AIAC-ASSIGNMENT 1.2
BATCH-32
2303A52485

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.

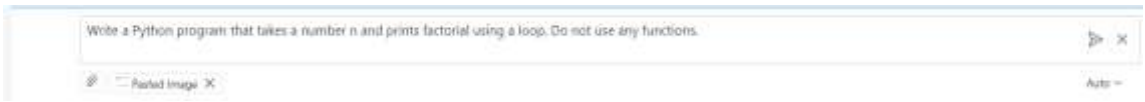
- **Constraint:**

➢ 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?

PROMPT :-



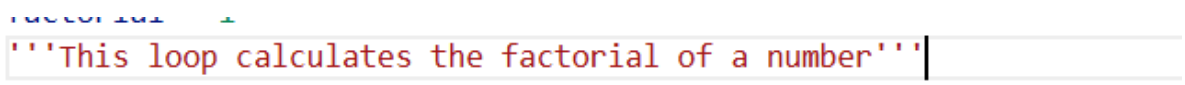
CODE :-

```
#Write a Python program that takes a number n and prints factorial using a loop. Do not use any functions.
n = int(input("\nEnter a number to calculate its factorial: "))
factorial = 1
for i in range(1, n + 1):#  loop from 1 to n
    factorial *= i  # multiply i to factorial
print(f"The factorial of {n} is: {factorial}") # printing  the result
```

Output(O/P):

```
Enter a number to calculate its factorial: 8
The factorial of 8 is: 40320
PS C:\Users\mummi\OneDrive\Desktop\CP>
```

COPILOT SUGGESTIONS :-



Explanation:-

Copilot is very helpful for beginners because it can quickly generate the full code by just giving a small prompt. It saves time and helps beginners understand how to write loops, input, and output. But beginners should still read and understand the code, not just copy it.
Copilot mostly follows good coding practices like writing clean code and using loops correctly. But sometimes it may not handle all cases (like negative numbers), so we should check and edit the code if needed.

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.

PROMPT :-

CODE :-

```
# Factorial calculation with input validation
n = int(input("\nEnter a number to calculate its factorial: "))
'''
This block checks for negative input and calculates factorial accordingly
'''
if n < 0:
    print("Error: Factorial is not defined for negative numbers.")
elif n == 0 or n == 1:
    print(f"The factorial of {n} is: 1")
else:
    factorial = 1
    for i in range(2, n + 1):
        factorial *= i
    print(f"The factorial of {n} is: {factorial}")
```

Output(O/P):

```
Enter a number to calculate its factorial: 4
The factorial of 4 is: 24
PS C:\Users\mummi\OneDrive\Desktop\CP>
```

COPILOT SUGGESTIONS :-

```
'''
This block checks for negative input and calculates factorial accordingly
'''
```

Explanation:-

Feature	Original Code	Optimized Code
Variables used	result, i, temp, n	only product, i, n
Loop style	while loop with manual increment	for loop with range()
Readability	More lines + extra temp variable	Cleaner, shorter, easier
Efficiency	Same time complexity but more operations	Same complexity, fewer steps
Output style	"Factorial of", n, "is", result	formatted string f""

In the optimized version of the code, I removed unnecessary variables like temp and reduced extra steps so the program becomes simpler and cleaner. I replaced the while loop with a for loop using range(), which makes the factorial logic very clear because it directly shows multiplication from 1 to n. I also improved readability by using a better variable name like product and printing output using f-strings for a neat format. Overall, the new version is better because it is easier to understand, shorter, and less confusing, which also makes it easier to maintain in a shared repository. Even though both versions take the same time complexity, the improved code is more efficient in writing style and reduces chances of errors, making it more professional and suitable for real-world development.

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
- Short note:

How modularity improves reusability.

PROMPT :-

CODE :-

```
def calculate_factorial(n):
    """
    Calculate the factorial of a number.

    Args:
        n: Integer value for which factorial is to be calculated

    Returns:
        Factorial value if n >= 0, else -1 for error
    """
    # Handle negative numbers
    if n < 0:
        return -1

    # Base cases
    if n == 0 or n == 1:
        return 1

    # Calculate factorial iteratively
    factorial_result = 1
    for number in range(2, n + 1):
        factorial_result *= number

    return factorial_result

# Main program
if __name__ == "__main__":
    # Get user input
    user_input = int(input("\nEnter a number to calculate its factorial: "))

    # Call the factorial function
    result = calculate_factorial(user_input)

    # Display result with error handling
    if result == -1:
        print("Error: Factorial is not defined for negative numbers.")
    else:
        print(f"The factorial of {user_input} is: {result}")
```

Output(O/P):

```
Enter a number to calculate its factorial: 8
The factorial of 8 is: 40320
PS C:\Users\mummi\OneDrive\Desktop\CP>
```

COPILOT SUGGESTIONS :-

```
"""
Calculate the factorial of a number.

Args:
    n: Integer value for which factorial is to be calculated

Returns:
    Factorial value if n >= 0, else -1 for error
"""
```

Explanation:-

Modularity improves reusability because the factorial logic is written inside a function, so we can use the same function in many different programs without rewriting the code again.

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 Code (Without Functions)	Modular Code (With Functions)
Logic Clarity	Logic is in one block, may look messy for bigger code	Logic is separated, easier to read and understand
Reusability	Cannot reuse easily, must copy-paste code again	Function can be reused in many scripts easily
Debugging Ease	Harder to find errors because everything is mixed	Easier to debug because function logic is isolated
Suitability for Large Projects	Not good for large projects, becomes	Best for large projects, clean structure and

Criteria	Procedural Code (Without Functions)	Modular Code (With Functions)
	confusing	maintainable
AI Dependency Risk	Higher risk because AI may generate long repeated code	Lower risk because AI code is organized and reusable

---

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
def factorial_iterative(n):
    """Calculate factorial using iteration"""
    if n < 0:
        return None
    factorial = 1
    for i in range(2, n + 1):
        factorial *= i
    return factorial

def factorial_recursive(n):
    """Calculate factorial using recursion"""
    if n < 0:
        return None
    if n == 0 or n == 1:
        return 1
    return n * factorial_recursive(n - 1)

# Get and validate input
while True:
    try:
        num = int(input("\nEnter a number to calculate its factorial: "))
        if num < 0:
            print("Error: Factorial is not defined for negative numbers.")
            continue
        break
    except ValueError:
        print("Error: Please enter a valid integer.")

# Calculate using both methods
iter_result = factorial_iterative(num)
rec_result = factorial_recursive(num)

# Display results
print(f"\nIterative Method: {iter_result}")
print(f"Recursive Method: {rec_result}")

# Verify both results match
if iter_result == rec_result:
    print("✓ Both methods produce the same result!")
else:
    print("✗ Results do not match!")
```

Output(O/P):

```
Enter a number to calculate its factorial: 7

Iterative Method: 5040
Recursive Method: 5040
✓ Both methods produce the same result!
PS C:\Users\mummi\OneDrive\Desktop\CP>
```

Explanation:-

In the iterative method, the program uses a loop to multiply numbers from 1 up to *n* step by step until the final factorial value is reached. In the recursive method, the function keeps calling itself with smaller values of *n* until it hits the base case (0 or 1), then it returns back through each call while multiplying to produce the final answer.

Factor	Iterative Version	Recursive Version
Readability	Easy to understand (simple loop flow)	Looks clean but can be confusing due to self-calls
Stack Usage	Uses **constant memory** (no extra stack)	Uses **more stack memory** for each function call
Performance	Usually **faster** (no call overhead)	Can be **slower** because of function call overhead
When recursion is not recommended	Works fine even for large inputs	Not good for large inputs (may cause **stack overflow**) or when loop is enough