AI Assistant Coding

NAME: K. Ruchitha HT NO:2303A52069
Batch:32

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

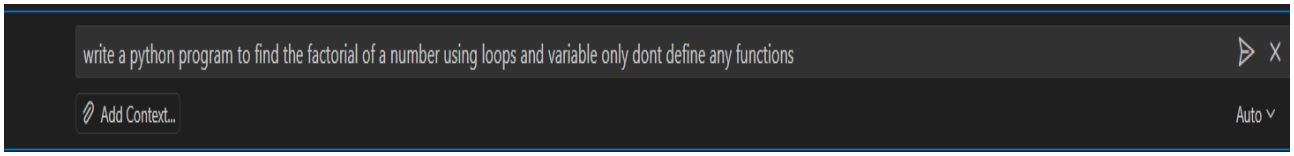
Scenario: You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

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:

1. Do not define any custom function
2. Logic must be implemented using loops and variables only

Prompt:



Code:

```python
# Get the number from user input
n = int(input("Enter a number: "))

# Initialize factorial to 1
factorial = 1

# Use a loop to calculate factorial
for i in range(1, n + 1):
    factorial *= i

# Print the result
print(f"The factorial of {n} is {factorial}")
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\ruchi\OneDrive\Documents\Desktop\course> & C:\Users\ruchi\AppDa
/d.py
enter a number:6
The factorial of 6 is 720
PS C:\Users\ruchi\OneDrive\Documents\Desktop\course>
```

Copilot Suggestions:

```
'''
Docstring for fact
This script calculates the factorial of a given number.
'''
```

Explanation:

The code generated from compile was well structured and for beginner it was very helpful. It reduces coding time and gives code without and logical or syntax error.

Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

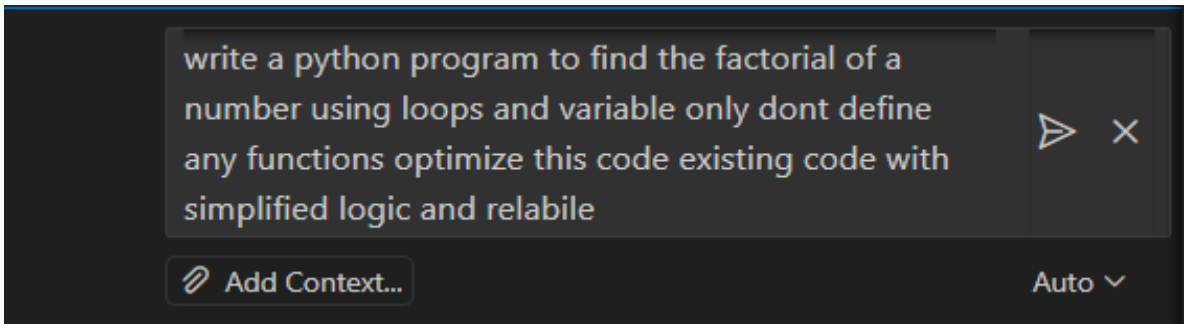
Scenario: Your team lead asks you to review AI-generated code before committing it to a shared repository.

Task Description: Analyse the code generated in Task 1 and use Copilot again to:

1. Reduce unnecessary variables
2. Improve loop clarity
3. Enhance readability and efficiency

Hint: Prompt Copilot with phrases like *“optimize this code”*, *“simplify logic”*, or *“make it more readable”*

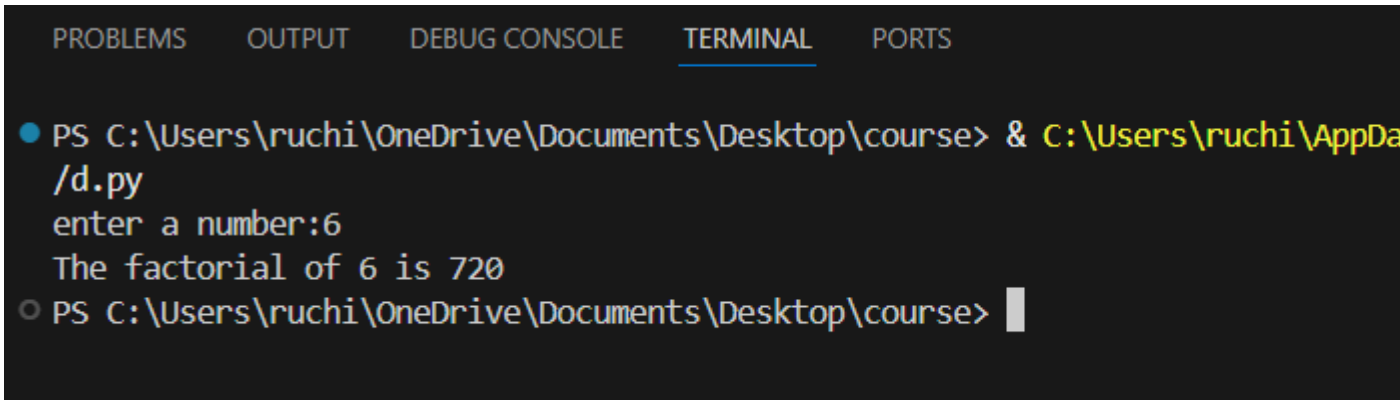
Prompt:



Code:

```python
# Get the number from user input
n = int(input("Enter a number: "))

# Handle edge cases for reliability
if n < 0:
    print("Factorial is not defined for negative numbers.")
elif n == 0:
    print("The factorial of 0 is 1")
else:
    # Initialize factorial to 1
    factorial = 1
    # Use a loop to calculate factorial (simplified and efficient for small n)
    for i in range(1, n + 1):
        factorial *= i
    # Print the result
    print(f"The factorial of {n} is {factorial}")
```

Output:



Explanation:

Compared to previous code this code was most efficient as it handles the edge case for negative and zero input

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:

1.Creating a user-defined function

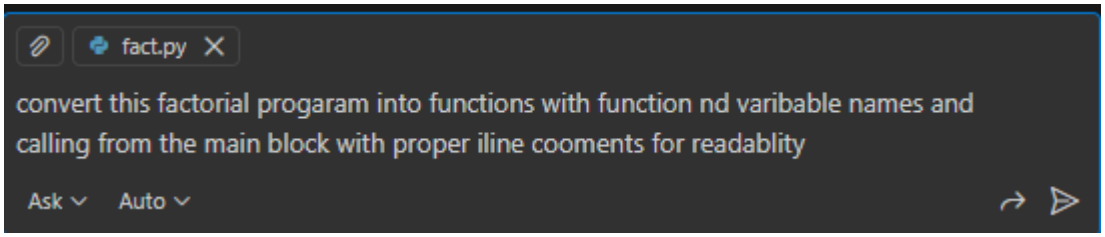
2.Calling the function from the main block

Constraints

1.Use meaningful function and variable names

2.Include inline comments (preferably suggested by Copilot)

Prompt:



Code:

```python
# write code to compute factorial of a number without using recursion and handle invalid input
def compute_factorial(n):
    """
    Docstring for compute_factorial

    :param n: non-negative integer
    :return: factorial of n
    """
    if not isinstance(n, int) or n < 0:
        return "Invalid input. Please enter a non-negative integer."
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial
# Example usage
if __name__ == "__main__":
    user_input = input("Enter a non-negative integer to compute its factorial: ")
    try:
        number = int(user_input)
        result = compute_factorial(number)
        print(f"The factorial of {number} is: {result}")
    except ValueError:
        print("Invalid input. Please enter a valid non-negative integer.")
```

Output:

```
PS C:\Users\ruchi\OneDrive\Documents\Desktop\course> & C:\Users\ruchi\AppData\Local\Programs
/d.py
Enter a non-negative integer to compute its factorial: 7
The factorial of 7 is: 5040
```

Explanation:

The code is converted into different functions with main block calling and improving readability

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:

1.Logic clarity

2.Reusability

3.Debugging ease

4.Suitability for large projects

5.AI dependency risk

	Without Functions	With Functions
Logic Clarity	Logic is written in a single flow,easy to understand	Logic is divided into functions, making the

	but becomes unreadable as lines increases.	code to read & understand easily.
Reusability	Code cannot be reused	Functions can be reused in other programs without writing the logic again.
Debugging Ease	Debugging becomes hard as all the logic at one place	Debugging becomes easy as all the logic written multiple functions
Suitability for Large Projects	Not suitable for large projects	Suitable for large projects due to proper structure
AI Dependency Risk	Higher risk for long procedural code, hard to review or modify.	Lower risk generated in functions, easy

		to review & Modify.

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)

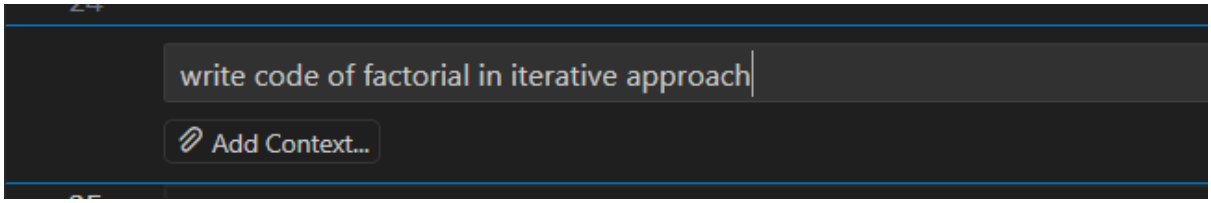
Comparison covering:

- Readability
- Stack usage
- Performance implications

When recursion is *not* recommended

Iterative:

Prompt:



Code:

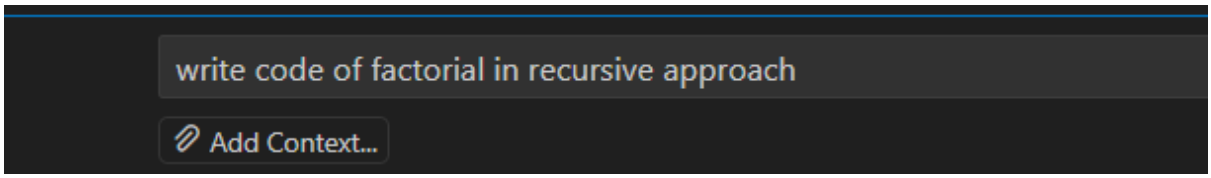
```python
#write a code to compute factorial of a number without using recursion and handling invalid input
def compute_factorial(n):
    """
    Docstring for compute_factorial

    :param n: non-negative integer
    :return: factorial of n
    """
    if not isinstance(n, int) or n < 0:
        return "Invalid input. Please enter a non-negative integer."
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial
# Example usage
if __name__ == "__main__":
    user_input = input("Enter a non-negative integer to compute its factorial: ")
    try:
        number = int(user_input)
        result = compute_factorial(number)
        print(f"The factorial of {number} is: {result}")
    except ValueError:
        print("Invalid input. Please enter a valid non-negative integer.")
```

Output:

```
PS C:\Users\ruchi\OneDrive\Documents\Desktop\course> & C:\Users\ruchi\AppData
sktop/course/aicc/d.py
Enter a non-negative integer to compute its factorial: 6
The factorial of 6 is: 720
```

Recursive:

Prompt:



Code:

```python
# write code to compute factorial of a number  using recursion and handle invalid input
def compute_factorial(n):
    """
    Docstring for compute_factorial

    :param n: non-negative integer
    :return: factorial of n
    """
    if not isinstance(n, int) or n < 0:
        return "Invalid input. Please enter a non-negative integer."
    if n == 0 or n == 1:
        return 1
    return n * compute_factorial(n - 1)
# Example usage
if __name__ == "__main__":
    user_input = input("Enter a non-negative integer to compute its factorial: ")
    try:
        number = int(user_input)
        result = compute_factorial(number)
        print(f"The factorial of {number} is: {result}")
    except ValueError:
        print("Invalid input. Please enter a valid non-negative integer.")
```

Output:

```
PS C:\Users\ruchi\OneDrive\Documents\Desktop\course> & C:\Users\ruchi\AppData
sktop/course/aicc/d.py
Enter a non-negative integer to compute its factorial: 6
The factorial of 6 is: 720
```

	Iterative Approach	Recursive Approach
Readability	Easy to understand for beginners	harder for beginners due to function calls
Stack Usage	Uses constant memory	Uses call stack for each function call
Performance	Faster and more memory-efficient	slower due to function call overhead