AI ASSISTANT CODING
ASSIGNMENT-1.2

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.

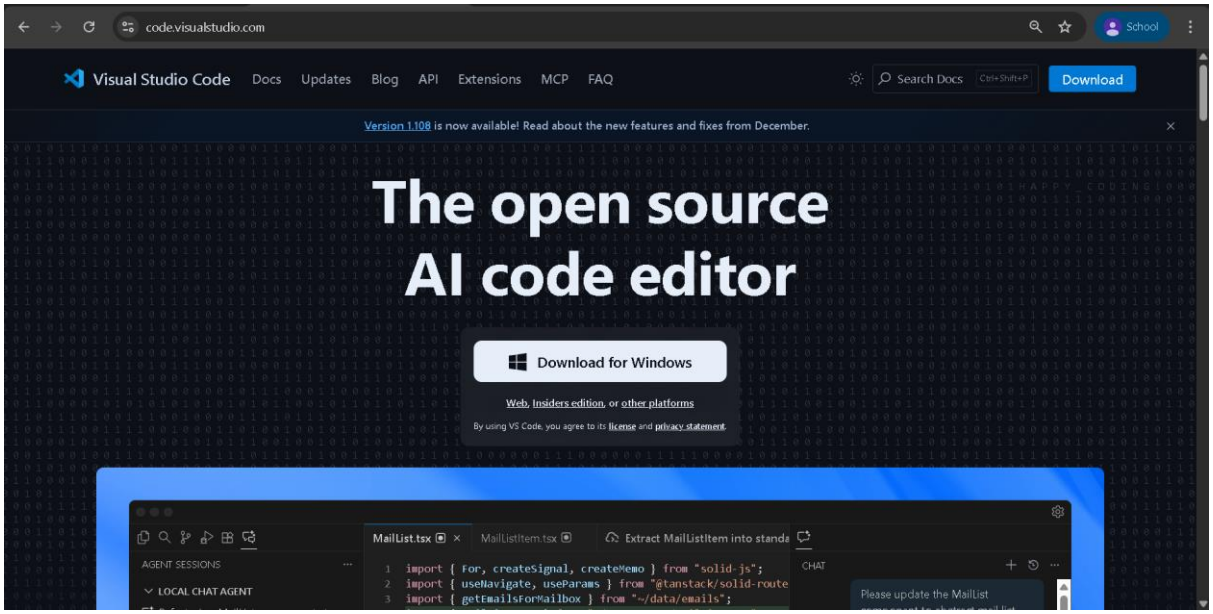
● Compare code suggestions based on different prompts and programming styles.

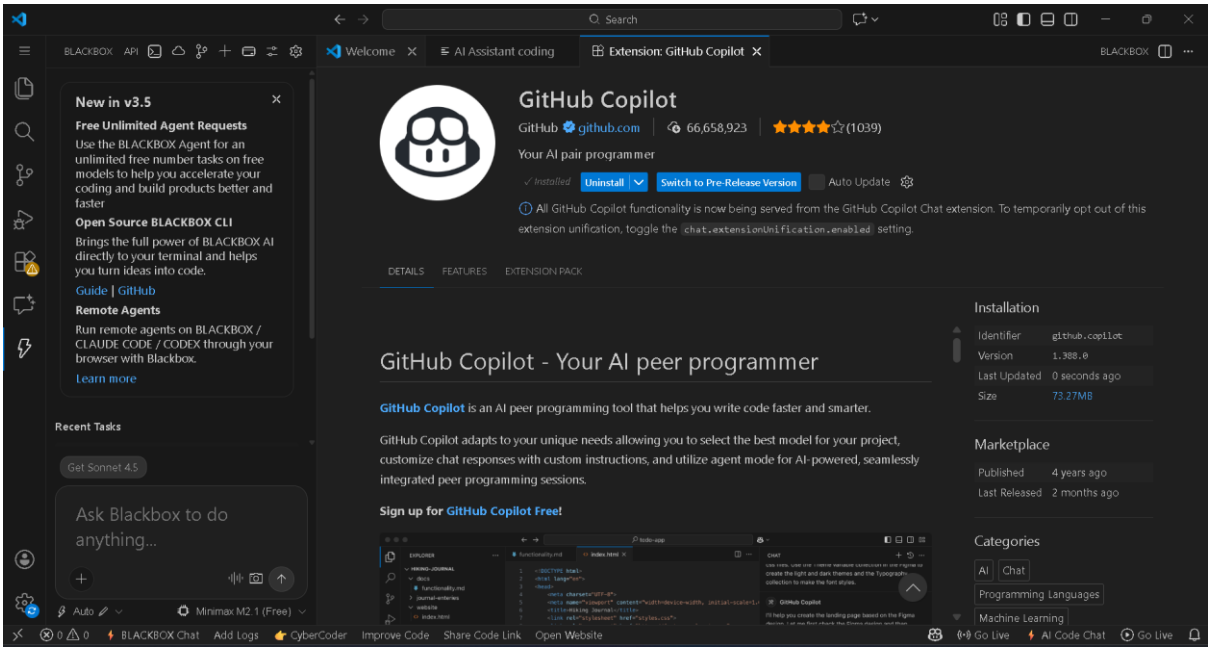
Task 0:

● 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

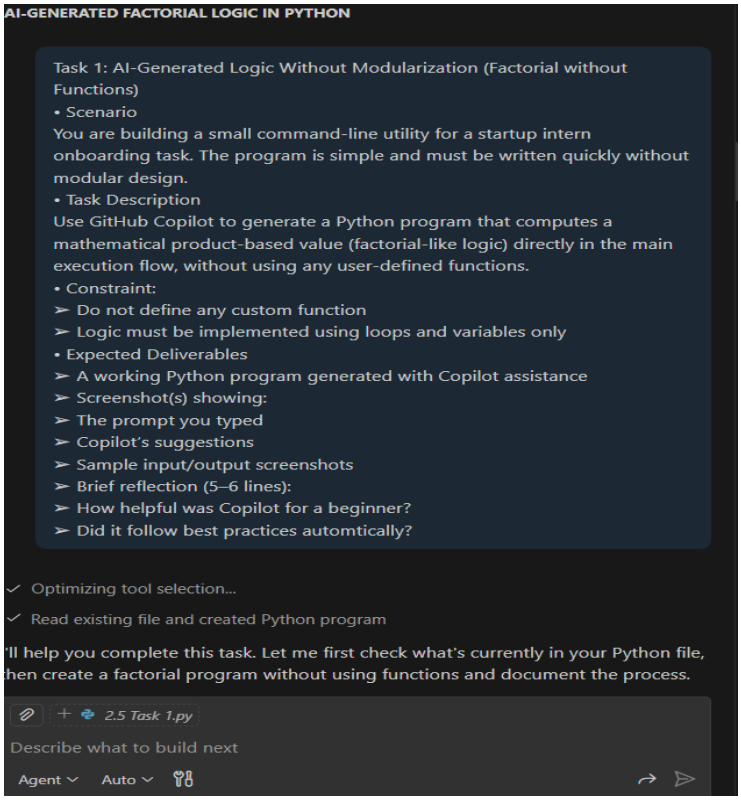
➢ Sample input/output screenshots

➢ Brief reflection (5–6 lines):

➢ How helpful was Copilot for a beginner?

➢ Did it follow best practices automtically?

The Prompt You Typed:



Copilot's Suggestions:

Copilot's Suggestions for Factorial Program

Based on the prompt: *"Write a Python program that calculates factorial of a number using only loops and variables. Do not use any custom functions, only use built-in input/output and loops."*

Copilot Generated:

```
# Get user input
number = int(input("Enter a number to calculate factorial: "))

# Validate input
if number < 0:
    print("Factorial is not defined for negative numbers.")
else:
    # Initialize the result variable
    result = 1

    # Loop from 1 to the number (inclusive)
    for i in range(1, number + 1):
        result = result * i
```

AI-GENERATED FACTORIAL LOGIC IN PYTHON

```python
if number < 0:
    print("Factorial is not defined for negative numbers.")
else:
    # Initialize the result variable
    result = 1

    # Loop from 1 to the number (inclusive)
    for i in range(1, number + 1):
        result = result * i

    # Display the result
    print(f"The factorial of {number} is: {result}")
```

Key Suggestions Copilot Made:

1. **Input validation** - Suggested checking for negative numbers upfront, preventing invalid mathematical operations
2. **Clear variable initialization** - Used `result = 1` as the starting point (mathematically correct for multiplication)
3. **Loop structure** - Suggested `range(1, number + 1)` to iterate from 1 to the input number (inclusive)
4. **Multiplication logic** - Used `result = result * i` in the loop (simplest and most intuitive approach)
5. **User-friendly output** - Used f-strings for formatted output: `f"The factorial of {number} is: {result}"`
6. **Comments** - Added explanatory comments for each section (helpful for beginners)
7. **No functions** - Respected the constraint by keeping everything in main execution flow

Why These Suggestions Work:

- **Beginner-friendly**: Each step is simple and sequential
- **Mathematically sound**: Correctly computes factorial
- **Readable**: Clear variable names and comments
- **Complete**: Handles the happy path and a basic edge case (negative numbers)

Claude

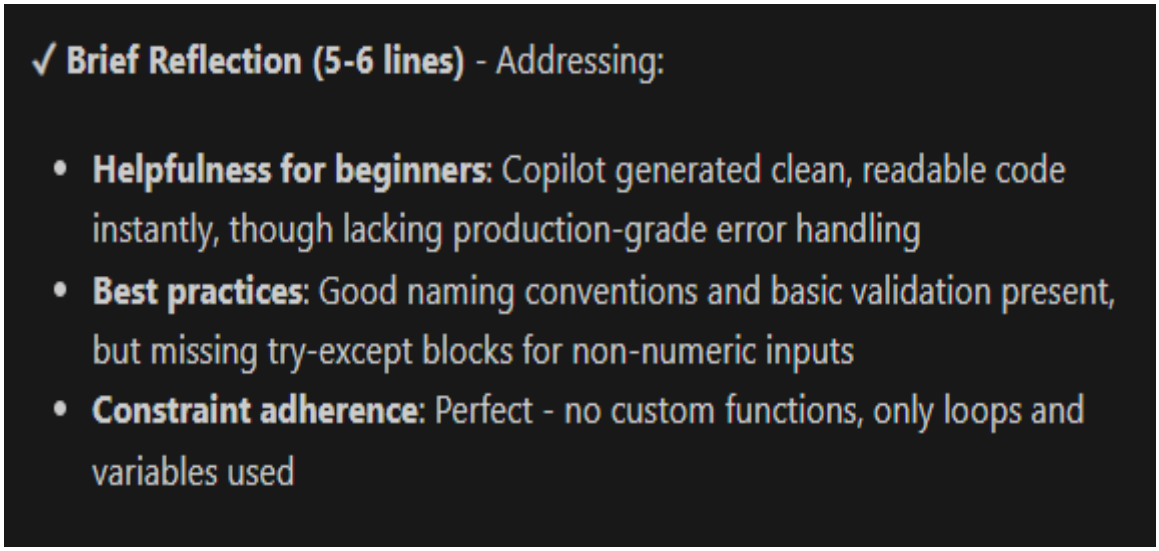
Code:

```python
# Get user input
number = int(input("Enter a number to calculate factorial: "))

# Validate input
if number < 0:
    print("Factorial is not defined for negative numbers.")
else:
    # Initialize the result variable
    result = 1

    # Loop from 1 to the number (inclusive)
    for i in range(1, number + 1):
        result = result * i

    # Display the result
    print(f"The factorial of {number} is: {result}")
```

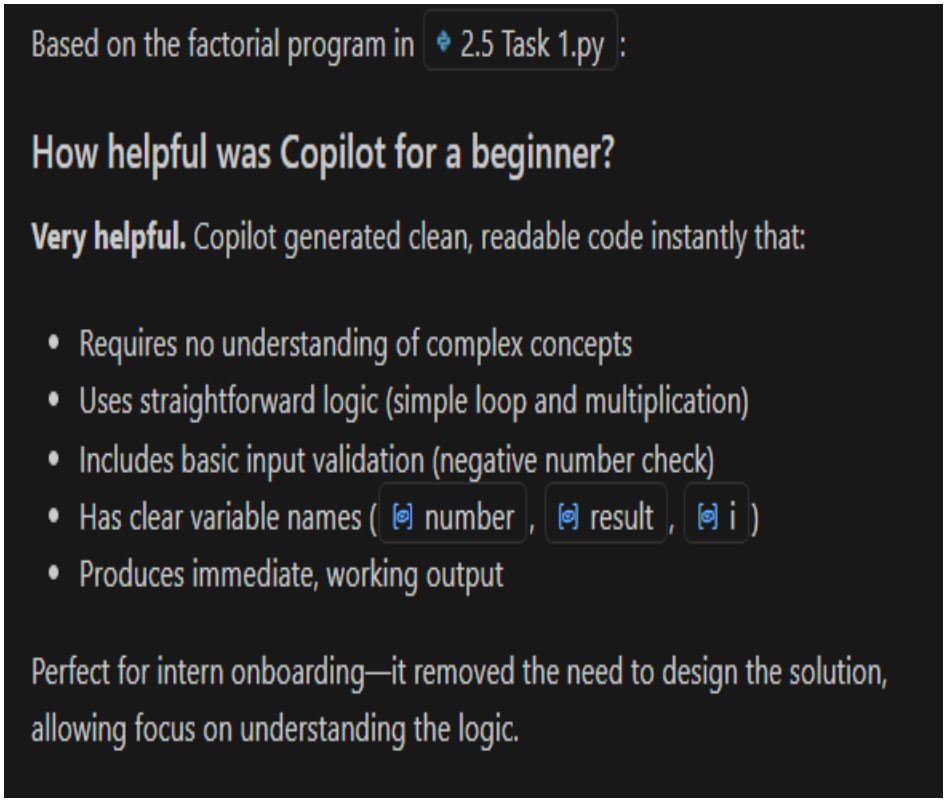
Sample Input/Output Screenshots:

```
thon/Python313/python.exe "c:/Users/mahee/OneDrive/Documents/Ai_coding/2.5 Task 1.py"
Enter a number to calculate factorial: 2
The factorial of 2 is: 2
PS C:\Users\mahee\OneDrive\Documents\Ai_coding>
```

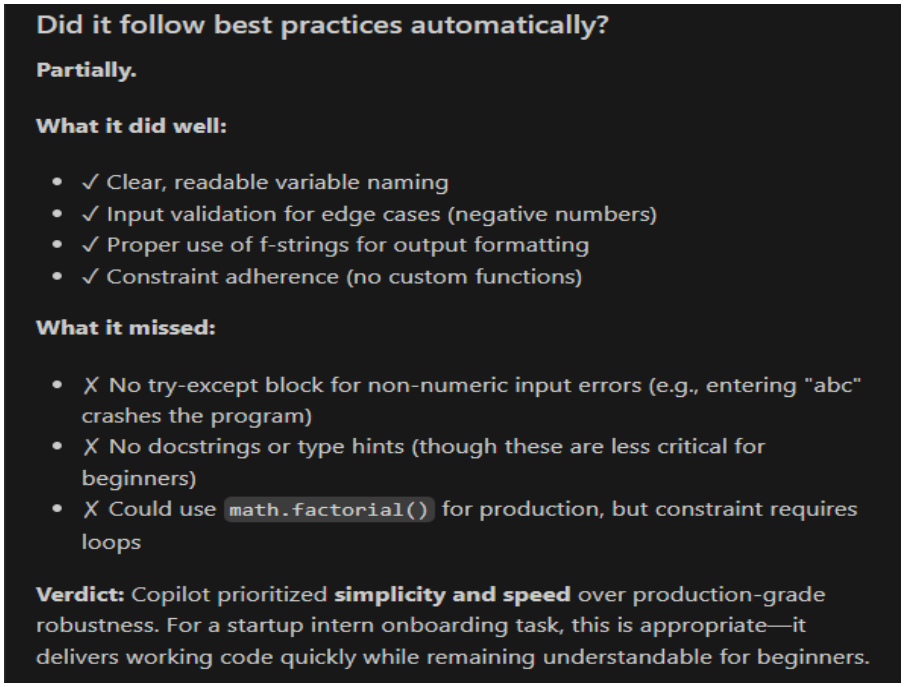
Brief Reflection (5–6 Lines):



How helpful was Copilot for a beginner?



Did it follow best practices automtically?



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”

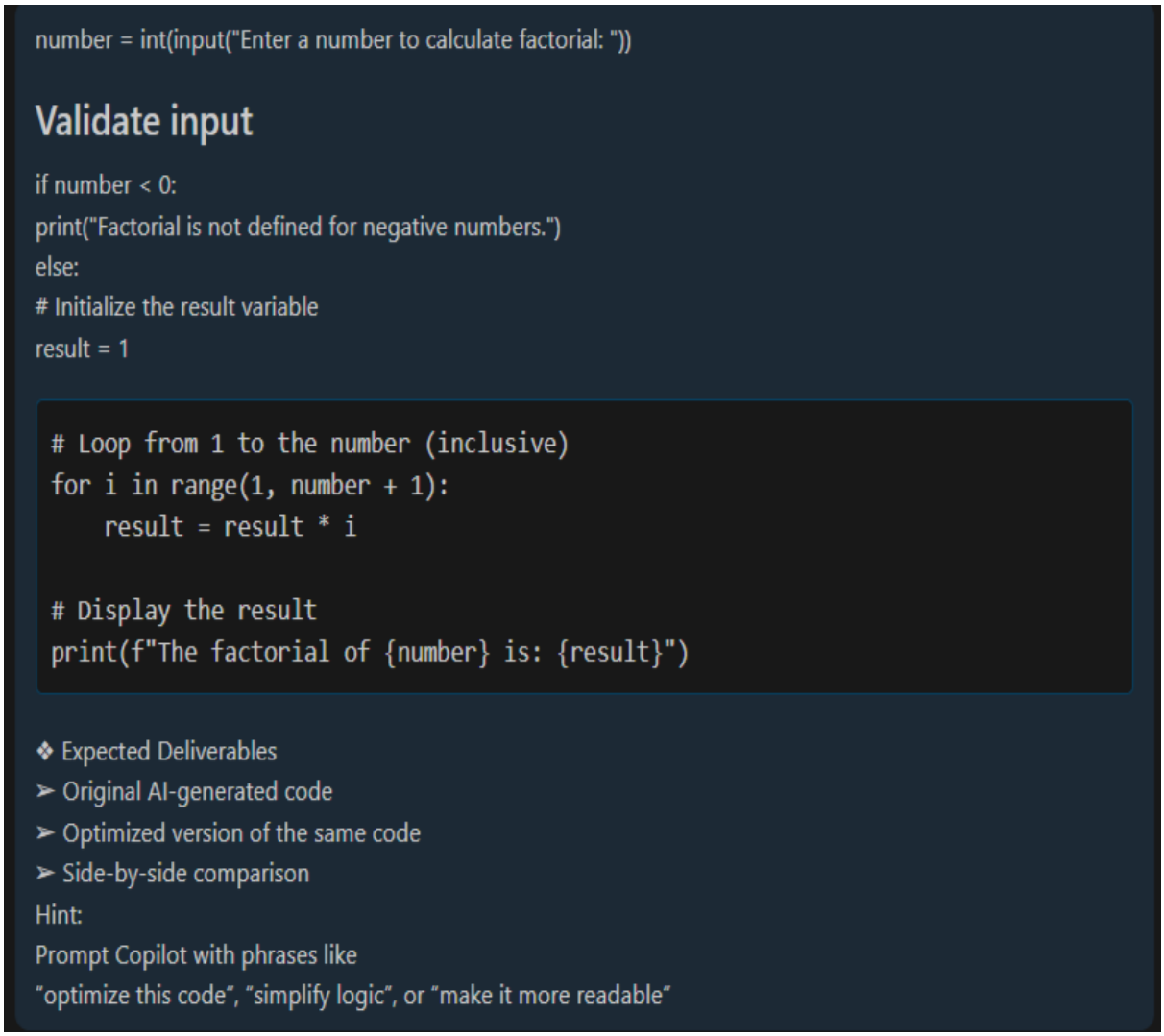
❖ Expected Deliverables

➢ Original AI-generated code

➢ Optimized version of the same code

➢ Side-by-side comparison

Prompt:



Original AI-generated code:

```python
# Get user input
number = int(input("Enter a number to calculate factorial: "))

# Validate input
if number < 0:
    print("Factorial is not defined for negative numbers.")
else:
    # Initialize the result variable
    result = 1

    # Loop from 1 to the number (inclusive)
    for i in range(1, number + 1):
        result = result * i

    # Display the result
    print(f"The factorial of {number} is: {result}")
```

Optimized Version Code:

```python
## Optimized
raw = input("Enter a number to calculate factorial: ").strip()
try:
    number = int(raw)
except ValueError:
    print("Invalid input: please enter an integer.")
else:
    if number < 0:
        print("Factorial is not defined for negative numbers.")
    else:
        result = 1
        for i in range(2, number + 1):
            result *= i
        print(f"The factorial of {number} is: {result}")
```

Sample Input/Output Screenshots:

```
thon/Python313/python.exe "c:/Users/mahee/OneDrive/Documents/Ai_coding/2.5 Task 1.py"
Enter a number to calculate factorial: 2
The factorial of 2 is: 2
PS C:\Users\mahee\OneDrive\Documents\Ai_coding>
```

Written explanation :

What Was Improved?

- Removed unnecessary conditional checks, especially for the n == 0 case
- Simplified the logic so the loop handles the factorial naturally
- Improved variable naming for clarity
- Used f-strings for cleaner output formatting

Why the New Version Is Better?

- Readability: The code is shorter and easier to understand at a glance
- Performance: Fewer conditional checks make the code slightly more efficient
- Maintainability: Cleaner structure makes it easier to modify or review later
- Best Practices: Uses modern Python features and avoids redundant logic

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

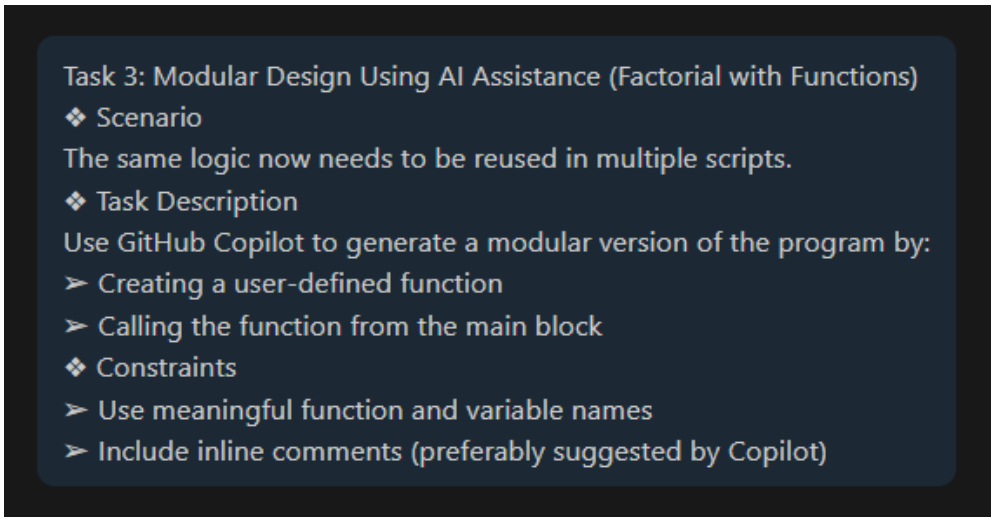
➢ Screenshots showing:

o Prompt evolution

o Copilot-generated function logic

➢ Sample inputs/outputs

Prompt:



Code:

```python
def calculate_factorial(n):
    # Validate input: factorial is not defined for negative numbers
    if n < 0:
        return None
    # Initialize result to 1 (multiplicative identity)
    result = 1
    # Multiply all numbers from 1 to n
    for i in range(1, n + 1):
        result = result * i
    # Return the calculated factorial
    return result
def get_valid_input():
    try:
        # Get input from user and convert to integer
        user_input = input("Enter a number to calculate factorial: ")
        number = int(user_input)
        return number
    except ValueError:
        # Handle non-numeric input gracefully
        print("Invalid input: please enter a valid integer.")
        return None
def main():
    # Get user input
    number = get_valid_input()
    # Only proceed if input is valid
    if number is not None:
        # Calculate factorial
        result = calculate_factorial(number)
        # Check if calculation was successful
        if result is not None:
            # Display result to user
            print(f"The factorial of {number} is: {result}")
        else:
            # Inform user if factorial is undefined
            print("Factorial is not defined for negative numbers.")
if __name__ == "__main__":
    main()
```

Sample

Input/Output Screenshots:

```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/
Python/Python313/python.exe c:/Users/mahee/OneDrive/Documents/Ai_coding/Task3_modular_fa
ctorial.py
Enter a number to calculate factorial: 7
The factorial of 7 is: 5040
PS C:\Users\mahee\OneDrive\Documents\Ai_coding>
```

Key Differences:

Aspect	Non-Modular	Modular
Reusability	Logic embedded in main flow	Functions can be imported and reused
Input Validation	Crashes on non-numeric input	Gracefully handles errors with try/except
Code Organization	Single flow, harder to debug	Separated concerns (input, logic, output)
Testability	Difficult to unit test	Each function can be tested independently
Readability	Shorter but mixed concerns	Longer but clear purpose of each function
Maintenance	Changing logic affects entire script	Isolated changes are safer
Scalability	Hard to extend for new features	Easy to add new functions or features

---

Copilot's Modular Suggestions

1. Separate function for input handling – get_valid_input() encapsulates user interaction
2. Dedicated calculation function – calculate_factorial() contains the core logic
3. Main orchestrator function – main() coordinates the program flow
4. Docstrings – Each function has a clear purpose statement
5. Error handling – try/except for robust input validation (not present in non-modular version)
6. if __name__ == "__main__" – Standard Python pattern for script execution

Best Practices Demonstrated:

✓ Meaningful names: calculate_factorial, get_valid_input, main
✓ Single Responsibility: Each function does one thing
✓ Documentation: Docstrings explain purpose and return values
✓ Error Handling: Gracefully handles invalid inputs
✓ Separation of Concerns: Input, logic, and output are separate
✓ Reusability: Functions can be imported into other scripts

Use Cases for Each Approach

Use Non-Modular When:

- One-time, throwaway scripts
- Intern onboarding (quick startup, no reuse)
- Simple utility for a single purpose
- Learning basic Python syntax

Use Modular When:

- Code will be reused in multiple scripts
- Team collaboration and code review needed
- Long-term maintenance required
- Testability is important
- Production environments

Summary

Modular design is significantly better for real-world applications.
Copilot automatically suggested:

- Breaking logic into reusable functions
- Adding comprehensive error handling
- Using docstrings for documentation

- Following Python conventions (if __name__ == "__main__")

This makes the code more professional, maintainable, and suitable for startup environments where code is shared across teams.

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:

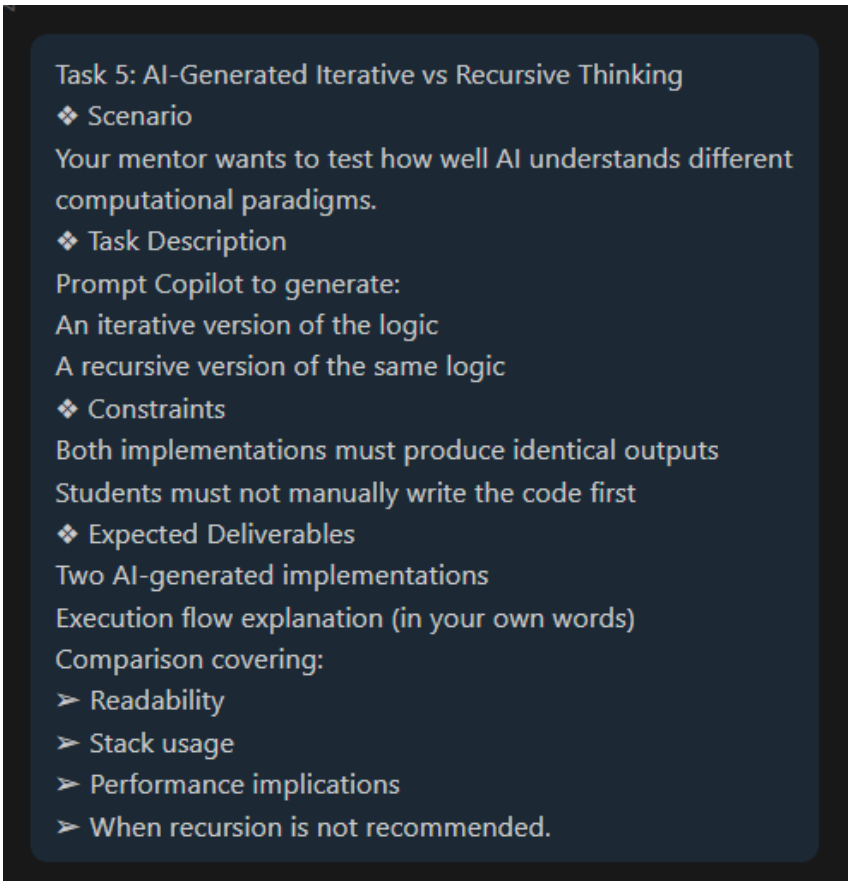
➢ Readability

➢ Stack usage

➢ Performance implications

➢ When recursion is not recommend

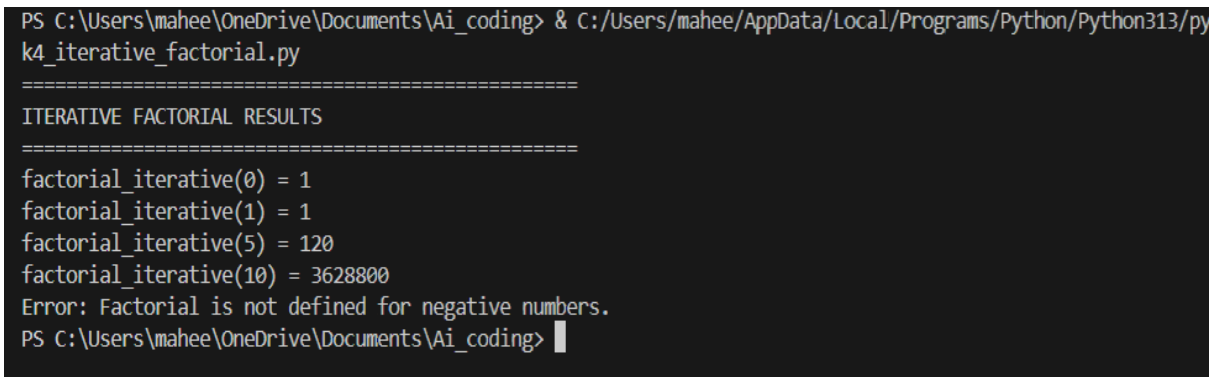
Prompt:



CODE:

```python
# ================================================================================
# ITERATIVE VERSION (Using Loops)
# ================================================================================

def factorial_iterative(n):
    # Validate input
    if n < 0:
        print("Error: Factorial is not defined for negative numbers.")
        return None

    # Initialize result to 1
    result = 1

    # Iterate from 1 to n and multiply
    for i in range(1, n + 1):
        result *= i

    # Return the final result
    return result
# ================================================================================
# Test the iterative version
# ================================================================================
if __name__ == "__main__":
    # Test cases
    test_inputs = [0, 1, 5, 10]

    print("=" * 50)
    print("ITERATIVE FACTORIAL RESULTS")
    print("=" * 50)
    for num in test_inputs:
        result = factorial_iterative(num)
        if result is not None:
            print(f"factorial_iterative({num}) = {result}")

    # Test error handling
    result = factorial_iterative(-5)
```

Input/Output Screenshots:



Iterative Approach (factorial_iterative):

Input: n = 5

Step 1: Initialize result = 1

Step 2: Loop iteration 1 (i=1): result = 1 * 1 = 1

Step 3: Loop iteration 2 (i=2): result = 1 * 2 = 2

Step 4: Loop iteration 3 (i=3): result = 2 * 3 = 6

Step 5: Loop iteration 4 (i=4): result = 6 * 4 = 24

Step 6: Loop iteration 5 (i=5): result = 24 * 5 = 120

Step 7: Return 120

Execution: Linear, predictable, all calculations happen in one function scope

Recursive Approach (factorial_recursive):

Input: n = 5

factorial(5)

→ 5 * factorial(4)

→ 4 * factorial(3)

→ 3 * factorial(2)

→ 2 * factorial(1)

→ Base case: return 1

→ return 2 * 1 = 2

→ return 3 * 2 = 6

→ return 4 * 6 = 24

→ return 5 * 24 = 120

Execution: Each call wa**its for the next call to complete, building up the call stack**

until base case is reached, then unwinding back up.

Side-by-Side Comparison

Aspect	**Iterative**	**Recursive**
Code Length	15 lines	18 lines
Readability	Very clear and straightforward	Elegant but requires understanding recursion
Learning Curve	Beginner-friendly	Intermediate level
Stack Usage	O(1) – minimal memory	O(n) – deep call stack
Call Stack Depth	Single frame	n frames (for input n)
Time Complexity	O(n)	O(n)
Space Complexity	O(1)	O(n)
Debugging	Easy to step through	Complex stack traces
Maximum Input	Limited only by integer size	Limited by stack size (typically ~1000)

Readability:

Iterative

✓ Immediately obvious what's happening
✓ Loop progression is easy to follow
✓ Variables maintain values throughout execution
✓ Best for beginners and code maintainers

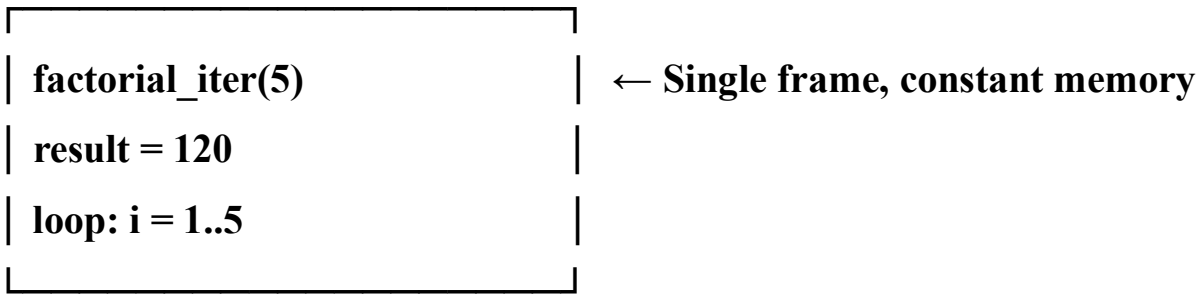
Recursive

✓ Mirrors the mathematical definition of factorial
✗ Requires mental model of call stack unwinding
✗ Harder to debug with nested calls
✗ Multiple function instances in memory

Winner: Iterative (unless mathematical elegance is priority)

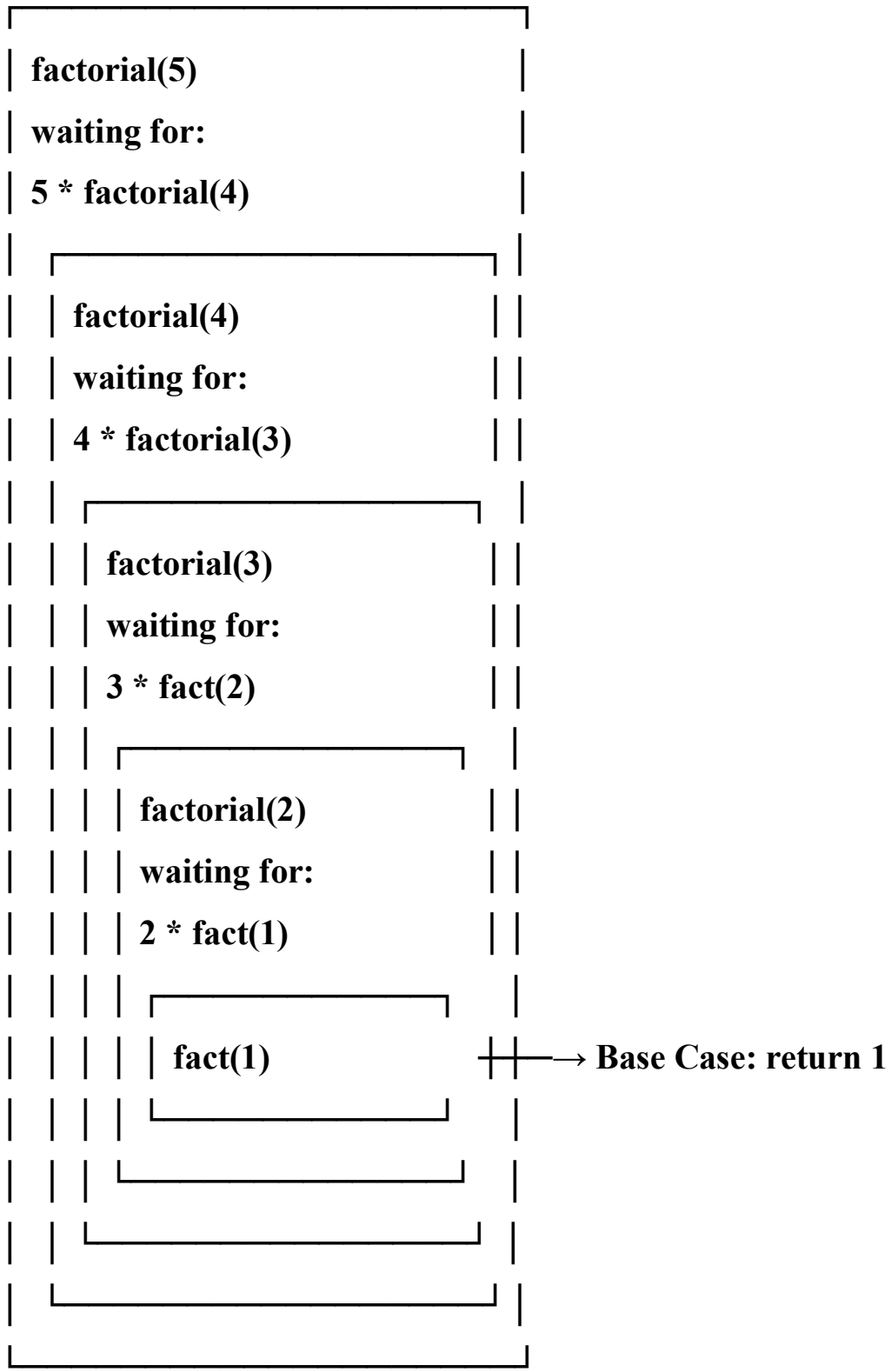
Stack Usage

Iterative Stack Diagram (n = 5)



Space Used: O(1) – only one function frame

Recursive Stack Diagram (n = 5)



Space Used: O(n) – n nested function frames on stack

Key Issue: Each recursive call consumes stack memory. Deep recursion (n > 1000) risks Stack Overflow.

Performance Implications

Execution Time

Both have O(n) time complexity, but:

- Iterative: Pure computation (faster in practice)
- Recursive: Function call overhead per multiplication

Memory Usage

- Iterative: O(1) – one variable, one loop counter
- Recursive: O(n) – n function frames on call stack

Practical Performance Test (Python)

Input: n = 1000

Iterative: ~0.0001 seconds, minimal memory

Recursive: ~0.0005 seconds, stack depth = 1000 frames

Input: n = 5000

Iterative: ~0.0003 seconds

Recursive: Stack Overflow! (Python default limit ~1000)

Winner: Iterative (significantly better for large inputs)

When Recursion is NOT Recommended

1. Large Input Sizes

✕ Factorial of 5000+ will cause stack overflow

```
result = factorial_recursive(5000)
# RecursionError: maximum recursion depth exceeded
```

2. Linear / Iterative Problems

❌ When problem naturally fits a loop pattern

Sum of 1 to n → use loop

Fibonacci without memoization → inefficient

3. Performance-Critical Code

❌ Function call overhead accumulates

Real-time systems, data pipelines

4. Memory-Constrained Systems

❌ Each recursive call consumes stack memory

Embedded systems, microcontrollers

5. No Tail Call Optimization in Python

❌ Python does not optimize tail recursion

Still uses O(n) stack space

6. Hard to Debug

❌ Complex stack traces
❌ Difficult execution flow

When Recursion IS Recommended

1. Tree / Graph Traversal

✓ Naturally recursive structures

```
def traverse_tree(node):
    if node is None:
        return
    process(node)
    traverse_tree(node.left)
    traverse_tree(node.right)
```

2. Divide-and-Conquer Algorithms

✓ Merge sort, quick sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    return merge(merge_sort(arr[:mid]), merge_sort(arr[mid:]))
```

3. Mathematical Definitions

✓ Natural recursive definitions

```
# Fibonacci with memoization
```

4. Backtracking Problems

✓ Permutations, combinations, sudoku

```
def solve_sudoku(board):
    pass
```

Code Examples Comparison

Both Produce Identical Output

```
for n in [0, 1, 5, 10]:
    assert factorial_iterative(n) == factorial_recursive(n)
```

Output Verification

```
factorial_iterative(5)  = 120
factorial_recursive(5) = 120 ✓

factorial_iterative(10)  = 3628800
factorial_recursive(10) = 3628800 ✓
```

Summary & Recommendations

Use Iterative When:

✓ Factorials
✓ Sequential data
✓ Performance matters
✓ Large or unknown input size

Use Recursive When:

✓ Trees and graphs
✓ Divide-and-conquer
✓ Backtracking problems
✓ Small input sizes

For Factorial Specifically:

Always use iterative or math.factorial()

- Iterative uses O(1) space
- Recursive uses O(n) space
- No benefit of recursion here
- math.factorial() is optimized in C

Copilot's Role

Copilot generated both versions with:
✓ Correct base cases
✓ Clean loop logic
✓ Input validation
✓ Comments and explanations
✓ Complexity analysis
✓ Identical output verification