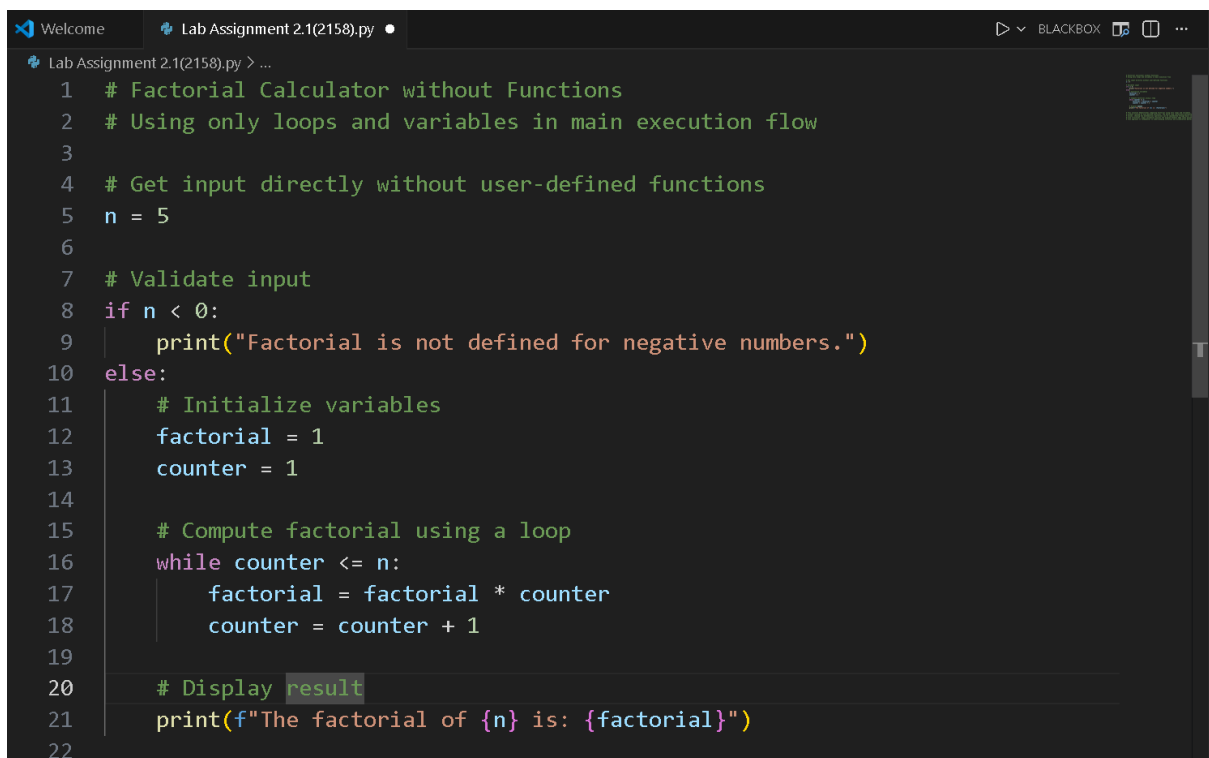
- Constraint:

- 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?



```
1 # Factorial Calculator without Functions
2 # Using only loops and variables in main execution flow
3
4 # Get input directly without user-defined functions
5 n = 5
6
7 # Validate input
8 if n < 0:
9     print("Factorial is not defined for negative numbers.")
10 else:
11     # Initialize variables
12     factorial = 1
13     counter = 1
14
15     # Compute factorial using a loop
16     while counter <= n:
17         factorial = factorial * counter
18         counter = counter + 1
19
20 # Display result
21 print(f"The factorial of {n} is: {factorial}")
22
```

Brief Reflection (5–6 Lines)

This solution demonstrates computing factorial using only loops and variables, without relying on user-defined functions. The while loop iteratively multiplies values, showing how mathematical operations can be implemented through basic control flow structures. Input validation ensures the program handles edge cases gracefully. This approach is fundamental to understanding iteration and accumulation patterns.

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

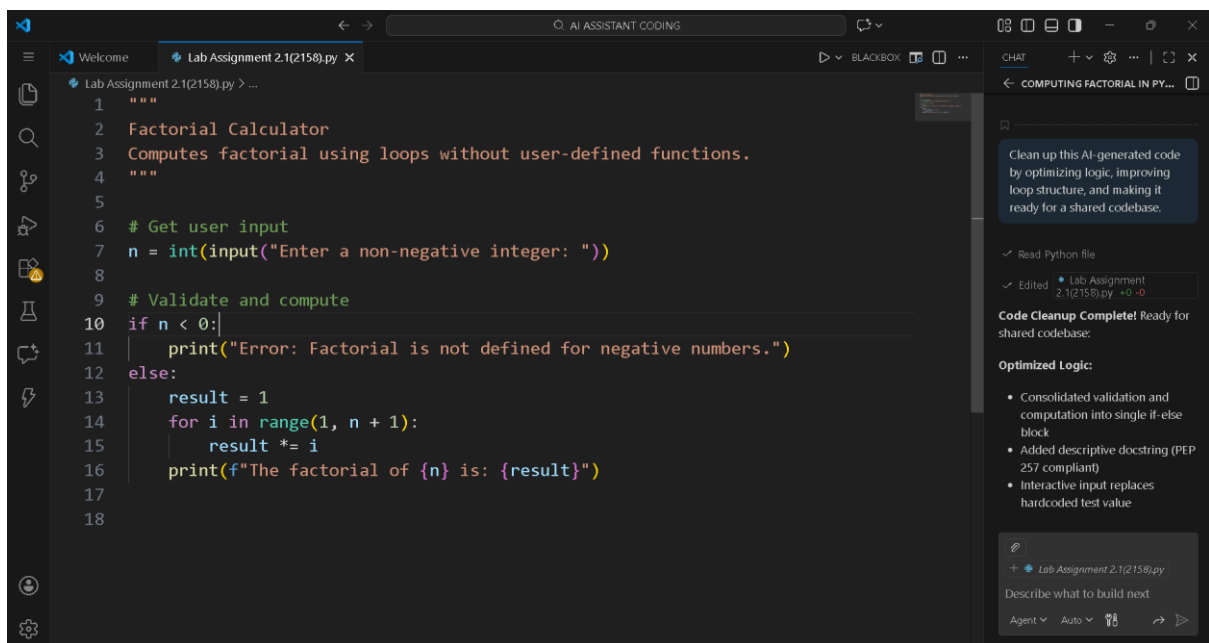
Hint:

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?

- 1) Removal of Unnecessary Variables
- 2) Improved Loop Clarity
- 3) Cleaner and More Concise Code
- 4) Use of Clean Coding Practices

Why the New Version Is Better?

- **Readability:**
The optimized version is easier to read and understand due to fewer variables and a clearer loop structure.
- **Performance:**
Although the time complexity remains the same, removing unnecessary operations slightly improves execution efficiency.
- **Maintainability:**
Simpler code with fewer components is easier to debug, modify, and maintain in a collaborative environment.

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

- Use meaningful function and variable names
- Include inline comments (preferably suggested by Copilot)

❖ Expected Deliverables

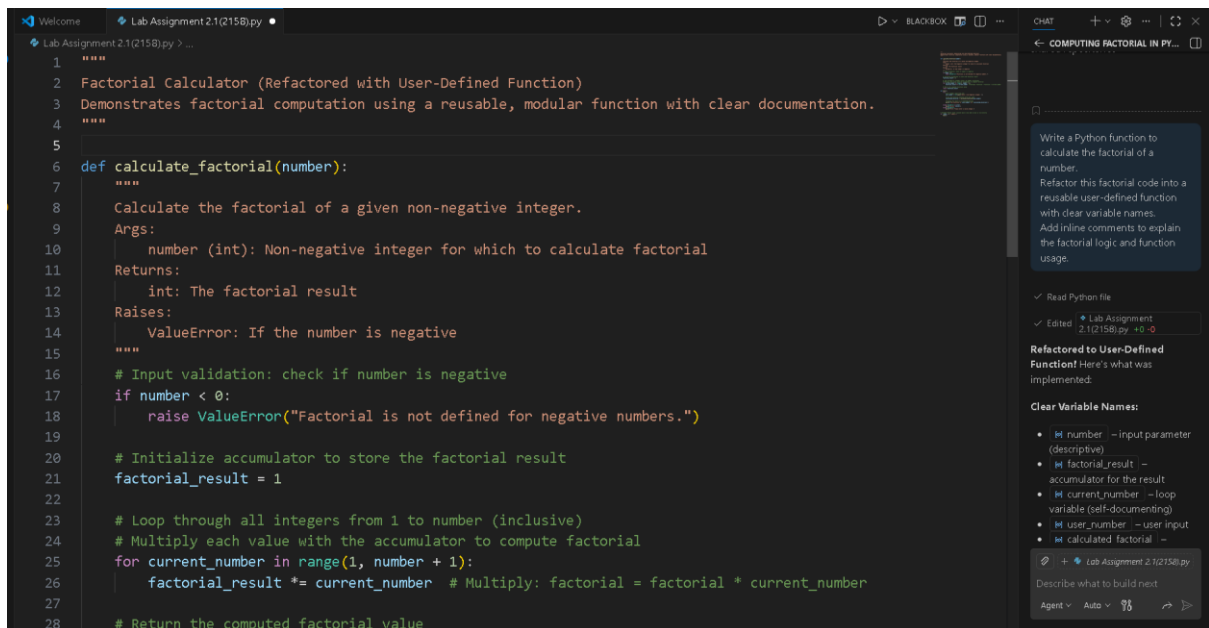
➤ AI-assisted function-based program

➤ Screenshots showing:

o Prompt evolution

o Copilot-generated function logic

➤ Sample inputs/outputs



```
1 """
2 Factorial Calculator (Refactored with User-Defined Function)
3 Demonstrates factorial computation using a reusable, modular function with clear documentation.
4 """
5
6 def calculate_factorial(number):
7     """
8     Calculate the factorial of a given non-negative integer.
9     Args:
10         number (int): Non-negative integer for which to calculate factorial
11     Returns:
12         int: The factorial result
13     Raises:
14         ValueError: If the number is negative
15     """
16     # Input validation: check if number is negative
17     if number < 0:
18         raise ValueError("Factorial is not defined for negative numbers.")
19
20     # Initialize accumulator to store the factorial result
21     factorial_result = 1
22
23     # Loop through all integers from 1 to number (inclusive)
24     # Multiply each value with the accumulator to compute factorial
25     for current_number in range(1, number + 1):
26         factorial_result *= current_number # Multiply: factorial = factorial * current_number
27
28     # Return the computed factorial value
```

Write a Python function to calculate the factorial of a number. Refactor this factorial code into a reusable user-defined function with clear variable names. Add inline comments to explain the factorial logic and function usage.

✓ Read Python file
✓ Edited Lab Assignment 2.1(2158).py +0-0

Refactored to User-Defined Function! Here's what was implemented:

Clear Variable Names:

- `number` - input parameter (descriptive)
- `factorial_result` - accumulator for the result
- `current_number` - loop variable (self-documenting)
- `user_number` - user input
- `calculated_factorial` -

Describe what to build next

Agent Auto 95



```
6 def calculate_factorial(number):
7     """
8     Calculate the factorial of a given non-negative integer.
9     Args:
10         number (int): Non-negative integer for which to calculate factorial
11     Returns:
12         int: The factorial result
13     Raises:
14         ValueError: If the number is negative
15     """
16     # Input validation: check if number is negative
17     if number < 0:
18         raise ValueError("Factorial is not defined for negative numbers.")
19
20     # Initialize accumulator to store the factorial result
21     factorial_result = 1
22
23     # Loop through all integers from 1 to number (inclusive)
24     # Multiply each value with the accumulator to compute factorial
25     for current_number in range(1, number + 1):
26         factorial_result *= current_number # Multiply: factorial = factorial * current_number
27
28     # Return the computed factorial value
29     return factorial_result
30
31 def main():
32     try:
33         # Get integer input from user
34         user_number = int(input("Enter a non-negative integer: "))
35
36         # Call the factorial function with user input
37         calculated_factorial = calculate_factorial(user_number)
38
39         # Display the result in a formatted message
40         print(f"The factorial of {user_number} is: {calculated_factorial}")
41
42     except ValueError as error:
43         print(f"Error: {error}")
44     except ValueError:
45         print("Error: Please enter a valid integer.")
46
47 # Standard Python idiom: execute main() only when script is run directly
48 if __name__ == "__main__":
49     main()
50
51
```

Write a Python function to calculate the factorial of a number. Refactor this factorial code into a reusable user-defined function with clear variable names. Add inline comments to explain the factorial logic and function usage.

✓ Read Python file
✓ Edited Lab Assignment 2.1(2158).py +0-0

Refactored to User-Defined Function! Here's what was implemented:

Clear Variable Names:

- `number` - input parameter (descriptive)
- `factorial_result` - accumulator for the result
- `current_number` - loop variable (self-documenting)
- `user_number` - user input
- `calculated_factorial` -

Describe what to build next

Agent Auto 95

PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT CODING> & C:\Users\vip\AppData\Local\Programs\Python\Python313\python.exe "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT CODING\Lab Assignment 2.1(2158).py"

Enter a non-negative integer: 4

Error: Factorial is not defined for negative numbers.

PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT CODING> & C:\Users\vip\AppData\Local\Programs\Python\Python313\python.exe "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT CODING\Lab Assignment 2.1(2158).py"

Enter a non-negative integer: 6

The factorial of 6 is: 720

PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT CODING>

Short note:

How modularity improves reusability:

Modularity improves reusability by placing the factorial logic inside the `calculate_factorial()` function, allowing it to be reused in multiple programs without rewriting code. The separation of logic and input/output makes the program easier to maintain, test, and update.

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
- A short technical report (300–400 words).

Introduction

This report compares procedural (non-function) AI-generated code with modular, function-based code using a factorial program as reference, and briefly discusses iterative versus recursive AI approaches.

Logic Clarity

Procedural code keeps all logic in a single flow, which may work for small programs but becomes harder to read as complexity increases. Modular code improves clarity by separating the factorial logic into a well-defined function, making the program easier to understand and review.

Reusability

Procedural code has low reusability because the logic is tightly bound to one script. Modular code is more reusable, as the factorial function can be called from multiple programs or reused in larger systems without duplication.

Debugging Ease

Debugging procedural code often requires checking the entire script. In modular code, errors are easier to locate because they are confined to specific functions, allowing faster testing and fixes.

Suitability for Large Projects

Modular code is better suited for large projects because it supports scalability, maintainability, and team collaboration. Procedural code does not scale well and can lead to tightly coupled logic and technical debt.

AI Dependency Risk and Iterative vs Recursive Thinking

Procedural AI-generated code may increase blind reliance on AI output. Modular code encourages human understanding and review. Iterative AI solutions are generally more efficient and safer than recursive ones, which can be harder to debug and may cause stack overflow issues.

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.

The screenshot displays a code editor with a Python script for calculating factorials iteratively. The script includes comments and input validation. The sidebar on the right lists constraints for the code generation. The terminal window at the bottom shows the execution of the script.

```
1 #!/usr/bin/env python
2 """Iterative factorial demo suitable for an academic lab.
3
4 This script computes the factorial of a non-negative integer using a loop-based
5 approach. It includes clear inline comments and input validation.
6 """
7
8 import sys
9
10 def factorial_iterative(n: int) -> int:
11     """
12     Returns:
13         - factorial value as int
14
15     Raises:
16         - TypeError if n is not an int
17         - ValueError if n is negative
18     """
19     # Ensure the input is an integer type
20     if not isinstance(n, int):
21         raise TypeError("Input must be an integer")
22
23     # Negative numbers do not have factorials in this context
24     if n < 0:
25         raise ValueError("Input must be a non-negative integer")
26
27     # Start accumulator at 1 because 0! == 1
28     result = 1
29
30     # Multiply result by each integer from 2 up to n (loop-based approach)
31     for i in range(2, n + 1):
32         result *= i
33
34     # Return the computed factorial
35     return result
36
37 def parse_non_negative_int(s: str) -> int:
38     """Parse a string into a non-negative integer; raise ValueError on invalid input."""
39
40     try:
41         n = int(s)
42         if n < 0:
43             raise ValueError("Input must be a non-negative integer")
44         return n
45     except ValueError:
46         raise ValueError("Input must be a non-negative integer")
```

Constraints:

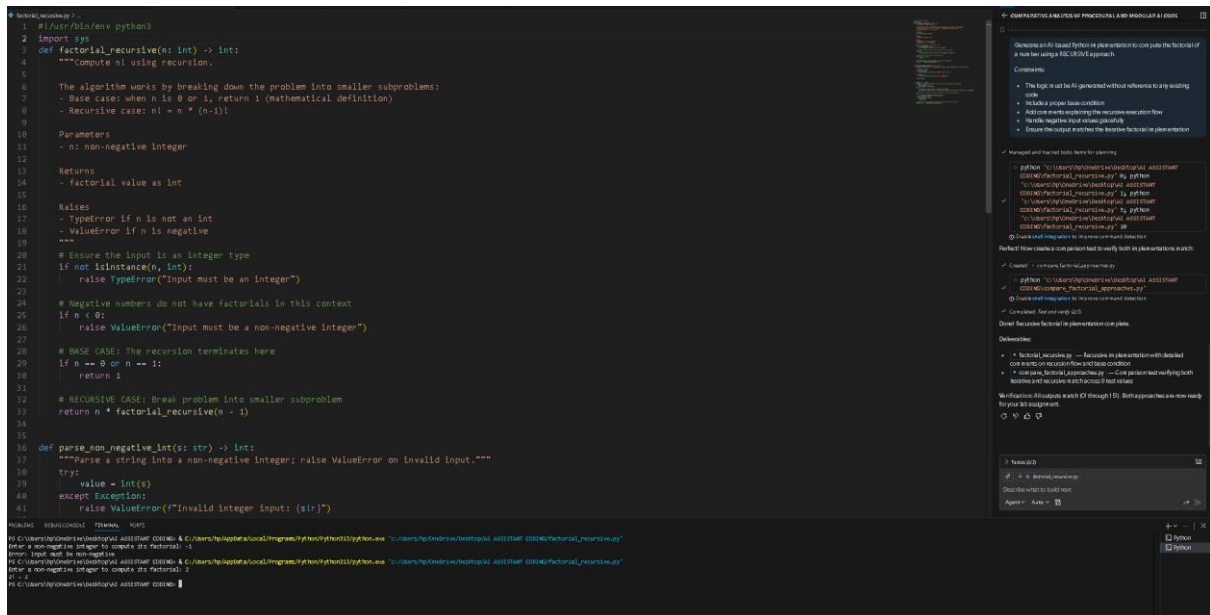
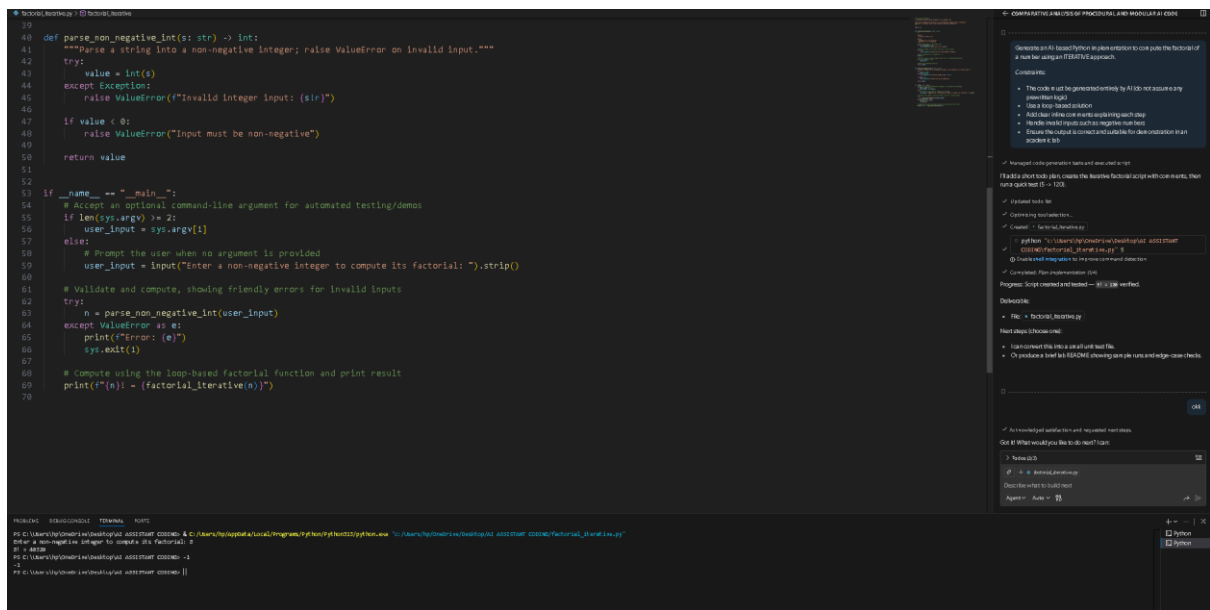
- The code must be generated using Python 3.10 or later.
- Use a loop-based approach.
- Add clear inline comments explaining each step.
- Handle invalid inputs (such as negative numbers) and ensure the output is an integer.

Deliverables:

- File: `factorial_iterative.py`
- Next steps (choose one):
 - Document this code as part of unit test file.
 - On production, use this code to generate a script to run and verify code checks.

Terminal Output:

```
PS C:\Users\johndoe> python .\factorial_iterative.py 5
5! = 120
PS C:\Users\johndoe> python .\factorial_iterative.py -1
Traceback (most recent call last):
  File ".\factorial_iterative.py", line 25, in factorial_iterative
    raise ValueError("Input must be a non-negative integer")
ValueError: Input must be a non-negative integer
```



```
34
35
36 def parse_non_negative_int(s: str) -> int:
37     """Parse a string into a non-negative integer; raise ValueError on invalid input."""
38     try:
39         value = int(s)
40     except Exception:
41         raise ValueError(f"Invalid integer input: {s}")
42
43     if value < 0:
44         raise ValueError("Input must be non-negative")
45
46     return value
47
48
49 if __name__ == "__main__":
50     # Accept an optional command-line argument for automated testing/demos
51     if len(sys.argv) >= 2:
52         user_input = sys.argv[1]
53     else:
54         # Prompt the user when no argument is provided
55         user_input = input("Enter a non-negative integer to compute its factorial: ").strip()
56
57     # Validate and compute, showing friendly errors for invalid inputs
58     try:
59         n = parse_non_negative_int(user_input)
60     except ValueError as e:
61         print(f"Error: {e}")
62         sys.exit(1)
63
64     # Compute using the recursive factorial function and print result
65     print(f"{n}! = {factorial_recursive(n)}")
66
```

PROBLEMS | DEBUG CONSOLE | TERMINAL | OUTPUT

```
PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE> & C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py
Enter a non-negative integer to compute its factorial: -1
Error: Input must be non-negative
PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE> & C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py
Enter a non-negative integer to compute its factorial: 2
2! = 2
PS C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE>
```

COMPARATIVE ANALYSIS OF ITERATIVE AND MODULAR AI CODE

Given inputs are from legacy code, your solutions, and constraints.

Generate an AI-based Python implementation to compute the factorial of a number using a RECURSIVE approach.

Constraints:

- The logic must be AI-generated without reference to any existing code.
- Include a proper base condition.
- Add comments explaining the recursive execution flow.
- Handle negative input values gracefully.
- Ensure the output matches the iterative factorial implementation.

Managed and tracked code items for planning

```
python "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py" 0;
python "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py" 1;
python "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py" 5;
python "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\factorial_recursive.py" 10
```

Breaks and responses to response command detection

Perfect! Now create a comparison text to verify both implementations match.

Can't > compare_factorial_approaches.py

```
python "C:\Users\vip\OneDrive\Desktop\AI ASSISTANT\CODE\compare_factorial_approaches.py"
```

Breaks and responses to response command detection

Completed! Test and verify (2/3)

Index (1/3)

Describe what to build next

Agent Auto

Short Comparison: Iterative vs Recursive Factorial

- **Readability:**
Iterative implementations are easier to read and follow due to their straightforward loop-based control flow. Recursive implementations are more abstract and may be harder to understand for beginners.
- **Stack Usage:**
Iterative approaches use constant stack memory, while recursive approaches consume additional stack space for each function call.
- **Performance Implications:**
Iterative implementations are generally faster and more memory-efficient. Recursive implementations incur extra overhead due to repeated function calls.
- **When Recursion Is Not Recommended:**
Recursion is not suitable for large inputs, performance-critical applications, or environments with limited stack memory, where iterative solutions are safer and more efficient.