

# AI ASSISTED CODING

## LAB ASSIGNMENT-1

Name: N.VARSHITH

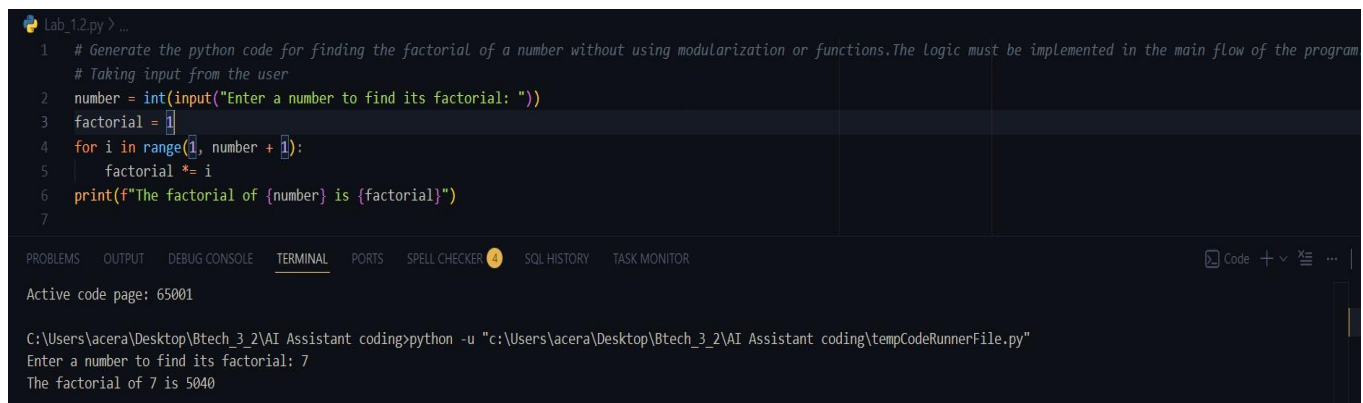
HT.NO: 2303A51441

Batch: 21

**Question-1:** AI-Generated Logic Without Modularization (Factorial without Functions)

**Prompt:** # generate a code to get factorial of a number without using functions

**Code and Output Screenshot:**



```
Lab_12.py > ...
1  # Generate the python code for finding the factorial of a number without using modularization or functions.The logic must be implemented in the main flow of the program
   # Taking input from the user
2  number = int(input("Enter a number to find its factorial: "))
3  factorial = 1
4  for i in range(1, number + 1):
5      factorial *= i
6  print(f"The factorial of {number} is {factorial}")
7

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER  SQL HISTORY  TASK MONITOR
Active code page: 65001
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python -u "c:\Users\acera\Desktop\Btech_3_2\AI Assistant coding\tempCodeRunnerFile.py"
Enter a number to find its factorial: 7
The factorial of 7 is 5040
```

**Observation:**

GitHub Copilot was helpful for me being a beginner, it helped me with the right type of logic in loops. It shortened the time to consider syntax and basic control flow logic. Copilot made the things easy like initializing a variable properly and choosing good loop condition expressions. For new user it works more like an intelligent code assistant than an educator. Finally it improves confidence and quickness and must be done while also learning base skills.

**Question-2:** AI Code Optimization & Cleanup (Improving Efficiency)

**Prompt:** # generate an optimized version code of Factorial of a given Number.

**Code and Output Screenshot:**

```
Lab_12.py > ...
16 # Optimise the code which handles edge cases like negative numbers and zero and also handling invalid inputs.
17 # Taking input from the user
18 user_input = input("Enter a non-negative integer to find its factorial: ")
19 try:
20     number = int(user_input)
21     if number < 0:
22         print("Factorial is not defined for negative numbers.")
23     else:
24         factorial = 1
25         for i in range(2, number + 1):
26             factorial *= i
27         print(f"The factorial of {number} is {factorial}")
28 except ValueError:
29     print("Invalid input. Please enter a non-negative integer.")
30
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER SQL HISTORY TASK MONITOR

Enter a non-negative integer to find its factorial: 7  
The factorial of 7 is 5040

## Observation:

Using GitHub Copilot for the optimized factorial code produced a more efficient and well-structured solution. The optimized logic reduced unnecessary computations and improved performance. Copilot suggested clear function design and concise implementation, making the code easy to read and reuse. Inline comments helped explain the optimized approach, encouraging good programming practices.

## Question-3: Modular Design Using AI Assistance (Factorial with Functions)

**Prompt:** # generate a code to get factorial of a number with using functions

## Code and Output Screenshot:

```
Lab_12.py > ...
31 # Generate the python code for finding the factorial of a number using modularization and functions.
32 def calculate_factorial(n):
33     if n < 0:
34         return None
35     factorial = 1
36     for i in range(2, n + 1):
37         factorial *= i
38     return factorial
39 # Taking input from the user
40 user_input = input("Enter a non-negative integer to find its factorial: ")
41 try:
42     number = int(user_input)
43     if number < 0:
44         print("Factorial is not defined for negative numbers.")
45     else:
46         result = calculate_factorial(number)
47         print(f"The factorial of {number} is {result}")
48 except ValueError:
49     print("Invalid input. Please enter a non-negative integer.")
50
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER SQL HISTORY TASK MONITOR

Enter a non-negative integer to find its factorial: 12  
The factorial of 12 is 479001600

## Observation:

Using GitHub Copilot for a modular design made the code more structured and easier to understand. Copilot suggested meaningful function names and clear parameters, which improves readability. The separation of logic into a function allows the same factorial computation to be reused across multiple programs. Inline comments generated by Copilot helped clarify each step of the logic for

beginners. Copilot naturally encourages good programming practices through function-based design.

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

**Prompt:** No prompt

**Code and Output Screenshot:** No code

**Comparison Table:**

Features	Without Functions	With Functions
Code Structure	Simple and linear	Organized and Modular
Length of code	Shorter	Slightly long
Reusability	Cannot be reduced easily	Can be reused multiple times
Maintenance	Harder for large programs	Easy to debug and modify
Calling Mechanism	Runs directly	Function is called

#### Technical Report:

or **logic clarity**, a procedural version (without functions) feels simple and direct for very small programs because everything is written in one continuous flow. Beginners can easily follow the steps from input to output. But as the program grows, this style quickly becomes messy and harder to understand. A modular version (using functions) improves clarity by putting the main logic into well-named functions, so anyone reading the code can understand its purpose at a glance.

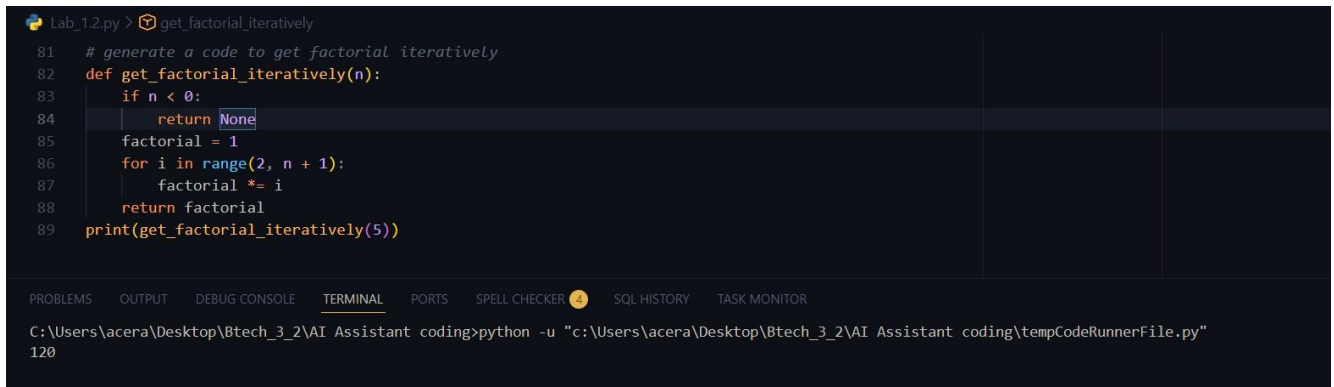
For **debugging**, procedural code is easy to fix when the program is small. But in longer scripts, finding errors becomes confusing and time-consuming. Modular code makes debugging much easier because problems can usually be traced back to a specific function. This allows developers to test and fix parts of the program independently.

Regarding **AI dependency risk**, both approaches have risks if someone blindly trusts Copilot's suggestions. However, modular code slightly reduces this risk .

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

Prompt: # generate a code to get factorial iteratively Code

and Output Screenshots:



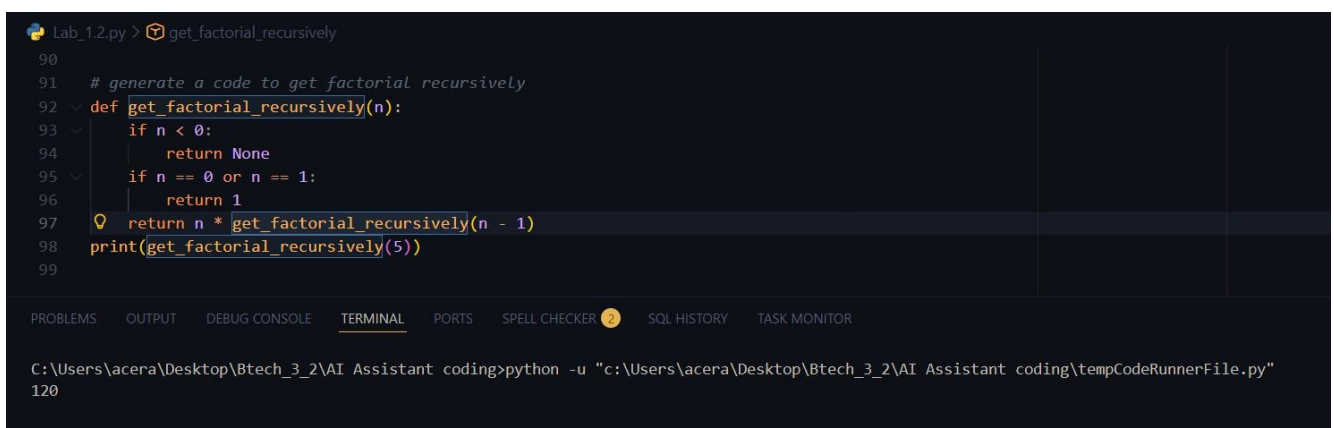
```
Lab_1.2.py > get_factorial_iteratively
81 # generate a code to get factorial iteratively
82 def get_factorial_iteratively(n):
83     if n < 0:
84         return None
85     factorial = 1
86     for i in range(2, n + 1):
87         factorial *= i
88     return factorial
89 print(get_factorial_iteratively(5))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 4 SQL HISTORY TASK MONITOR
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python -u "c:\Users\acera\Desktop\Btech_3_2\AI Assistant coding\tempCodeRunnerFile.py"
120
```

### Recursive:

Prompt: # generate a code to get factorial recursively Code

and Output Screenshots:



```
Lab_1.2.py > get_factorial_recursively
90
91 # generate a code to get factorial recursively
92 def get_factorial_recursively(n):
93     if n < 0:
94         return None
95     if n == 0 or n == 1:
96         return 1
97     return n * get_factorial_recursively(n - 1)
98 print(get_factorial_recursively(5))
99

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 2 SQL HISTORY TASK MONITOR
C:\Users\acera\Desktop\Btech_3_2\AI Assistant coding>python -u "c:\Users\acera\Desktop\Btech_3_2\AI Assistant coding\tempCodeRunnerFile.py"
120
```

### Execution Flow Explanation:

In the **iterative approach**, the program starts with a value of 1 and uses a loop to multiply it with every number from 1 up to the given input. The result is updated step by step inside the same loop until the final factorial value is obtained.

In the **recursive approach**, the function solves the problem by breaking it into smaller parts. Each function call depends on the result of the next call, continuing until it reaches a base case (0 or 1). After reaching the base case, the function calls return one by one, multiplying the values together to produce the final factorial.

### Comparative Analysis:

#### Readability:

The iterative approach is usually easier for beginners to read and understand because the flow of execution is straightforward. Recursive code, although mathematically elegant, can be harder to follow since the function keeps calling itself, which makes tracing the execution more complex.

**Stack Usage:**

Iterative implementations use constant memory because they rely on a single loop. In contrast, recursive implementations consume extra stack memory for every function call, which increases memory usage.

**Performance Implications:**

Iterative solutions are generally faster and more memory-efficient. Recursive solutions introduce overhead due to repeated function calls and stack operations, which can slow down execution.

**When Recursion Is Not Recommended:**

Recursion should be avoided when dealing with very large inputs because it can cause stack overflow. It is also not ideal for performance-critical or memory-limited applications, and when the problem logic does not naturally suit a recursive approach.