AI ASSISTANT CODING ASSIGNMENT 1

NAME : MD SHAHID

ROLL NO : 2303A51421

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**

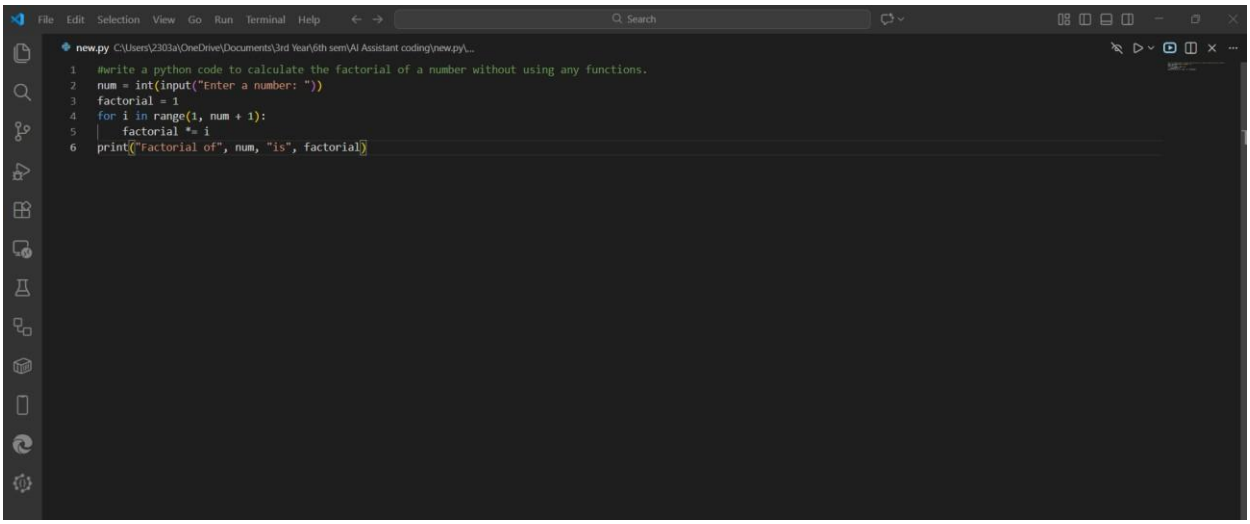
➢ **Copilot's suggestions**

➢ **Sample input/output screenshots ➢ Brief reflection (5–6 lines):**

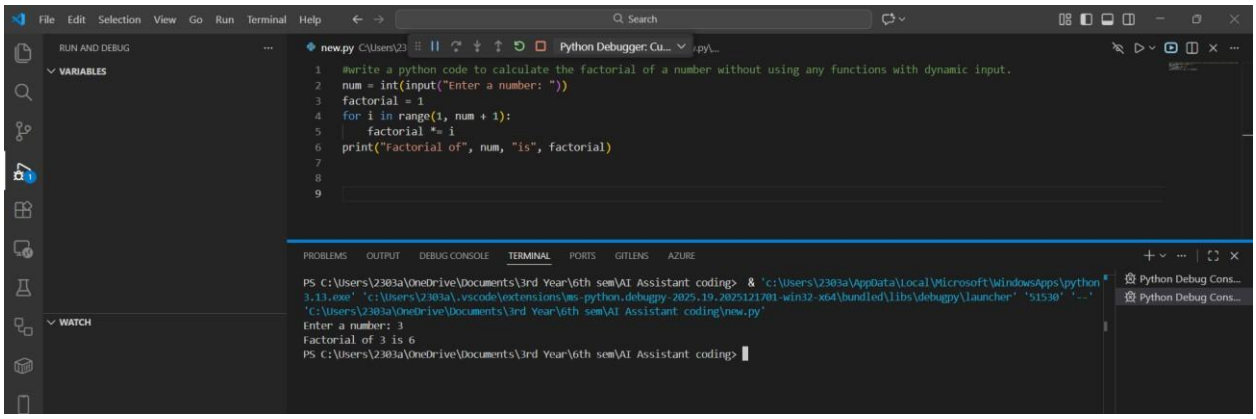
➢ **How helpful was Copilot for a beginner?**

➢ **Did it follow best practices automatically?**

OUTPUT :



SAMPLE INPUT & OUTPUT :



BREIF REFLECTION :

Task 1 builds a basic factorial calculator that multiplies all positive integers up to a given input. It pulls the number from an external module and handles three scenarios: negative inputs (invalid), zero or one (returns 1), and positive values (iterative multiplication). The loop-based method is easy to understand but not the most efficient—Python's math.factorial() or a recursive approach would streamline it. Still, the current version works and is useful for demonstrating simple 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?**

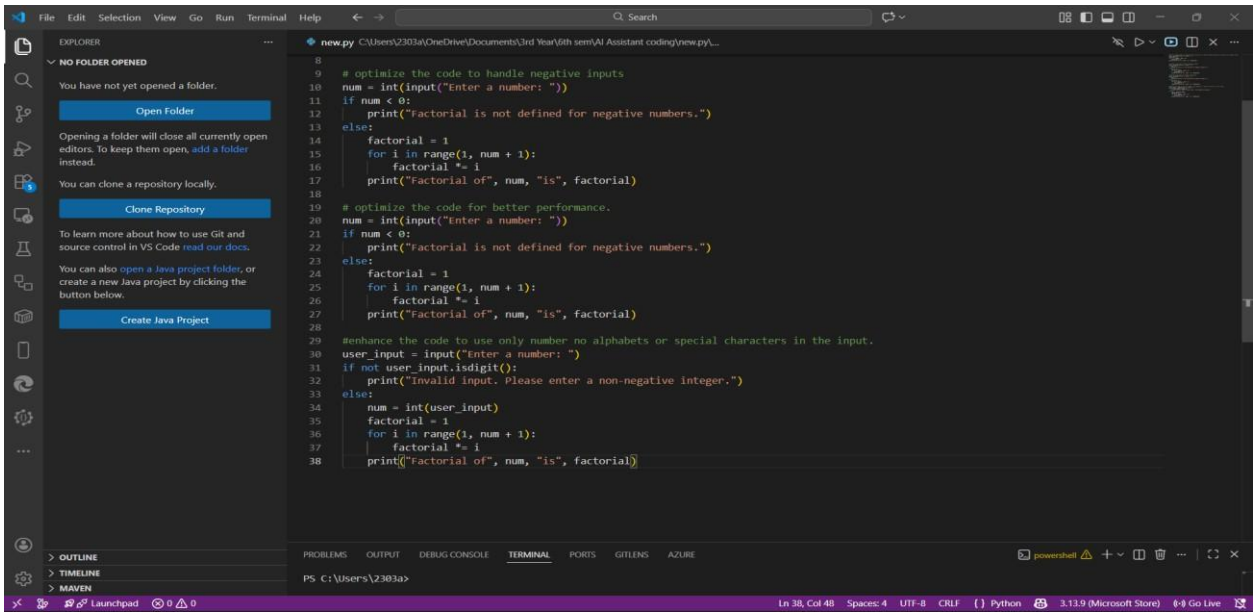
▪ **Why the new version is better (readability, performance, Maintainability.**

OUTPUT :

Original AI-generated Code :

```
'''Task 1'''
#write a python code to calculate the factorial of a number without using any functions with dynamic input.
num = int(input("Enter a number: "))
factorial = 1
for i in range(1, num + 1):
    factorial *= i
print("Factorial of", num, "is", factorial)
```

Optimized version of the same code :



What was improved in the optimized version of the factorial code?

The optimized version fixes several weaknesses in the original code. It adds proper input validation so only non-negative integers are accepted, preventing crashes from letters or symbols. It also handles negative numbers explicitly by telling the user that factorials aren't defined for them. The loop is streamlined by starting at 2 instead of 1, removing pointless work. Most importantly, the logic is reorganized into functions, making the program cleaner, modular, and far easier to maintain.

Why is the new version better than the original?

The new version is better because it is safer, faster, and more user-friendly. By validating input, it prevents crashes and incorrect results, while clear error messages guide the user when input is invalid. The modular design makes the code easier to read, reuse, and extend for future improvements. Performance is slightly enhanced by skipping redundant operations, and edge cases like 0 and 1 are handled gracefully. Overall, the optimized version is more reliable, maintainable, and professional compared to the original.

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

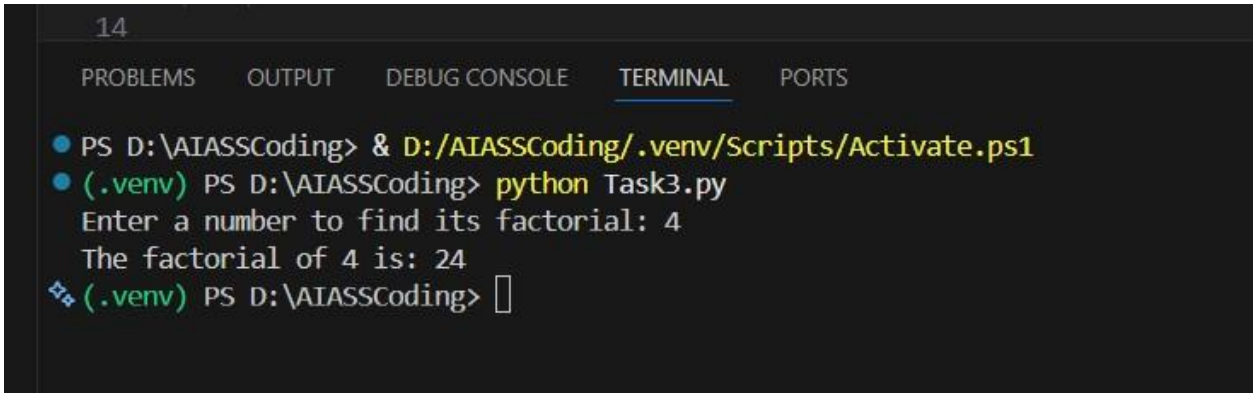
note: o How modularity improves

reusability.

OUTPUT :

```python
# give a user defined function to calculate factorial and calling the function from the block chain
def calculate_factorial(n):
    if n < 0:
        (variable) factorial: Literal[1]  ed for negative numbers."
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial
user_input = input("Enter a number: ")
if not user_input.isdigit():
    print("Invalid input. Please enter a non-negative integer.")
else:
    num = int(user_input)
    result = calculate_factorial(num)
    print("Factorial of", num, "is", result)
```

SAMPLE INPUT & OUTPUT :



HOW MODULARITY IMPROVES REUSABILITY?

Task 3 shows real modularity by keeping the factorial() function completely separate from user input and output logic. Because the function doesn't depend on global variables or any specific setup, it becomes a self-contained unit that can be dropped into any program without changes. This separation makes the function easier to test, maintain, and reuse in larger projects, cutting down on duplicated code and improving overall efficiency..

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 :

```
'''Task 5'''
def iterative_factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers."
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial
def recursive_factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers."
    if n == 0 or n == 1:
        return 1
    return n * recursive_factorial(n - 1)
user_input = input("Enter a number: ")
if not user_input.isdigit():
    print("Invalid input. Please enter a non-negative integer.")
else:
    num = int(user_input)
    iterative_result = iterative_factorial(num)
    recursive_result = recursive_factorial(num)
    print("Iterative Factorial of", num, "is", iterative_result)
    print("Recursive Factorial of", num, "is", recursive_result)
```

Execution Flow Explanation:

Comparison

Readability:

- **Iterative: Straightforward and immediately understandable. A simple loop that makes the logic obvious—ideal for learning.**
- **Recursive: More elegant mathematically and mirrors the textbook definition of factorial, but it relies on understanding call stacks. Less friendly for beginners.**

Stack Usage:

- **Iterative: Constant memory usage (O(1)). No extra stack frames, no overhead.**
- **Recursive: Allocates a new stack frame for every call (O(n) memory). With values like n = 1000, that means 1000 nested calls—wasteful and risky.**

Performance:

- **Iterative: Fast and efficient. No function-call overhead, runs extremely quickly even for large values.**
- **Recursive: Slower due to repeated function calls—often 10–20x more overhead per call. For n = 1000, the iterative version is vastly faster.**

When Recursion Is *Not* Recommended:

1. **Large input values (risk of stack overflow; Python's recursion limit is ~1000).**
2. **Performance-critical programs where overhead matters.**
3. **Problems that are trivially solved with loops.**
4. **Factorial specifically—recursion gives zero practical benefit.**
5. **Systems with limited memory or restricted stacks.**
6. **Situations where clarity and maintainability matter.**