

# **AI ASSISTANT CODING ASSIGNMENT-01**

**Name:** Sanjay karupothula

**Enrollment no:** 2303A52337

**Batch:** 34

**Semester/Year:** VI

**Branch:** CSE(AIML)- B.TECH

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:

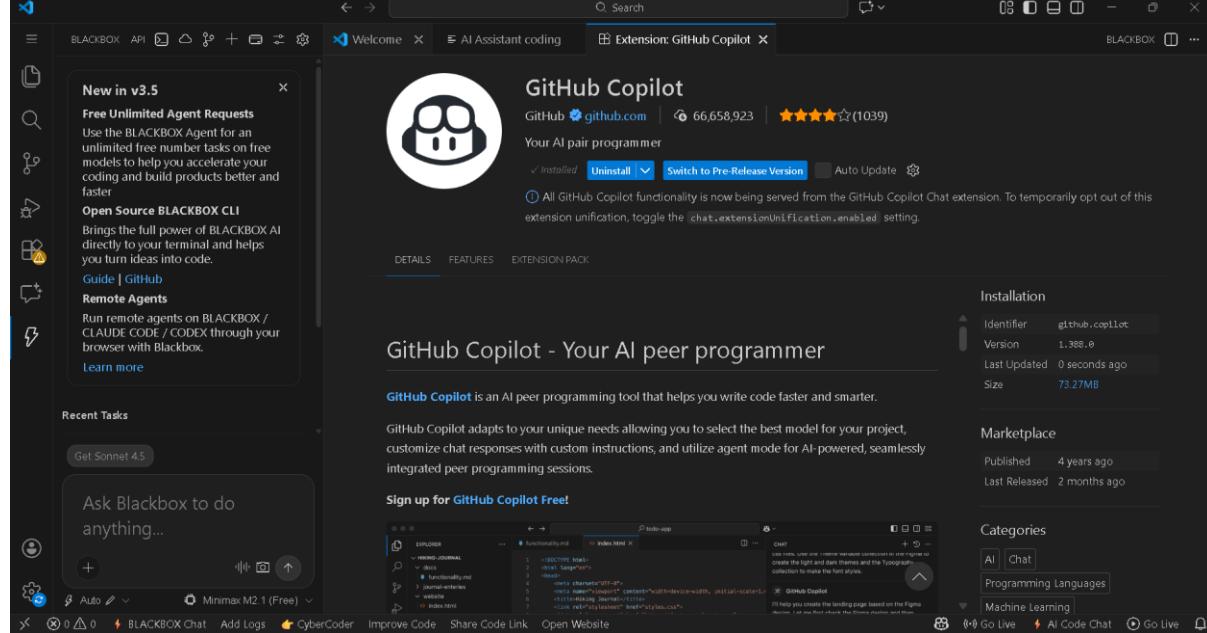
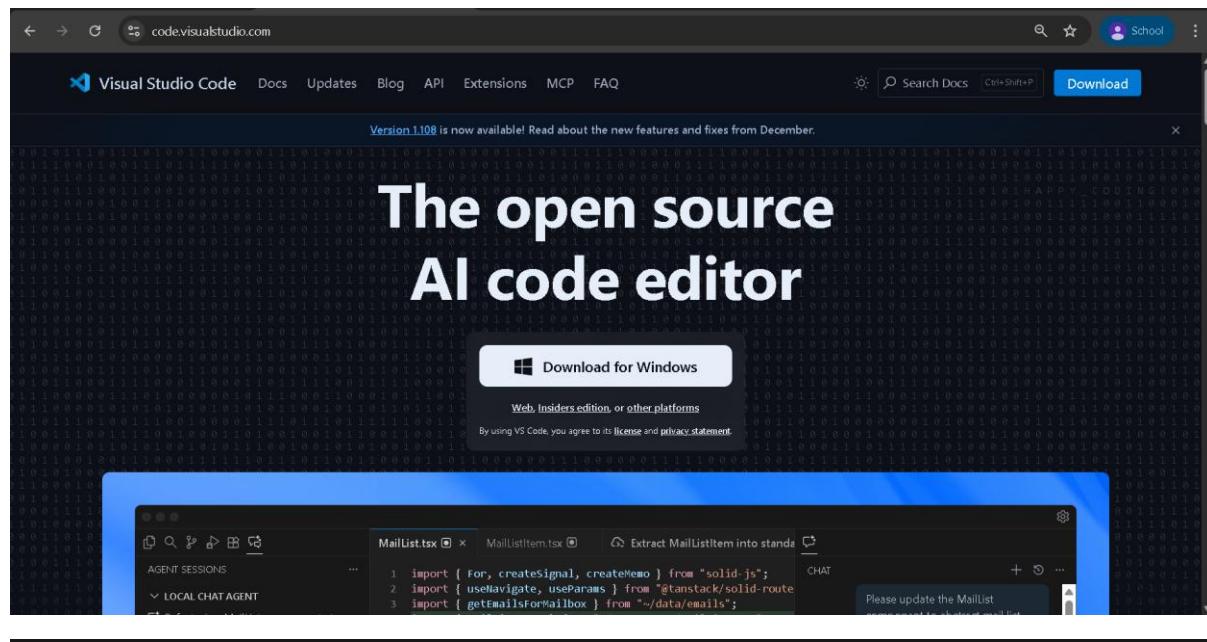
- Set up GitHub Copilot in VS Code successfully.
- Use inline comments and context to generate code with Copilot.
- Evaluate AI-generated code for correctness and readability.
- Compare code suggestions based on different prompts and programming styles.

## 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

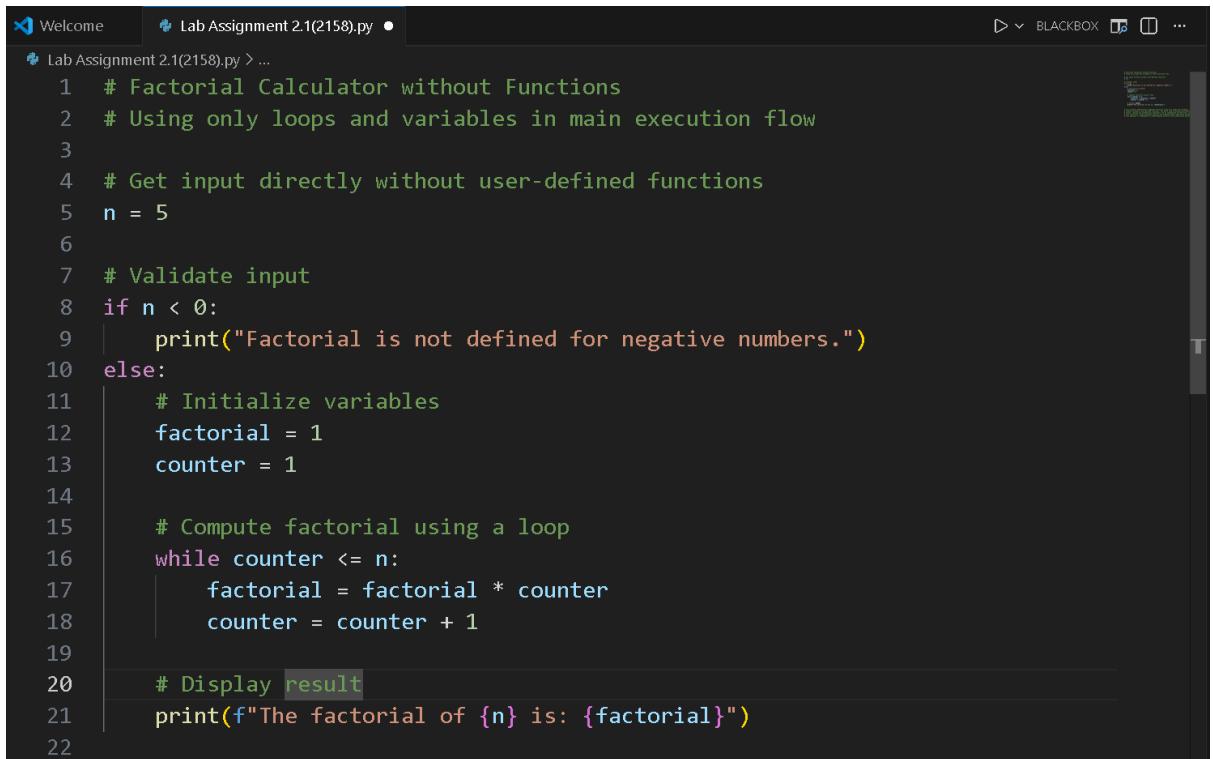
- A working Python program generated with Copilot assistance

- 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?



The screenshot shows a code editor window with a dark theme. The title bar says "Welcome" and "Lab Assignment 2.1(2158).py". The code itself is as follows:

```
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

The screenshot shows a software interface for AI-assisted coding. On the left, there's a code editor window titled "Lab Assignment 2.1(2158).py" containing the following Python code:

```
1 """
2 Factorial Calculator
3 Computes factorial using loops without user-defined functions.
4 """
5
6 # Get user input
7 n = int(input("Enter a non-negative integer: "))
8
9 # Validate and compute
10 if n < 0:
11     print("Error: Factorial is not defined for negative numbers.")
12 else:
13     result = 1
14     for i in range(1, n + 1):
15         result *= i
16     print(f"The factorial of {n} is: {result}")
17
18
```

To the right of the code editor, there's a sidebar with several sections:

- CHAT**: A message box with the text "Clean up this AI-generated code by optimizing logic, improving loop structure, and making it ready for a shared codebase."
- Code Cleanup Complete!**: A message indicating "Ready for shared codebase".
- Optimized Logic:**
  - Consolidated validation and computation into single if-else block
  - Added descriptive docstring (PEP 257 compliant)
  - Interactive input replaces hardcoded test value
- Describe what to build next**: A text input field.

## 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:
  - Prompt evolution
  - 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  
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` –

+ Lab Assignment 2.1(2158).py  
Describe what to build next  
Agent Auto

```

1 def calculate_factorial(number):
2     """
3         # Return the computed factorial value
4         return factorial_result
5     """
6
7     def main():
8         try:
9             # Get integer input from user
10            user_number = int(input("Enter a non-negative integer: "))
11
12            # Call the factorial function with user input
13            calculated_factorial = calculate_factorial(user_number)
14
15            # Display the result in a formatted message
16            print(f"The factorial of {user_number} is: {calculated_factorial}")
17
18        except ValueError as error:
19            print(f"Error: {error}")
20        except:
21            print("Error: Please enter a valid integer.")
22
23    # Standard Python idiom: execute main() only when script is run directly
24    if __name__ == "__main__":
25        main()
26
27

```

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  
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` –

+ Lab Assignment 2.1(2158).py  
Describe what to build next  
Agent Auto

PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/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\hp\OneDrive\Desktop\AI ASSISTANT CODING & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/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\hp\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 shows a software interface for comparing two Python implementations. On the left is a code editor with the following content:

```
# factorial.py
# Iterative factorial demo suitable for an academic lab.
# This script computes the factorial of a non-negative integer using a loop-based approach. It includes clear inline comments and input validation.
# 
# Import sys
# 
# def factorial_iterative(n: int) -> int:
#     """
#     Returns
#     - factorial value as int
#     Raises
#     - TypeError if n is not an int
#     - ValueError if n is negative
#     """
#     # Ensure the input is an Integer type
#     if not isinstance(n, int):
#         raise TypeError("Input must be an integer")
# 
#     # Negative numbers do not have factorials in this context
#     if n < 0:
#         raise ValueError("Input must be a non-negative Integer")
# 
#     # Start accumulator at 1 because 0! == 1
#     result = 1
# 
#     # Multiply result by each integer from 2 up to n (loop-based approach)
#     for i in range(2, n + 1):
#         result *= i
# 
#     # Return the computed factorial
#     return result
# 
# def parse_non_negative_int(s: str) -> int:
#     """Parse a string into a non-negative integer; raise ValueError on invalid input."""
# 
```

On the right is another code editor pane with the same code. Between them is a large central panel titled "COMPARISON ANALYSIS OF PROCEDURAL AND MODULAR APPROACH". The analysis section contains the following text:

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

Comments:

- The code is not well-structured by AI due to assume a very previous knowledge.
- Only one function.
- Additional info can be provided during each step.
- Very good.
- Ensure the output is correct and suitable for demonstration in an academic lab.

The "Test" section lists:

- Managed code generation test and executed script.
- If this short code does, create the recursive factorial script with one event, then run a quick test (S = 120).
- Updated tests file.
- Optimizing test cases.
- Create a -> Factorial recursive.

The "Fix" section lists:

- Fix the code to use a loop-based approach instead of recursion.
- Recursive implementation is fine, but it's better to use a loop.

The "Script" section lists:

- Completed. Script created and tested — S = 120 verified.
- Details:
- File: factorial.py
- Run these (choose one):
- Harness with the script and unit test file.
- Or produce a test file as README showing sample runs and edge-case checks.

The bottom right corner shows a terminal window with the following text:

```
WORKING DIRECTORY: TERMINAL WORKS
D:\Users\yash\OneDrive\Desktop\assassin\c00100 & C:\Users\yash\AppData\Local\Programs\Python\Python311\python.exe "D:\Users\yash\OneDrive\Desktop\c00100\factorial_iterative.py"
Enter a non-negative integer to compute its factorial:
0
C:\Users\yash\OneDrive\Desktop\assassin\c00100>
D:\Users\yash\OneDrive\Desktop\assassin\c00100>
```

The screenshot shows a Jupyter Notebook cell containing Python code. The code defines a recursive function `factorial_recursive` and an iterative function `factorial_iterative`. It includes a check function `check_factorial` and a main block that prints the results of both functions for values 0 through 5.

```
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n-1)

def factorial_iterative(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

def check_factorial():
    for i in range(6):
        assert factorial_iterative(i) == factorial_recursive(i), f"Test failed for n={i} (Recursive vs Iterative)."

check_factorial()
```

Below the code cell, the terminal output shows the execution of the notebook and the results of the check function.

```
PS C:\Users\hp\OneDrive\Desktop\assistant\code & C:\Users\hp\OneDrive\Desktop\assistant\code> python -m nbconvert factorial.ipynb --to python
# %%writefile factorial.py
# !/usr/bin/env python3
# coding: utf-8
# Author: zhengyuan
# Email: zhengyuan@zjhu.edu.cn
# Date: 2023-09-01 14:45:10
# Last Modified: 2023-09-01 14:45:10
# Version: 1.0
# Description: This file contains two implementations of the factorial function: a recursive version and an iterative version. A check function is also provided to verify the correctness of the implementations.

# Recursive implementation
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n-1)

# Iterative implementation
def factorial_iterative(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

# Check function to verify the correctness of the implementations
def check_factorial():
    for i in range(6):
        assert factorial_iterative(i) == factorial_recursive(i), f"Test failed for n={i} (Recursive vs Iterative)."

# Run the check function
check_factorial()

# Print the results of the factorial function for n from 0 to 5
for i in range(6):
    print(f"Factorial of {i} (Recursive): {factorial_recursive(i)}")
    print(f"Factorial of {i} (Iterative): {factorial_iterative(i)}")
```

```
def factorial(n: int) -> int:
    """Compute n! using recursion.

    The algorithm works by breaking down the problem into smaller subproblems:
    - Base case: when n is 0 or 1, return n (mathematical definition)
    - Recursive case: n! = n * (n-1)!

    Parameters
    - n: non-negative integer

    Returns
    - factorial value as int

    Raises
    - TypeError if n is not an int
    - ValueError if n is negative
    """
    # Ensure the input is an integer type
    if not isinstance(n, int):
        raise TypeError("Input must be an integer")

    # Negative numbers do not have factorials in this context
    if n < 0:
        raise ValueError("Input must be a non-negative integer")

    # BASE CASE: The recursion terminates here
    if n == 0 or n == 1:
        return 1

    # RECURSIVE CASE: Break problem into smaller subproblem
    return n * factorial(n - 1)

def parse_non_negative_int(s: str) -> int:
    """Parse a string into a non-negative integer; raise ValueError on invalid input."""
    try:
        value = int(s)
    except Exception:
        raise ValueError(f"Invalid integer input: {s!r}")

if __name__ == "__main__":
    print(parse_non_negative_int("42"))
    print(parse_non_negative_int("0"))
    print(parse_non_negative_int("-42"))

    # Test cases
    assert parse_non_negative_int("42") == 42
    assert parse_non_negative_int("0") == 0
    assert parse_non_negative_int("-42") == None
```

Welcome Lab Assignment 2.102158.py factorial\_recursive.py test\_factorial\_iterative.py factorial\_iterative.py compare\_factorial\_approaches.py

pydantic\_recursive.py ...

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!r}")  
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 PORTS

PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\& C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py"  
Enter a non-negative integer to compute its factorial: 1  
1  
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\& C:\Users\hp\AppData\Local\Programs\Python\Python313\python.exe "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py"  
Enter a non-negative integer to compute its factorial: 2  
2!  
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\&

BUCKDOCK CHAT COMPARATIVE ANALYSIS OF PROCEDURAL AND MODULAR CODE  
about https://github.com/yourusername/yourrepository

Generates an Aligned Python implementation to compute the factorial of a number using a RECURSIVE approach.

Constraints:

- The logic must be Al-generated without reference to any existing code
- Handle a single 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 todo items for planning

- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 01
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 02
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 1
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 5
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 10
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\factorial\_recursive.py" 10

Enable shell integration to improve command detection

Perf: Now create a comparison test to verify both implementations match:

- Created "compare\_factorial\_approaches.py"
- python "C:\Users\hp\OneDrive\Desktop\AI ASSISTANT\CODE\compare\_factorial\_approaches.py"
- CODING! compare\_factorial\_approaches.py
- Enable shell integration to improve command detection

Completed Test and verify (0/0)

> Tasks (3/3)

P: ⚡ fastest, memory

Describe what to build next:

Agent: Auto - 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.