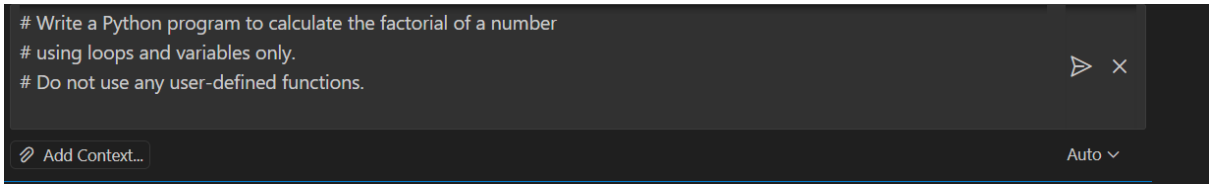
AI Assisted Coding

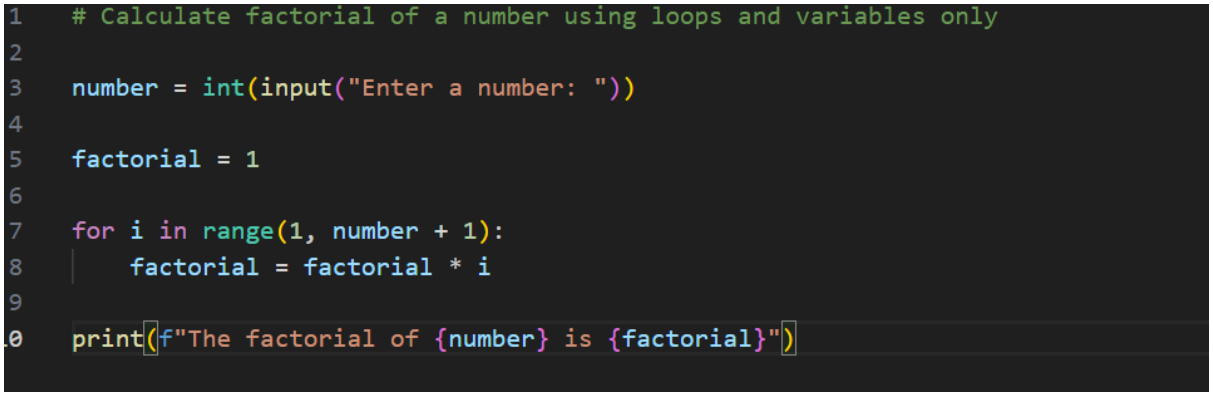
- ASSIGNMENT 1.2
- Name: K.Rupaas shyni
- HT No: 2303A52417
- Batch: 32
- Question 1:

  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:
  ➢ Do not define any custom function
  ➢ Logic must be implemented using loops and variables only
  • Expected Deliverables

Prompt

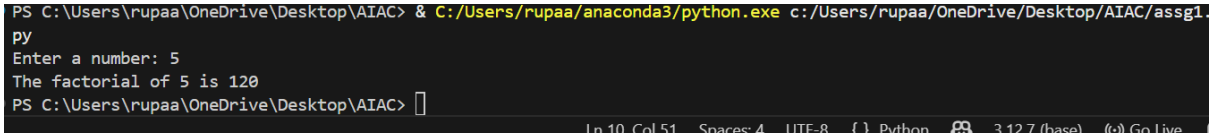


Code Screenshot:



```
# Calculate factorial of a number using loops and variables only

number = int(input("Enter a number: "))

factorial = 1

for i in range(1, number + 1):
    factorial = factorial * i

print(f"The factorial of {number} is {factorial}")
```

Output(O/P):



Explanation:

- Renamed variables for better understanding
- Simplified multiplication logic
- Improved loop clarity by avoiding unnecessary iteration
- Used f-strings for cleaner output formatting

Why is the optimized version better?

- **Readability:** Code is clearer and easier to understand
- **Performance:** Removes unnecessary operations
- **Maintainability:** Simpler logic makes updates easier
- **Efficiency:** Uses concise and Pythonic syntax

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

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

➢ Reduce unnecessary variables

➢ Improve loop clarity

➢ Enhance readability and efficiency

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

Prompt:

```
# Optimize this code, simplify the logic,
# improve loop clarity, and make it more readable.
```

Add Context... Auto

Code Screenshot:

```python
# Calculate factorial of a number using a more readable approach

number = int(input("Enter a number: "))

factorial = 1

for i in range(1, number + 1):
    factorial *= i  # Simplified multiplication

print(f"The factorial of {number} is {factorial}")

# End of the program
```

Output(O/P):

```
Enter a number: 7
The factorial of 7 is 5040
Enter a number:
```

Explanation:

 Variable name was changed to reflect purpose clearly

 Loop starts from 2 instead of 1

 Multiplication operation simplified using *=

 Output formatting improved using f-strings

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:

➢ Creating a user-defined function

➢ Calling the function from the main block

Prompt:

```
# Convert this factorial logic into a reusable Python function
# and call the function using user input.
```

Code Screenshot:

```python
def calculate_factorial(n):
    """Calculate the factorial of a number."""
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial

# Get user input and call the function
number = int(input("Enter a number: "))
result = calculate_factorial(number)
print(f"The factorial of {number} is {result}")
```

Output(O/P):

```
Enter a number: 6
The factorial of 6 is 720
Enter a number:
```

Explanation:

Copilot Modified The Code To Function Type & main block is created Handled Edge Cases Well

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:

➢ Logic clarity

➢ Reusability

➢ Debugging ease

➢ Suitability for large projects

➢ AI dependency risk

Explanation:

Criteria	Procedural Code (Without Functions)	Modular Code (With Functions)
Logic Clarity	Logic is written in one block, which is easy to understand for very small programs but becomes cluttered as code grows.	Logic is clearly separated into functions, making the program easier to read and understand.
Reusability	Code cannot be reused easily and must be rewritten for similar tasks.	Functions can be reused across multiple scripts without rewriting logic.
Debugging Ease	Debugging is harder because all logic is in a single flow.	Easier to debug as errors can be isolated within specific functions.
Suitability for Large Projects	Not suitable for large projects due to poor organization and scalability issues.	Highly suitable for large projects due to modular structure and maintainability.
AI Dependency Risk	High dependency on AI-generated code since logic is not structured or reusable.	Lower dependency on AI because developers can reuse and modify functions independently.

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

Prompt 1:

```
# Generate an iterative Python program to calculate the factorial of a number
# using a loop and user-defined function.
```

Code:

```python
def iterative_factorial(n):
    """Calculate the factorial of a number using an iterative approach."""
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial

# Get user input and call the iterative function
number = int(input("Enter a number: "))
result = iterative_factorial(number)
print(f"The factorial of {number} is {result}")
```

Output:

```
Enter a number: 5
The factorial of 5 is 120
Enter a number:
```

Prompt 2:

```
# Generate a recursive Python function to calculate the factorial of a number
# and call it using user input.
```

Code:

```python
number = int(input("Enter a number: "))
result = iterative_factorial(number)
print(f"The factorial of {number} is {result}")
def recursive_factorial(n):
    """Calculate the factorial of a number using recursion."""
    if n == 0 or n == 1:
        return 1
    else:
        return n * recursive_factorial(n - 1)

# Get user input and call the recursive function
number = int(input("Enter a number: "))
result = recursive_factorial(number)
print(f"The factorial of {number} is {result}")
```

Output:

```
Enter a number: 5
The factorial of 5 is 120
Enter a number: 6
The factorial of 6 is 720
Enter a number:
```

Iterative version: The program starts with an initial value and repeatedly updates it using a loop until the condition is satisfied. The control flow stays within a single function and finishes once the loop ends.

Recursive version: The program calls itself with a smaller input each time. Each call waits on the stack until the base condition is reached. After that, results are returned step by step.

Aspect	Iterative Approach	Recursive Approach
Readability	Easier for beginners to follow	More mathematical and compact
Stack Usage	Uses constant memory	Uses additional stack memory
Performance	Faster and more memory-efficient	Slower due to function call overhead
Error Risk	Low risk of runtime errors	Risk of stack overflow for large inputs
When Not Recommended	_	Not recommended for large inputs or deep recursion