AI ASSISTED CODING

LAB ASSIGNMENT-1

Name: Kushal Mandal

HT.NO: 2303A51621

Batch: 22

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:

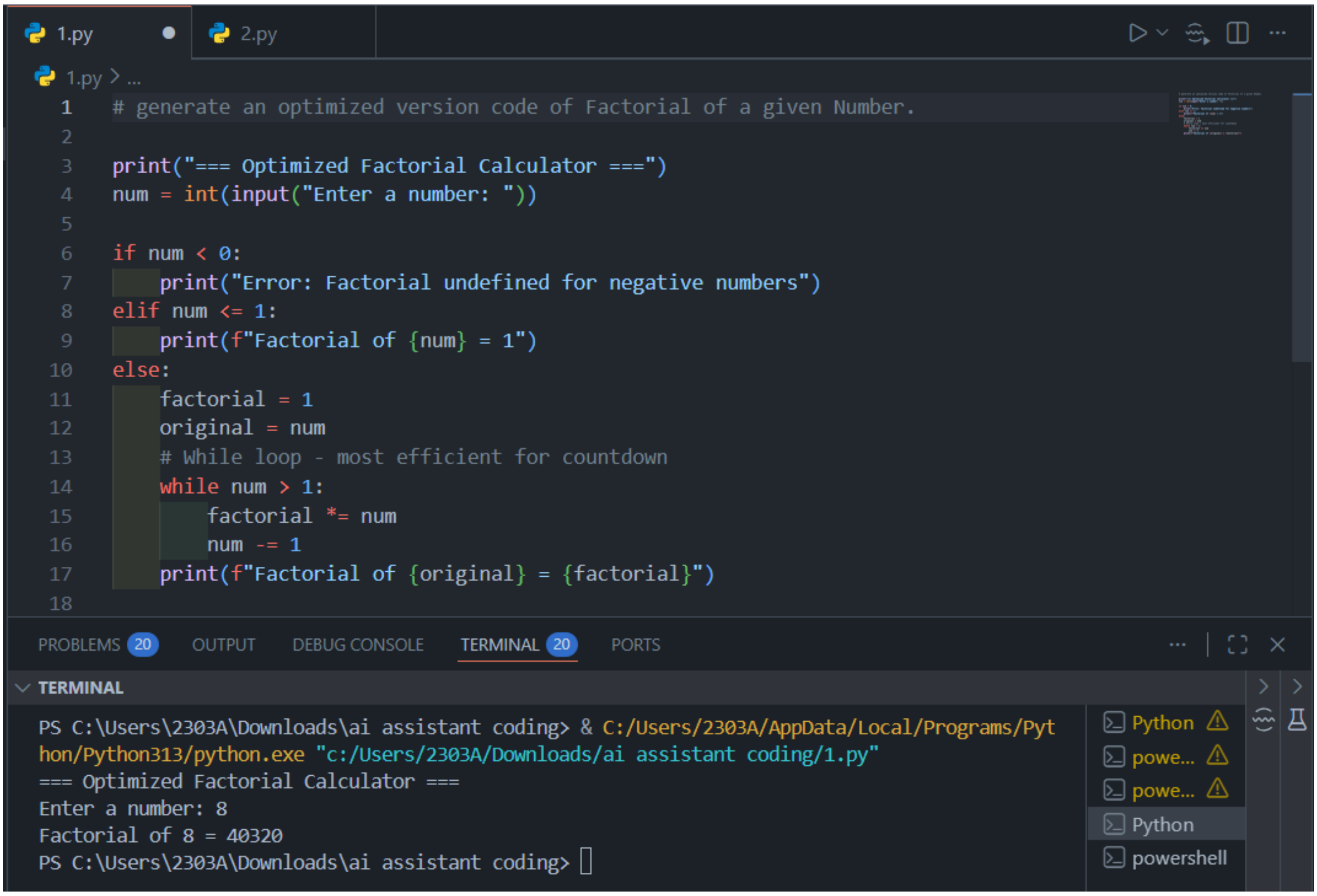
Observation:

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:



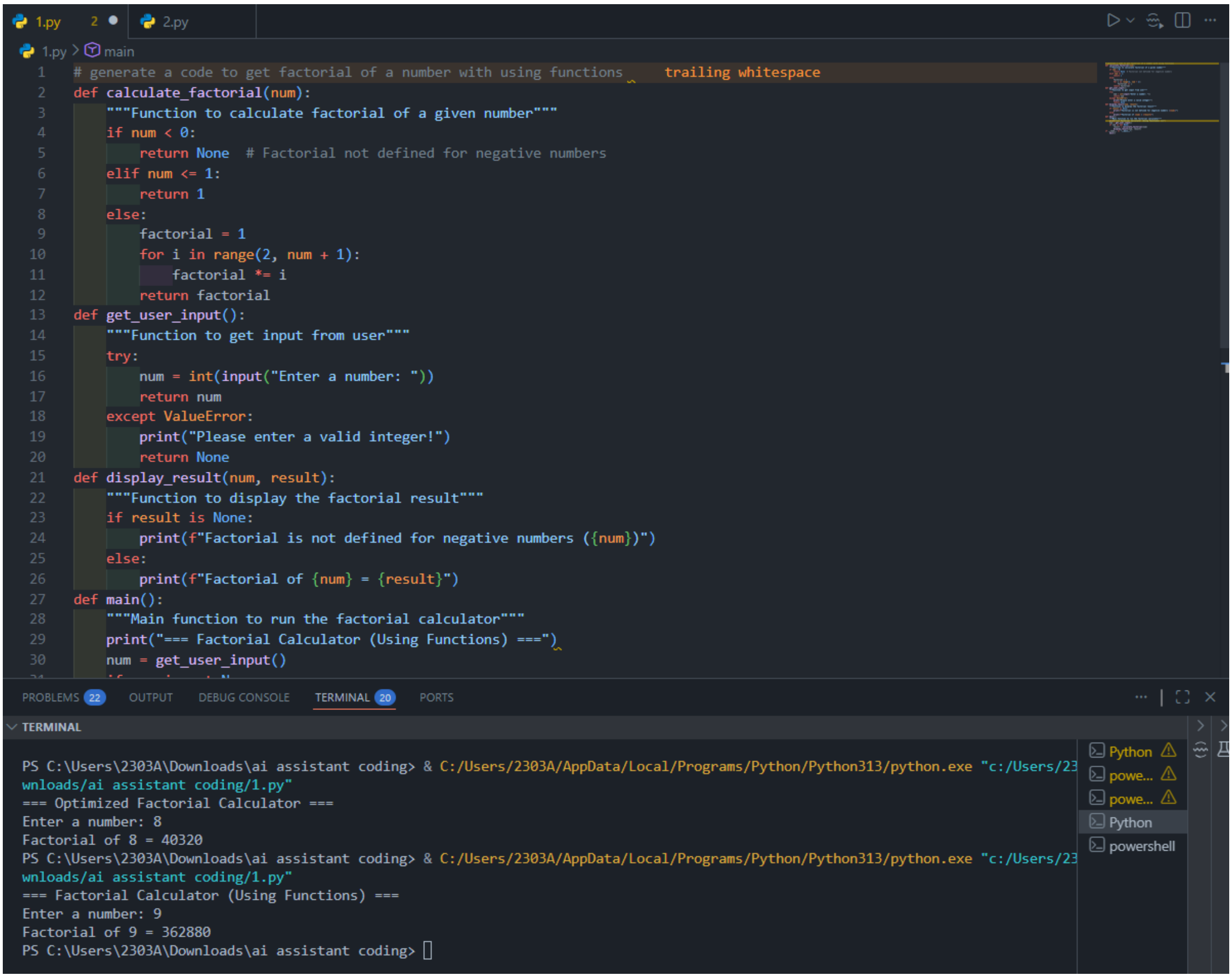
Observation:

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:



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.

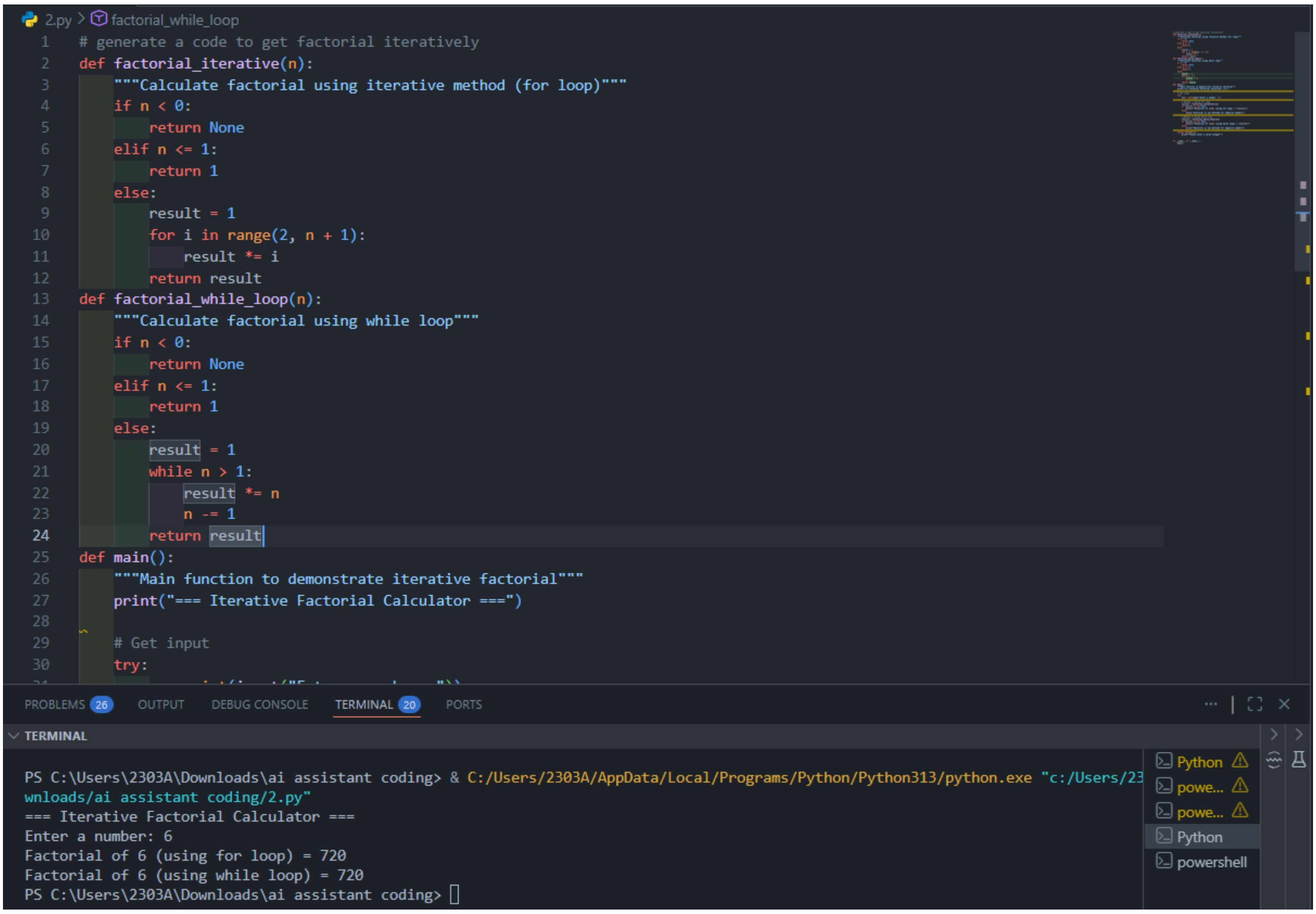
Regarding **AI dependency risk**, both approaches have risks if someone blindly trusts Copilot's suggestions. However, modular code slightly reduces this risk .

Question-5: AI-Generated Iterative vs Recursive Thinking

Iterative:

Prompt: # generate a code to get factorial iteratively

Code and Output Screenshots:



2.py > factorial_while_loop

```python
# generate a code to get factorial iteratively
def factorial_iterative(n):
    """Calculate factorial using iterative method (for loop)"""
    if n < 0:
        return None
    elif n <= 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result
def factorial_while_loop(n):
    """Calculate factorial using while loop"""
    if n < 0:
        return None
    elif n <= 1:
        return 1
    else:
        result = 1
        while n > 1:
            result *= n
            n -= 1
        return result
def main():
    """Main function to demonstrate iterative factorial"""
    print("=== Iterative Factorial Calculator ===")

    # Get input
    try:
```

PROBLEMS 26 OUTPUT DEBUG CONSOLE TERMINAL 20 PORTS

TERMINAL

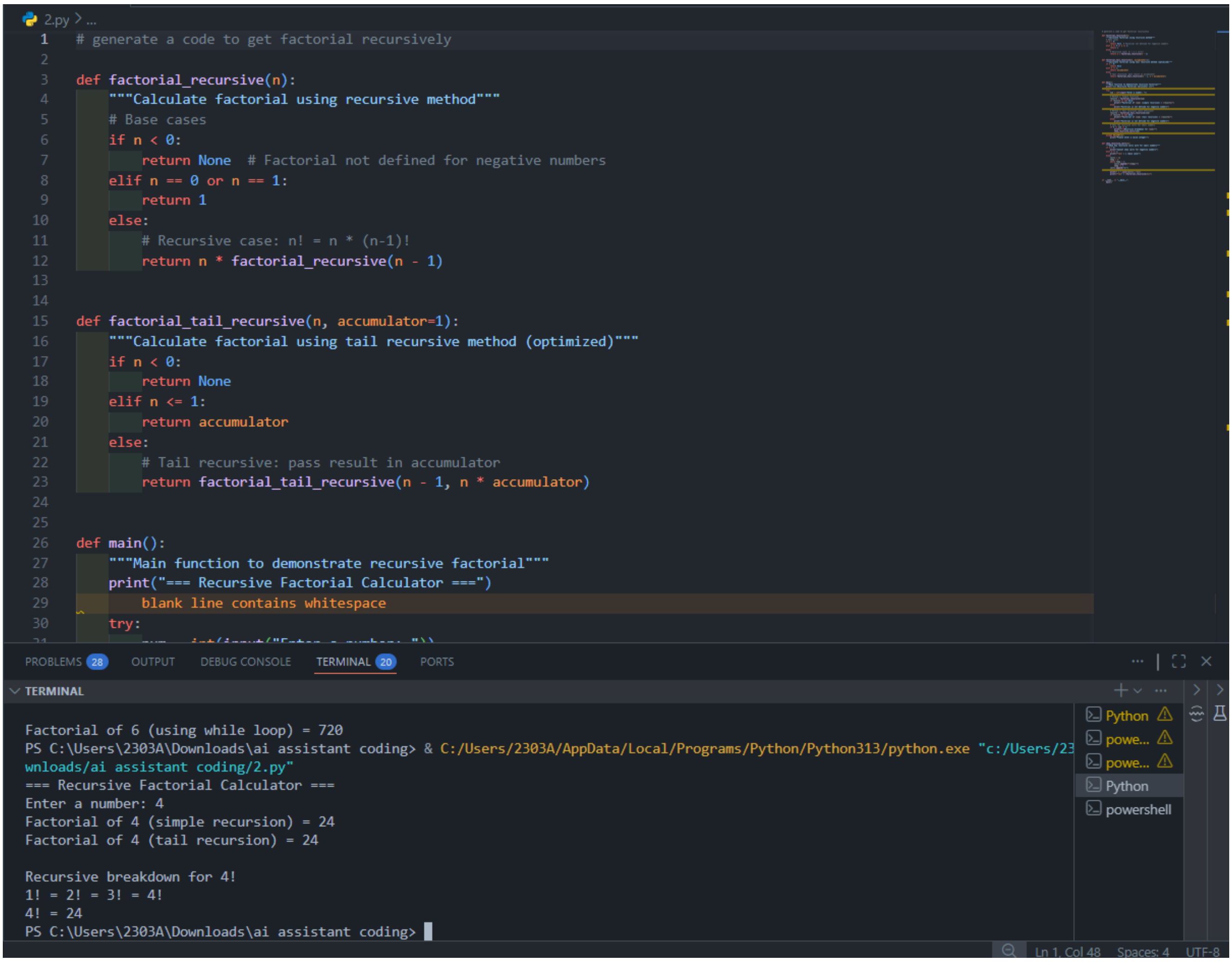
```
PS C:\Users\2303A\Downloads\ai assistant coding> & C:/Users/2303A/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/23
wnloads/ai assistant coding/2.py"
=== Iterative Factorial Calculator ===
Enter a number: 6
Factorial of 6 (using for loop) = 720
Factorial of 6 (using while loop) = 720
PS C:\Users\2303A\Downloads\ai assistant coding>
```

Python
powe...
powe...
Python
powershell

Recursive:

Prompt: # generate a code to get factorial recursively

Code and Output Screenshots:



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.