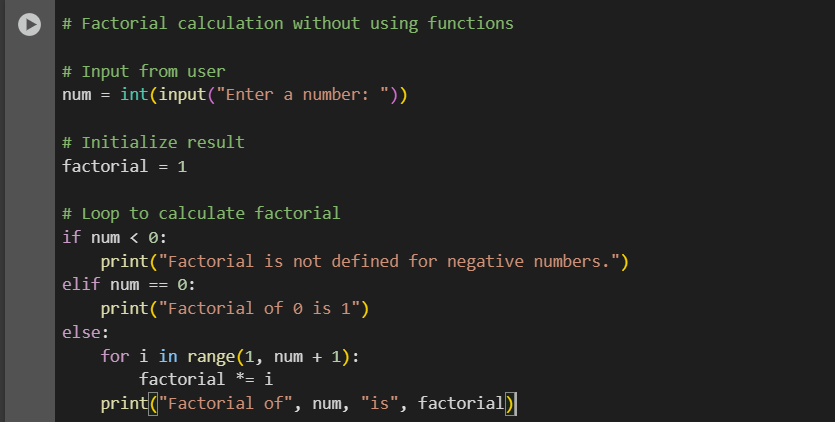
Factorial without Functions  
● Description:  
Use GitHub Copilot to generate a Python program that calculates the  
factorial of a number without defining any functions (using loops  
directly in the main code)

Code:



Output:



Explanation:

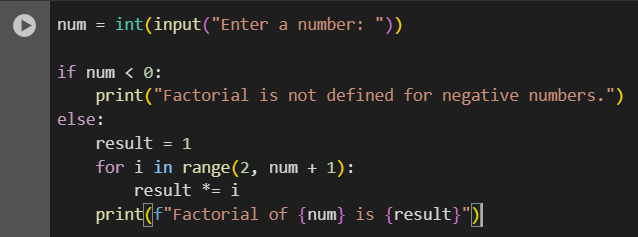
# This program calculates the factorial of a number using a loop. # It does not use any functions—everything is written directly in the main code block. # Step 1: Take input from the user num = int(input("Enter a number: ")) # Step 2: Initialize a variable to store the result factorial = 1 # Step 3: Handle special cases # If the number is negative, factorial is not defined if num < 0: print("Factorial is not defined for negative numbers.") # If the number is 0, factorial is 1 by definition elif num == 0: print("Factorial of 0 is 1") # Step 4: Use a loop to multiply numbers from 1 to num else: for i in range(1, num + 1): factorial \*= i print("Factorial of", num, "is", factorial)

Task2:

Improving Efficiency  
● Description:  
Examine the Copilot-generated code from Task 1 and demonstrate  
how its efficiency can be improved (e.g., removing unnecessary

variables, optimizing loops).

Code:



Output:



Explanation:

Removed the special case for num == 0 since the loop handles it naturally.

• Started the loop from 2 instead of 1 to avoid unnecessary multiplication by 1.

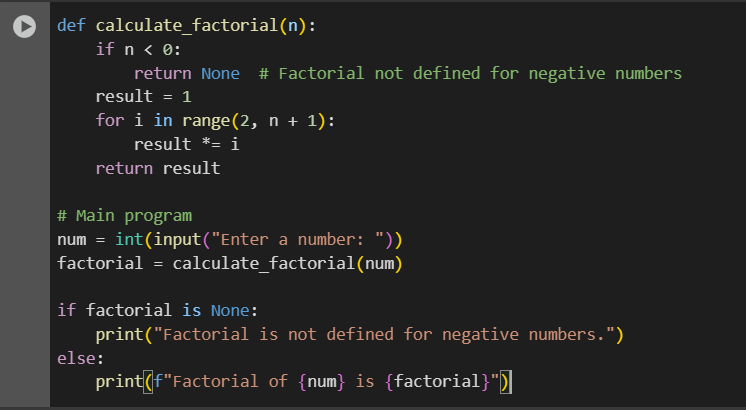
• Reduced the number of conditional branches for cleaner logic.

• Used f-string formatting for more concise and readable output.

Task3:

Factorial with Functions  
● Description:  
Use GitHub Copilot to generate a Python program that calculates the  
factorial of a number using a user-defined function

Code:



Output:



Explanation:

This program defines a function called calculate\_factorial that takes an integer as input and returns its factorial. The function uses a loop to multiply numbers from 2 up to the input value. The main program handles user input and prints the result. This approach improves modularity and makes the code reusable in other projects.

Task 4: Comparative Analysis – With vs Without Functions  
● Description:  
Differentiate between the Copilot-generated factorial program with  
functions and without functions in terms of logic, reusability, and  
execution.  
● Expected Output:  
o A comparison table or short report explaining the differences

Table:

+-------------------+-----------------------------------------------+-------------------------------------------------------------+

| Aspect | Without Functions | With Functions |

+-------------------+-----------------------------------------------+-------------------------------------------------------------+

| Logic | All logic is written in the main block. | Logic is encapsulated in a reusable function. |

| Reusability | Low – cannot reuse the factorial logic. | High – function can be reused in other programs. |

| Execution | Direct execution; harder to test or extend. | Cleaner flow; easier to test and integrate. |

| Modularity | Poor – everything is in one place. | Strong – separates logic from input/output. |

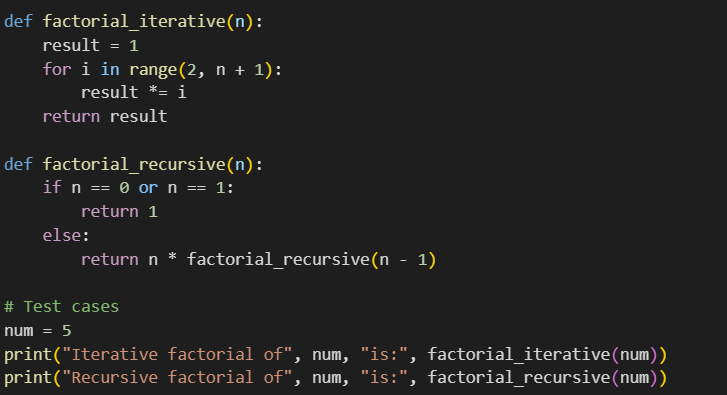
| Maintainability | Harder to update or debug. | Easier to maintain and modify. |

| Readability | Less readable for larger programs. | More readable and organized. |

|  |  |  |
| --- | --- | --- |
|  |  |  |

Task 5: Iterative vs Recursive Factorial  
● Description:  
Prompt GitHub Copilot to generate both iterative and recursive  
versions of the factorial function.  
● Expected Output:  
o Two correct implementations.  
o A documented comparison of logic, performance, and  
execution flow between iterative and recursive approaches

Code:



Output:



Table:

+----------------------+------------------------------+----------------------------------+

| Aspect | Iterative Approach | Recursive Approach |

+----------------------+------------------------------+----------------------------------+

| Logic | Uses a loop to multiply | Calls itself with n-1 until base |

| | values from 2 to n | case is reached (n == 0 or 1) |

| Performance | Faster, uses less memory | Slower for large n due to call |

| | | stack overhead |

| Execution Flow | Linear, predictable | Branching, stack-based |

| Readability | Simple for beginners | Elegant and concise |

| Stack Usage | Constant | Grows with n (risk of overflow) |

| Tail Call Optimization| Not needed | Not supported in Python |

| Debugging Ease | Easier to trace | Harder to trace deep recursion |

| Use Case | Preferred for large n | Good for small n or teaching |

+----------------------+------------------------------+----------------------------------+