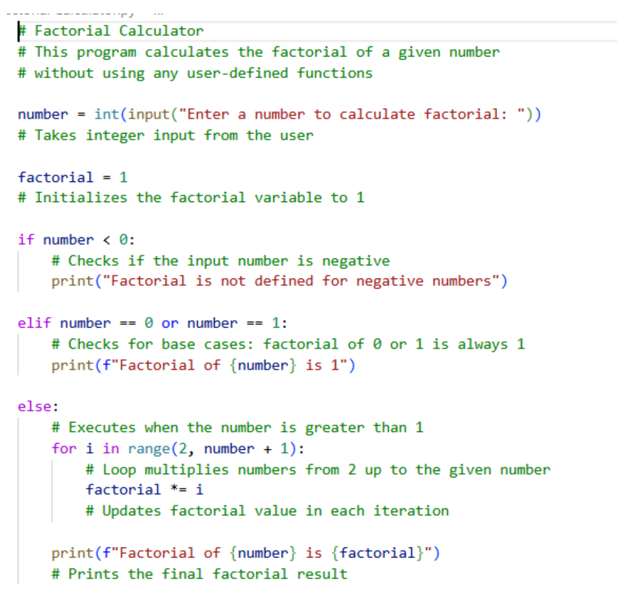
## Task 1: AI-Generated Logic Without Modularization (Factorial without Functions):

### Given prompt :

Write a Python program to calculate factorial of a number
Do not use any user-defined functions
Use loops and variables only
Implement logic directly in the main program flow

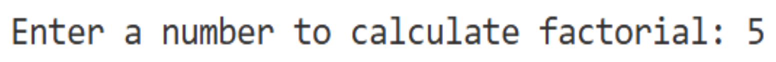
### Code suggested:



```python
# Factorial Calculator
# This program calculates the factorial of a given number
# without using any user-defined functions

number = int(input("Enter a number to calculate factorial: "))
# Takes integer input from the user

factorial = 1
# Initializes the factorial variable to 1

if number < 0:
    # Checks if the input number is negative
    print("Factorial is not defined for negative numbers")

elif number == 0 or number == 1:
    # Checks for base cases: factorial of 0 or 1 is always 1
    print(f"Factorial of {number} is 1")

else:
    # Executes when the number is greater than 1
    for i in range(2, number + 1):
        # Loop multiplies numbers from 2 up to the given number
        factorial *= i
        # Updates factorial value in each iteration

    print(f"Factorial of {number} is {factorial}")
    # Prints the final factorial result
```

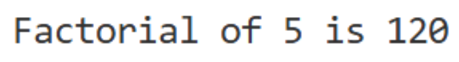
### Sample Input / Output

### Input:



```
Enter a number to calculate factorial: 5
```

### Output:



```
Factorial of 5 is 120
```

### Reflection (5–6 Lines):

1. GitHub Copilot generated correct and efficient logic based on simple prompts.
2. It handled edge cases like negative numbers and zero automatically.
3. The AI reduced coding time significantly for beginners.
4. However, it does not optimize structure unless explicitly prompted.

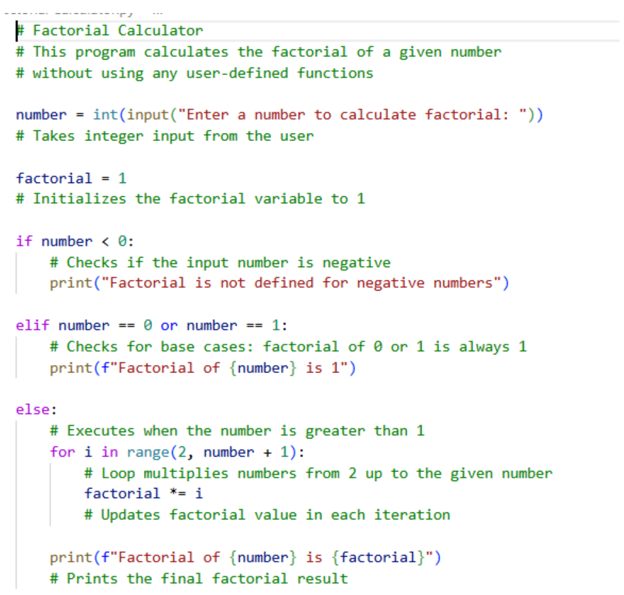
5. Understanding the generated code is essential before using it in projects.
6. **Copilot** is best used as an assistant, not a replacement for logic building.
7. Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

## <u>Task 2: AI Code Optimization & Cleanup (Improving Efficiency):</u>

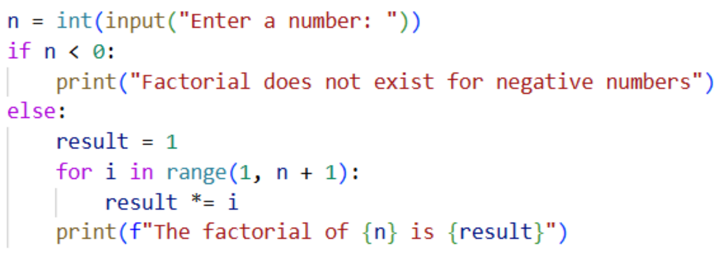
### <u>Givenprompt:</u>

Simplify this factorial program and make it more readable and efficient without using functions.

### <u>Code Suggested & Original Code:</u>

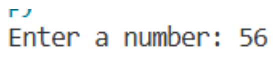


```
# Factorial Calculator
# This program calculates the factorial of a given number
# without using any user-defined functions

number = int(input("Enter a number to calculate factorial: "))
# Takes integer input from the user

factorial = 1
# Initializes the factorial variable to 1

if number < 0:
    # Checks if the input number is negative
    print("Factorial is not defined for negative numbers")

elif number == 0 or number == 1:
    # Checks for base cases: factorial of 0 or 1 is always 1
    print(f"Factorial of {number} is 1")

else:
    # Executes when the number is greater than 1
    for i in range(2, number + 1):
        # Loop multiplies numbers from 2 up to the given number
        factorial *= i
        # Updates factorial value in each iteration

    print(f"Factorial of {number} is {factorial}")
    # Prints the final factorial result
```



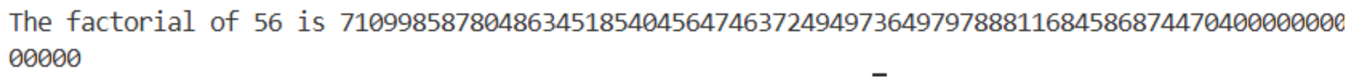
```
n = int(input("Enter a number: "))
if n < 0:
    print("Factorial does not exist for negative numbers")
else:
    result = 1
    for i in range(1, n + 1):
        result *= i
    print(f"The factorial of {n} is {result}")
```

### <u>Sample Input:-</u>



```
Enter a number: 56
```

### <u>Sample output:-</u>



```
The factorial of 56 is 71099858780486345185404564746372494973649797888116845868744704000000000
00000
```

### <u>Written explanation</u> :

**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?**

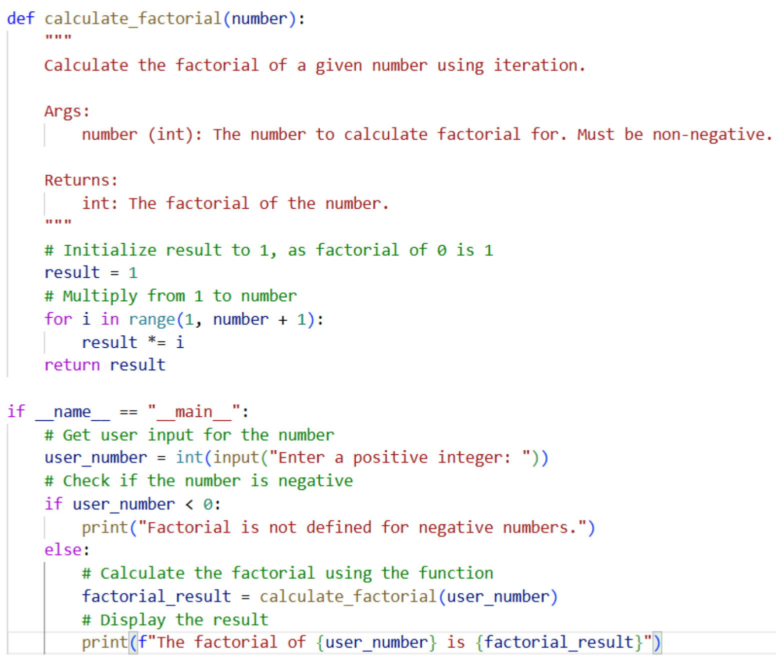
- **Readability:** The code is shorter and easier to understand at a glance
- **Performance:** Fewer conditional checks make the code slightly more efficient
- **Maintainability:** Cleaner structure makes it easier to modify or review later
- **Best Practices:** Uses modern Python features and avoids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

**<u>Task 3: Modular Design Using AI Assistance (Factorial with Functions)</u>:**

**<u>Given prompt</u> :**

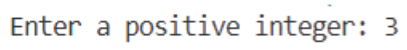
Write a Python program to calculate factorial using a user-defined function. Use meaningful function and variable names, include inline comments, and call the function from the main block.

**<u>Suggest Code</u>:**



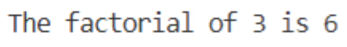
```python
def calculate_factorial(number):
    """
    Calculate the factorial of a given number using iteration.

    Args:
        number (int): The number to calculate factorial for. Must be non-negative.

    Returns:
        int: The factorial of the number.
    """
    # Initialize result to 1, as factorial of 0 is 1
    result = 1
    # Multiply from 1 to number
    for i in range(1, number + 1):
        result *= i
    return result

if __name__ == "__main__":
    # Get user input for the number
    user_number = int(input("Enter a positive integer: "))
    # Check if the number is negative
    if user_number < 0:
        print("Factorial is not defined for negative numbers.")
    else:
        # Calculate the factorial using the function
        factorial_result = calculate_factorial(user_number)
        # Display the result
        print(f"The factorial of {user_number} is {factorial_result}")
```

**<u>Sample inputs</u>:**



```
Enter a positive integer: 3
```

**<u>Sample outputs</u>:**



```
The factorial of 3 is 6
```

**<u>Short note</u>:**

**How modularity improves reusability.**

Modularity allows the factorial logic to be reused in multiple programs without rewriting code.

The function can be imported into other scripts, improving consistency and reducing errors. It also makes the code easier to test, debug, and maintain.

## Task 4: Comparative Analysis – Procedural vs Modular AI Code (With vs Without Functions)

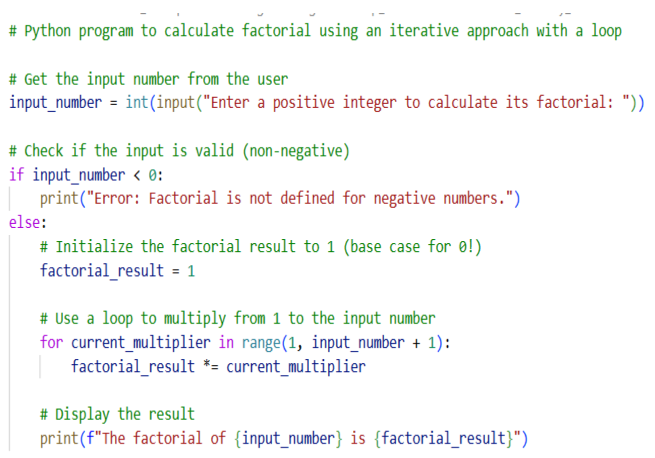
| **Criteria** | **Procedural Code (Without Functions)** | **Modular Code (With Functions)** |
|---|---|---|
| **Logic Clarity** | Logic is written in a single block, which is easy for small programs but harder to follow as code grows | Logic is divided into functions, making the program flow clearer and more organized |
| **Reusability** | Code cannot be reused directly and must be rewritten in other scripts | Functions can be reused across multiple programs without duplication |
| **Debugging Ease** | ~~Debugging is difficult because all logic is mixed together~~ | Easier debugging since errors can be isolated within a function |
| **Suitability for Large Projects** | Not suitable for large projects due to poor structure and scalability | Well-suited for large projects with clean and maintainable design |
| **AI Dependency Risk** | Higher risk, as developers may rely on AI-generated blocks without understanding them | Lower risk, since modular code encourages understanding, testing, and reuse |

## Task 5: AI-Generated Iterative vs Recursive Thinking

### Given Prompt:

Generate a Python program to calculate factorial using an iterative approach with a loop and clear variable names.

### Suggested Code:



```
# Python program to calculate factorial using an iterative approach with a loop

# Get the input number from the user
input_number = int(input("Enter a positive integer to calculate its factorial: "))

# Check if the input is valid (non-negative)
if input_number < 0:
    print("Error: Factorial is not defined for negative numbers.")
else:
    # Initialize the factorial result to 1 (base case for 0!)
    factorial_result = 1

    # Use a loop to multiply from 1 to the input number
    for current_multiplier in range(1, input_number + 1):
        factorial_result *= current_multiplier

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

**Sample Input:**

```
Enter a positive integer to calculate its factorial: 5
```

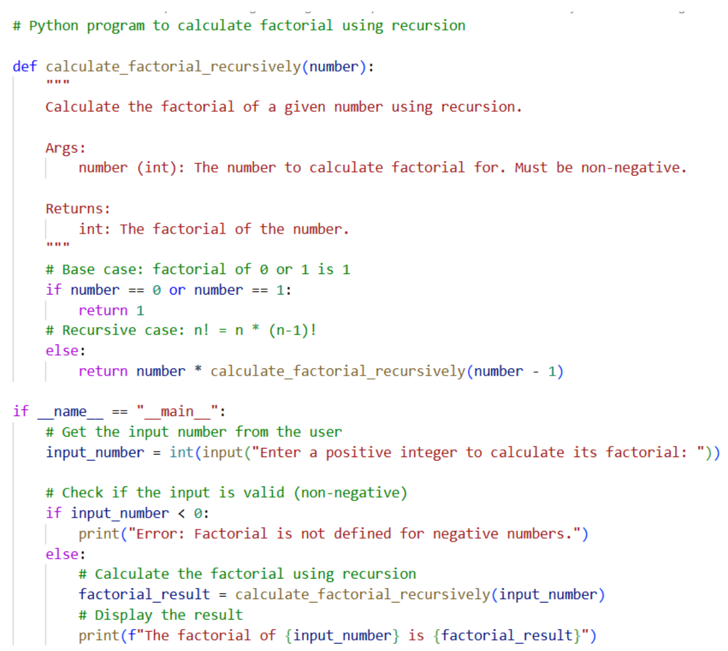
**Sample output:**

```
The factorial of 5 is 120
```

**Given Code:**

Generate a Python program to calculate factorial using recursion with proper base cases and comments.

**Suggested Code :**



```python
# Python program to calculate factorial using recursion

def calculate_factorial_recursively(number):
    """
    Calculate the factorial of a given number using recursion.

    Args:
        number (int): The number to calculate factorial for. Must be non-negative.

    Returns:
        int: The factorial of the number.
    """
    # Base case: factorial of 0 or 1 is 1
    if number == 0 or number == 1:
        return 1
    # Recursive case: n! = n * (n-1)!
    else:
        return number * calculate_factorial_recursively(number - 1)

if __name__ == "__main__":
    # Get the input number from the user
    input_number = int(input("Enter a positive integer to calculate its factorial: "))

    # Check if the input is valid (non-negative)
    if input_number < 0:
        print("Error: Factorial is not defined for negative numbers.")
    else:
        # Calculate the factorial using recursion
        factorial_result = calculate_factorial_recursively(input_number)
        # Display the result
        print(f"The factorial of {input_number} is {factorial_result}")
```

**Sample Input:**

```
Enter a positive integer to calculate its factorial: 34
```

**Sample output:**

```
The factorial of 34 is 295232799039604140847618609643520000000
```

**Execution Flow Explanation :**

- In the iterative approach, the program uses a loop to multiply numbers from 1 to n, updating the result step by step.

- In the recursive approach, the function calls itself with a smaller value until it reaches the base case (n = 0 or 1), then returns the result back through the call stack.
- Both methods follow different thinking styles but compute the same final result.

## **Comparison: Iterative vs Recursive**

| Aspect | Iterative Approach | Recursive Approach |
|---|---|---|
| Readability | Easier for beginners to understand | Cleaner and mathematically expressive |
| Stack Usage | Uses constant memory | Uses call stack for each function call |
| Performance | More efficient, no overhead | Slower due to repeated function calls |
| When Recursion Is Not Recommended | | For large inputs (may cause stack overflow) |