Assignment : 1.2

Name :K.Lokesh

Ht.no :2303A51201

Course Name: Ai Assistant Coding Batch:21

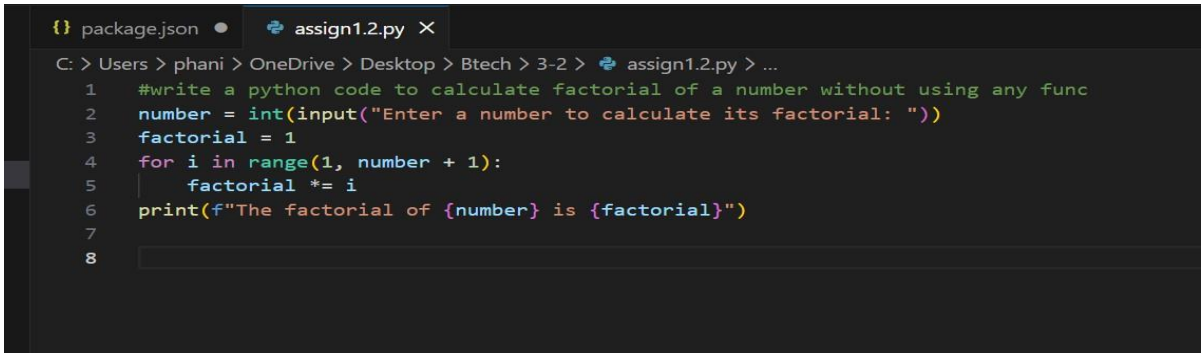
Task 1: AI-Generated Logic Without Modularization (Factorial without Functions)

- Scenario

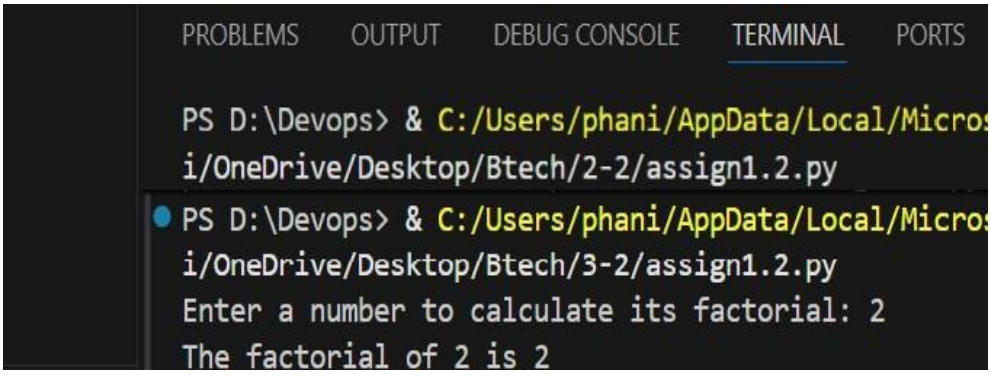
You are building a small command-line utility for a startup intern onboarding task. The program is simple and must be written quickly without modular design.

• 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



Output:



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

```
package.json ●  assign1.2.py ●
C: > Users > phani > OneDrive > Desktop > Btech > 3-2 > assign1.2.py > ...
1   #write a python code to calculate factorial of a number without using any func
2   # optimize the code below for better performance by adding comments
3   # optimize the code below for better performance by adding inline comments where necessary
4
5   num = 5  # Initialize the number for which we want to calculate the factorial
6   factorial = 1  # Initialize factorial variable to store the result
7
8   for i in range(1, num + 1):
9       factorial *= i
10
11  print(f"The factorial of {num} is {factorial}")
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Devops> & C:/Users/phani/AppData/Local/Microsoft/WindowsApps/pyt
Desktop/Btech/3-2/assign1.2.py
The factorial of 5 is 120
PS D:\Devops>
```

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

```python
# give user defined function to calculate factorial of a number
def calculate_factorial(n):
    """Calculate the factorial of a given number n."""
    result = 1  # Initialize result variable to store the factorial value
    for i in range(1, n + 1):  # Loop from 1 to n (inclusive)
        result *= i  # Multiply result by the current number i
    return result  # Return the final factorial value
```

HOW MODULARITY IMPROVES REUSABILITY?

Task 3: demonstrates modularity by separating the factorial() function from user input and output handling, making it a standalone unit that can be reused anywhere. Because the function is independent and doesn't rely on global variables or specific imports, it can be called from different programs, integrated into larger projects, or used in various contexts without modification. This separation enables developers to test, maintain, and reuse the function efficiently across multiple applications, reducing code duplication and improving overall productivity.

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

CRITERIA	PROCEDURAL(Task1)	MODULAR(Task 2/3)
Logic Clarity	Linear flow but mixed with I/O; harder to isolate logic from output statements	Clear separation of logic and I/O; function purpose is explicit and documented with docstrings
Reusability	Limited; code runs at module level once; cannot be called multiple times or imported easily	High; functions can be called repeatedly with different inputs; easily imported into other modules
Debugging Ease	Difficult; global state makes it hard to track variable changes; print statements clutter output	Easy; input/output separation allows isolated testing; return values simplify tracing and verification
Suitability for Large Projects	Poor; doesn't scale; mixing procedural code creates maintenance nightmares; hard to organize multiple operations	Excellent; modular structure supports larger codebases; functions can be organized into modules and package
AI Dependency Risk	High; AI must regenerate entire logic if context changes; procedural code is context-dependent	Lower; function abstraction reduces AI regeneration needs; stable interfaces minimize prompt changes

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:

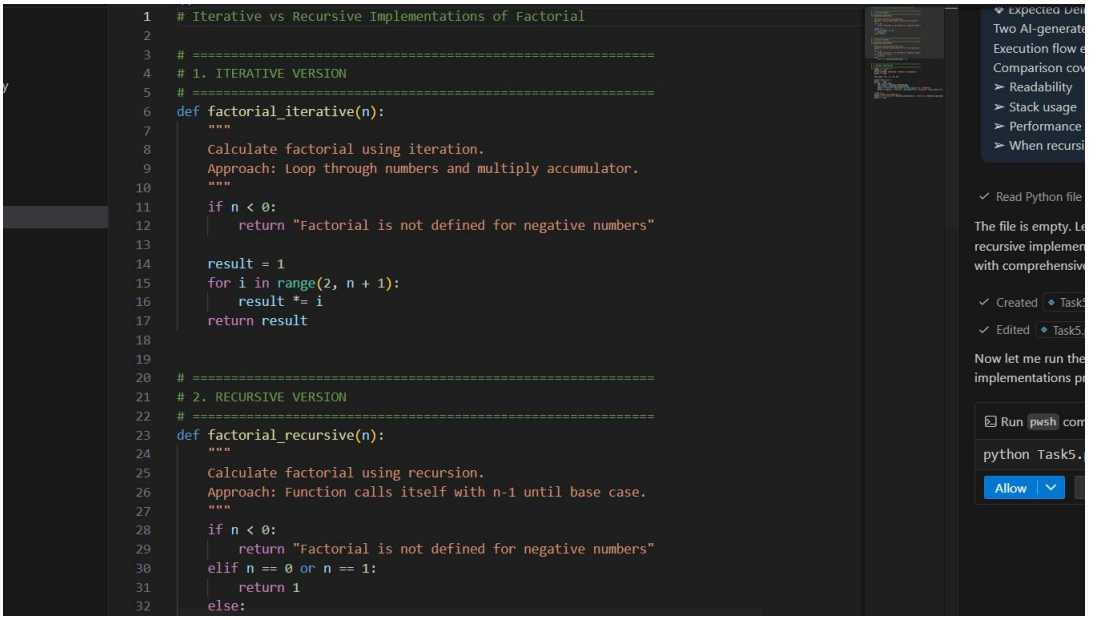
➢ **Readability**

➢ **Stack usage**

➢ **Performance implications**

➢ **When recursion is not recommended.**

OUTPUTS :




```python
# Iterative vs Recursive Implementations of Factorial

# ============================================================
# 1. ITERATIVE VERSION
# ============================================================
def factorial_iterative(n):
    """
    Calculate factorial using iteration.
    Approach: Loop through numbers and multiply accumulator.
    """
    if n < 0:
        return "Factorial is not defined for negative numbers"

    result = 1
    for i in range(2, n + 1):
        result *= i
    return result


# ============================================================
# 2. RECURSIVE VERSION
# ============================================================
def factorial_recursive(n):
    """
    Calculate factorial using recursion.
    Approach: Function calls itself with n-1 until base case.
    """
    if n < 0:
        return "Factorial is not defined for negative numbers"
    elif n == 0 or n == 1:
        return 1
    else:
```

```python
def factorial_recursive(n):
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)


# ============================================================
# 3. TESTING & VERIFICATION
# ============================================================
if __name__ == "__main__":
    print("=" * 60)
    print("FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE")
    print("=" * 60)

    test_cases = [0, 1, 5, 10, 15]

    print("\nTest Cases:")
    print("-" * 60)
    for num in test_cases:
        iter_result = factorial_iterative(num)
        recur_result = factorial_recursive(num)
        match = "✓ MATCH" if iter_result == recur_result else "✗ MISMATCH"
        print(f"n={num:2d} | Iterative: {iter_result:>18} | Recursive: {recur_result:>18}

    # Edge case
    print(f"\nEdge Case (negative):")
    print(f"n=-5 | Iterative: {factorial_iterative(-5)} | Recursive: {factorial_recursive
    print("=" * 60)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(.venv) PS D:\AIASSCoding> python Task5.py
============================================================
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
============================================================
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
FACTORIAL COMPARISON: ITERATIVE vs RECURSIVE
============================================================

Test Cases:
------------------------------------------------------------
n= 0 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
n= 1 | Iterative:                  1 | Recursive:                  1 | ✓ MATCH
n= 5 | Iterative:                120 | Recursive:                120 | ✓ MATCH
```

Execution Flow Explanation:

Comparison

Readability :

Iterative: Crystal clear for most developers. A simple loop that anyone can understand instantly. Better for learning loops.

Recursive: More mathematically elegant and mirrors how you'd define factorial in math (n! = n × (n-1)!), but requires understanding function call stacks. Harder for beginners.

Stack Usage :

Iterative: Uses constant memory O(1). Only stores one variable (result). No function call overhead.

Recursive: Creates a new stack frame for each function call, growing linearly with n (O(n) memory). For n=1000, it needs 1000 stack frames—risky and wasteful.

Performance Implications :

Iterative: Fast. No function call overhead. Runs in microseconds even for large n.

Recursive: Slow. Each function call has overhead (10-20x slower per call). For n=1000, the iterative version is orders of magnitude faster.

When Recursion Is NOT Recommended :

1. **Large n values** – Stack overflow risk; Python's limit is ~1000 calls
2. **Performance-critical code** – Function call overhead is expensive
3. **Simple problems with loops** – Unnecessary complexity and slowdown
4. **Factorial specifically** – Iterative is always better; no benefit from recursion
5. **Embedded/resource-limited system**s – Limited stack memory
6. **When clarity matters** – Loops are more intuitive for most people