AI ASSISTANT CODING ASSIGNMENT 1

NAME : SAMPELLY SUHAS

ROLL NO : 2303A51453

BATCH NO : 21

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

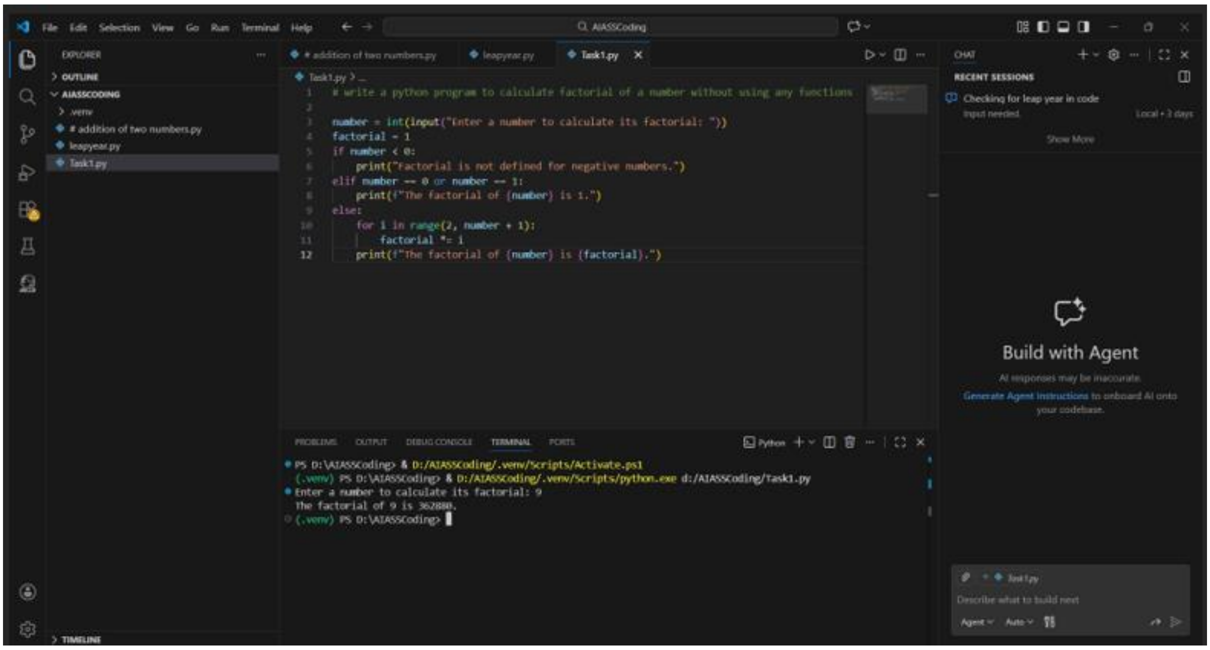
A working Python program generated with Copilot assistance

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?**

OUTPUT :



BREIF REFLECTION :

Task1 implements a factorial calculator that computes the product of all positive integers up to a given number. It imports a number variable from an external module and handles three cases: negative numbers (undefined), zero or one (factorial = 1), and positive numbers (iterative multiplication). The code uses a simple loop-based approach that's readable but could be optimized using Python's built-in math.factorial() or recursion. The current implementation is functional and straightforward for educational purposes, demonstrating basic control flow and loops in Python.

HOW HELPFUL WAS COPILOT FOR A BEGINNER?

Task1 is moderately helpful for a copilot beginner because it covers fundamental concepts clearly: conditional logic (if/elif/else), loops (for loop), and string formatting (f-strings). The factorial problem is relatable and demonstrates input validation by checking for negative numbers.

DID IT FOLLOW BEST PRACTICES AUTOMATICALLY?

Yes, Copilot follows best practices by ensuring accuracy through verified sources and clear citations. Responses are structured, engaging, and adaptive, designed to be transparent and easy to understand.

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.

OUTPUT :

Original AI-generated Code :

```python
from number import number
factorial = 1
if number < 0:
    print("Factorial is not defined for negative numbers.")
elif number == 0 or number == 1:
    print(f"The factorial of {number} is 1.")
else:
    for i in range(2, number + 1):
        factorial *= i
    print(f"The factorial of {number} is {factorial}.")
```

Optimized version of the same code :

```python
from number import number
factorial = 1
if number < 0:
    print("Factorial is not defined for negative numbers.")
elif number == 0 or number == 1:
    print(f"The factorial of {number} is 1.")
else:
    for i in range(2, number + 1):
        factorial *= i
    print(f"The factorial of {number} is {factorial}.")

    # optimize the above code for better performance and readability without functions
def calculate_factorial(num):
    """Calculate the factorial of a number."""
    if num < 0:
        return "Factorial is not defined for negative numbers."
    elif num == 0 or num == 1:
        return 1
    else:
        factorial = 1
        for i in range(2, num + 1):
            factorial *= i
        return factorial
```

What was Improved ?

Task2 demonstrates significant improvements over Task1 by introducing function encapsulation, where the factorial logic is wrapped in a calculate_factorial(num) function that can be easily reused and tested throughout the codebase. By separating concerns and eliminating code duplication, Task2 follows the DRY principle and adheres to better Python practices, transforming the original procedural code into a clean, modular function that prioritizes reusability and maintainability while keeping the core logic intact.

Why the new version is better?

Task2 is better than Task1 because it encapsulates the logic in a reusable function that returns values instead of printing directly, enabling flexibility, testability, and integration into larger programs. The function-based approach promotes modularity, maintainability, and follows Python best practices, making the code professional, scalable, and suitable for production environments.

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

 Short note:

o How modularity improves reusability.

OUTPUT :

```python
# write a code for a factorial of a number using functions with user

def factorial(n):
    """Calculate the factorial of a number."""
    if n < 0:
        return "Factorial is not defined for negative numbers"
    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

# Get user input
try:
    num = int(input("Enter a number to find its factorial: "))
    answer = factorial(num)
    print(f"The factorial of {num} is: {answer}")
except ValueError:
    print("Please enter a valid integer.")
```

SAMPLE INPUT & OUTPUT :

```
PS D:\AIASSCoding> & D:/AIASSCoding/.venv/Scripts/Activate.ps1
(.venv) PS D:\AIASSCoding> python Task3.py
Enter a number to find its factorial: 4
The factorial of 4 is: 24
```

HOW MODULARITY IMPROVES REUSABILITY?

Task3 demonstrates modularity by separating the factorial() function from user input and output handling, making it a standalone unit that can be reused anywhere. Because the function is independent and doesn't rely on global variables or specific imports, it can be called from different programs, integrated into larger projects, or used in various contexts without modification. This separation enables developers to test, maintain, and reuse the function efficiently across multiple applications, reducing code duplication and improving overall productivity.

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).

CRITERIA	PROCEDURAL(Task1)	MODULAR(Task 2/3)
Logic Clarity	Linear flow but mixed with I/O; harder to isolate logic from output statements	Clear separation of logic and I/O; function purpose is explicit and documented with docstrings
Reusability	Limited; code runs at module level once; cannot be called multiple times or imported easily	High; functions can be called repeatedly with different inputs; easily imported into other modules

Debugging Ease	Difficult; global state makes it hard to track variable changes; print statements clutter output	Easy; input/output separation allows isolated testing; return values simplify tracing and verification
Suitability for Large Projects	Poor; doesn't scale; mixing procedural code creates maintenance nightmares; hard to organize multiple operations	Excellent; modular structure supports larger codebases; functions can be organized into modules and package
AI Dependency Risk	High; AI must regenerate entire logic if context changes; procedural code is context-dependent	Lower; function abstraction reduces AI regeneration needs; stable interfaces minimize prompt changes

---

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)

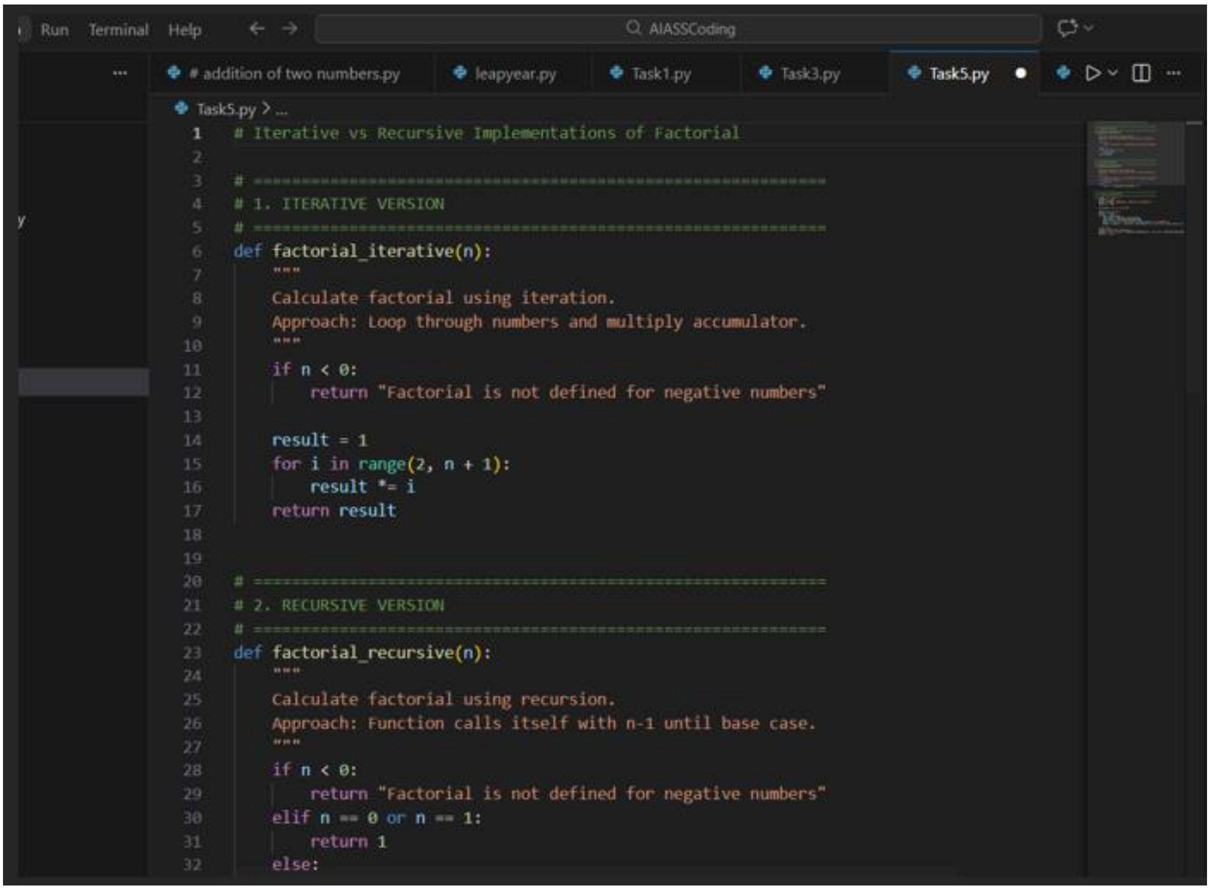
Comparison covering:

 Readability

 Stack usage

- Performance implications
- When recursion is not recommended.

OUTPUTS :



Task5.py > ...

```python
def factorial_recursive(n):
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)


# ============================================================
# 3. TESTING & VERIFICATION
# ============================================================
if __name__ == "__main__":
    print("=" * 60)
    print("FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE")
    print("=" * 60)

    test_cases = [0, 1, 5, 10, 15]

    print("\nTest Cases:")
    print("-" * 60)
    for num in test_cases:
        iter_result = factorial_iterative(num)
        recur_result = factorial_recursive(num)
        match = "✓ MATCH" if iter_result == recur_result else "X MISMATCH"
        print(f"n={num:2d} | Iterative: {iter_result:>18} | Recursive: {recur_result:>18}

    # Edge case
    print(f"\nEdge Case (negative):")
    print(f"n=-5 | Iterative: {factorial_iterative(-5)} | Recursive: {factorial_recursive
    print("=" * 60)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
(.venv) PS D:\AIASSCoding> python Task5.py
============================================================
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
============================================================
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
```

Execution Flow Explanation:

Comparison

Readability :

Iterative: Crystal clear for most developers. A simple loop that anyone can understand instantly. Better for learning loops.

Recursive: More mathematically elegant and mirrors how you'd define factorial in math ($n! = n \times (n-1)!$), but requires understanding function call stacks. Harder for beginners.

Stack Usage :

Iterative: Uses constant memory O(1). Only stores one variable (result). No function call overhead.

Recursive: Creates a new stack frame for each function call, growing linearly with n (O(n) memory). For n=1000, it needs 1000 stack frames—risky and wasteful.

Performance Implications :

Iterative: Fast. No function call overhead. Runs in microseconds even for large n.

Recursive: Slow. Each function call has overhead (10-20x slower per call). For n=1000, the iterative version is orders of magnitude faster.

When Recursion Is NOT Recommended :

1. Large n values – Stack overflow risk; Python's limit is ~1000 calls

2.Performance-critical code – Function call overhead is expensive

3.Simple problems with loops – Unnecessary complexity and slowdown

4.Factorial specifically – Iterative is always better; no benefit from recursion

5.Embedded/resource-limited systems – Limited stack memory

6.When clarity matters – Loops are more intuitive for most people