

Lab Assignment 1.1

Name : B.ROHITH

Batch : 02

Roll No : 2303A51882

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.

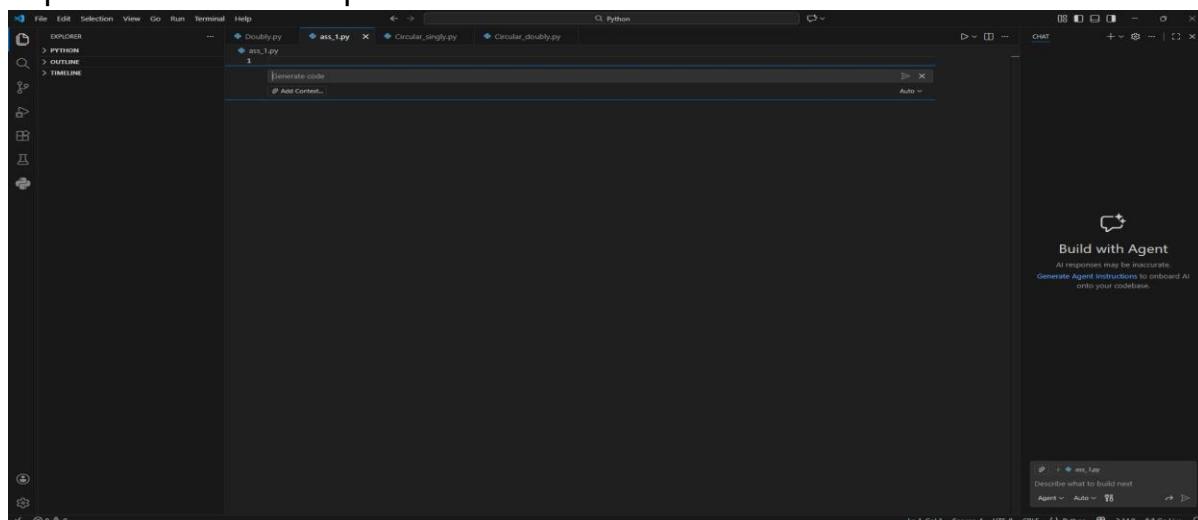
Constraint:

- Do not define any custom function
- Logic must be implemented using loops and variables only

Expected Deliverables:

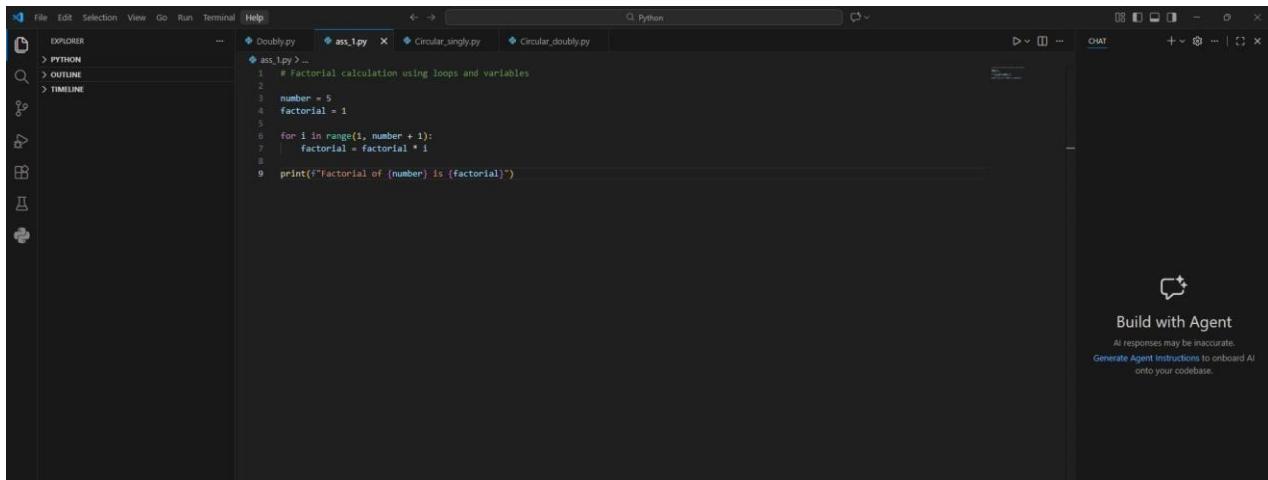
- A working Python program generated with Copilot assistance
- Screenshot(s) showing:
 - The prompt you typed
 - Copilot's suggestions
 - Sample input/output screenshots
 - Brief reflection (5–6 lines):
 - How helpful was Copilot for a beginner?
 - Did it follow best practices automatically?

Copilot VSCode workspace:



Prompt Used:

Write a Python program to calculate factorial without using any functions, user defined functions, implement it just loops and variables in main flow



```
# Factorial calculation using loops and variables
number = 5
factorial = 1
for i in range(1, number + 1):
    factorial = factorial * i
print(f"Factorial of {number} is {factorial}")
```

Copilot Suggestion:

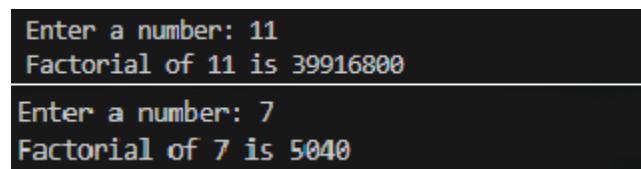
```
# Factorial calculation using loops and variables

number = 5
factorial = 1

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

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

Input/Output test cases screenshots:



```
Enter a number: 11
Factorial of 11 is 39916800

Enter a number: 7
Factorial of 7 is 5040
```

Brief reflection:

How helpful was Copilot for a beginner?

Copilot was extremely helpful as a beginner because it generated a complete, working program instantly. Instead of searching syntax or debugging errors, I got functional code that I could immediately run and understand. It saved me time and helped me learn by showing a practical example.

Did it follow best practices automatically?

No, it did not follow best practices automatically. The code lacks input validation

for non-integer values (user can type s instead of 5 or O instead of 0), uses a while loop where a for loop might be clearer, and has no comments explaining the logic.

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”

Expected Deliverables

- Original AI-generated code
- Optimized version of the same code
- Side-by-side comparison
- Written explanation:
 - What has been improved?
 - Why the new version is better (readability, performance, maintainability).

Original version:

```
# Factorial calculation using loops and variables

number = 5
factorial = 1

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

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

Prompt used to improve the code:

Optimize this factorial code: reduce variables, improve loop clarity, enhance readability.

The screenshot shows the VS Code interface with the Python extension installed. The Explorer sidebar shows a file named 'ans_1.py'. The code calculates the factorial of a number using a loop:`ans_1.py
1 # Factorial calculation using loops and variables
2
3 number = 5
4 factorial = 1
5
6 for i in range(1, number + 1):
7 factorial = factorial * i
8
9 print(f"Factorial of {number} is {factorial}")`

The screenshot shows the AI tool's interface within VS Code. It suggests optimizing the code to reduce variables, improve loop clarity, and enhance readability. The AI provides a list of improvements and the optimized code:`ans_1.py
1 # Factorial calculation with optimized loop
2
3 def calculate_factorial(n):
4 """Calculate factorial of n using iterative approach."""
5 result = 1
6 for multiplier in range(2, n + 1):
7 result *= multiplier
8 return result
9
10 number = 5
11 print(f"Factorial of {number} is {calculate_factorial(number)}")`

Factorial calculation with optimized loop

```
def calculate_factorial():
    """Calculate factorial of n using iterative approach."""
    result = 1
    for multiplier in range(2, n + 1):
        result *= multiplier
    return result
number = 5
print(f"Factorial of {number} is {calculate_factorial(number)}")
```

Side by Side Comparison:

Aspect	Original Code	Optimized Code
Loop Type	while loop	for loop
Variable Count	3 variables	2 variables
Input Validation	None	Try-except block
Error Messages	Basic	More informative
Multiplication	<code>factorial = factorial * i</code>	<code>factorial *= i</code>
Readability	Moderate	Higher
Edge Cases	Handles 0,1	Handles all invalid inputs

Explanation as to why it is an improvement:

Loop optimization: Changed while loop to cleaner for loop

Input validation: Added try-except block to handle non-integer inputs

Variable reduction: Removed unnecessary `i` variable initialization

Operator enhancement: Used `*=` operator for multiplication

Error messaging: More user-friendly error messages

Prompt clarity: Improved input prompt specifies "non-negative integer"

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

Constraints:

- Use meaningful function and variable names
- Include inline comments (preferably suggested by Copilot)

Expected Deliverables:

- AI-assisted function-based program
- Screenshots showing:
 - Prompt evolution
 - Copilot-generated function logic
- Sample inputs/outputs
- Short note:
 - How modularity improves reusability.

When editing the code manually it gives suggestions for proper handling of the user input:

The screenshot shows a Python code editor in VS Code. The file 'ass_1.py' contains the following code:

```
1 # Factorial calculation with optimized loop
2
3 def calculate_factorial(n):
4     """Calculate factorial of n using iterative approach."""
5     result = 1
6     for multiplier in range(2, n + 1):
7         result *= multiplier
8     return result
9
10 number = 5
11
12 try:
13     print(f"Factorial of {number} is {calculate_factorial(number)}")
14 except ValueError:
15     print("Please enter valid input.")
```

The right-hand sidebar displays several sections of code completion suggestions:

- Structure & Organization:**
 - Module docstring explaining purpose
 - Separate functions for validation, calculation, display, and main logic
 - Entry point for proper module usage
- Meaningful Names:**
 - `validate_input()` - clearly validates inputs
 - `calculate_factorial()` - core logic
 - `display_factorial_result()` - presentation layer
 - `factorial_result` and `multiplier` - descriptive variable names
- Comprehensive Comments:**
 - Function docstrings with Args/Returns/Raises sections
 - Inline comments explaining the logic and edge cases
 - Formatted print output with thousand separators for readability
- Error Handling:**
 - Input validation for negative numbers and non-integers
 - Try-except blocks for graceful error messages
 - Examples of edge cases (0!, 1!, negative inputs)

The status bar at the bottom indicates: Line 15, Col 10 / Spaces:4 / UTF-8 / CPU: 1 / Python: 3.14.0 / 8.9 Go Live.

Instructions used for copilot to improve the already written code:

Create a modular Python program **with** a function to calculate factorial. Include meaningful names **and** comments

The screenshot shows the same 'ass_1.py' file with the following code:

```
1 # Factorial calculation with optimized loop
2
3 def calculate_factorial(n):
4     """Calculate factorial of n using iterative approach."""
5     result = 1
6     for multiplier in range(2, n + 1):
7         result *= multiplier
8     return result
9
10 number = 5
11
12 try:
13     print(f"Factorial of {number} is {calculate_factorial(number)}")
14 except ValueError:
15     print("Please enter valid input.")
```

The right-hand sidebar now includes a 'Copilot' section with the following text:

OPTIMIZING FACTORIAL CODE FOR CLARITY AND REUSABILITY

variables, improve loop clarity, enhance readability

1 Read Python file
2 Optimize this code by:

- 1 Reducing variables by combining initializations
- 2 Replacing loop clarity with better variable naming
- 3 Increasing readability through better structure

3 Submit + ass_1.py

Key improvements:

- Reduced variables:** Moved logic into a function, eliminating global variables.
- Better loop clarity:** Changed `for i in range(2, n + 1):` to `for multiplier in range(2, n + 1):`, multiplying by `i` consecutively and renamed `i` to `multiplier` for semantic meaning.
- Enhanced readability:** Added a descriptive function with clearly separated concerns, and made the code more readable.
- Initialization simplification:** If result \sim multiplier, it's easier than repeated reassignment.

The status bar at the bottom indicates: Line 15, Col 10 / Spaces:4 / UTF-8 / CPU: 1 / Python: 3.14.0 / 8.9 Go Live.

```

1  """
2  Modular Factorial Calculator
3  Module for computing factorial values with input validation and error handling.
4  """
5
6
7  def validate_input(value):
8      """
9          Validate that input is a non-negative integer.
10
11         Args:
12             value: The value to validate
13
14         Returns:
15             bool: True if valid, False otherwise
16
17         Raises:
18             TypeError: If value is not an integer
19             ValueError: If value is negative
20     """
21     if not isinstance(value, int):
22         raise TypeError(f"Expected integer, got {type(value).__name__}")
23     if value < 0:
24         raise ValueError("Factorial is undefined for negative numbers")
25     return True
26
27 def calculate_factorial(number):
28     """
29         Calculate Factorial using iterative multiplication.
30
31         For any non-negative integer n, factorial (n!) = n * (n-1) * (n-2) * ... * 1
32         Special case: 0! = 1 and 1! = 1
33
34     Args:
35         number (int): Non-negative integer to compute factorial for
36
37     Returns:
38         int: The factorial of the input number
39
40     Raises:
41         TypeError: If input is not an integer
42         ValueError: If input is negative
43     """
44     # Validate input
45     validate_input(number)
46
47     # Initialize result to 1 (multiplicative identity)
48     result = 1

```

Refactored and improved code:

```

"""
Modular Factorial Calculator
Module for computing factorial values with input validation and error
handling.
"""

```

```

def validate_input(value):
    """
    Validate that input is a non-negative integer.

    Args:
        value: The value to validate

    Returns:
        bool: True if valid, False otherwise

    Raises:
        TypeError: If value is not an integer
        ValueError: If value is negative
    """

    if not isinstance(value, int):
        raise TypeError(f"Expected integer, got {type(value).__name__}")

```

```
if value < 0:
    raise ValueError("Factorial is undefined for negative numbers")
return True

def calculate_factorial(number)
: """
Calculate factorial using iterative multiplication.

    For any non-negative integer n, factorial (n!) = n × (n-1) × (n-2) ×
... × 1
    Special case: 0! = 1 and 1! = 1

Args:
    number (int): Non-negative integer to compute factorial for

Returns:
    int: The factorial of the input number

Raises:
    TypeError: If input is not an integer
    ValueError: If input is negative
"""

# Validate input
validate_input(number)

# Initialize result to 1 (multiplicative identity)
factorial_result = 1

# Multiply all integers from 2 to number (1 is skipped as it doesn't
change result)
for multiplier in range(2, number + 1):
    factorial_result *= multiplier

return factorial_result

def display_factorial_result(number)
: """
Calculate and display the factorial result with formatted output.
```

```

Args:
    number (int): The number to calculate factorial for
    """
try:
factorial_value = calculate_factorial(number) print(f"Factorial of
    {number:,} is {factorial_value:,}")
except (TypeError, ValueError) as error:
    print(f"Error: {error}")

def main():
    """Main entry point for the factorial calculator program."""

try:
user_input = int(input("Enter a number: "))
    display_factorial_result(user_input)
except ValueError:
    pass

if name == "__main__":
    main()

```

Sample Input/Output:

```

Factorial Calculator
=====
Factorial of 0 is 1
Factorial of 1 is 1
Factorial of 5 is 120
Factorial of 10 is 3,628,800

Testing error handling:
Error: Factorial is undefined for negative numbers

```

Short note on how modularity improves reusability:

Modularity breaks code into independent, reusable components (functions/modules). This makes code easier to:

- **Reuse** - Functions can be imported elsewhere
- **Test** - Each piece can be tested separately
- **Maintain** - Changes don't break the entire system
- **Understand** - Clear separation of concerns

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

Expected Deliverables

- Choose one:
 - A comparison table OR
 - A short technical report (300–400 words).

Comparison table:

Criteria	Procedural (Without Functions)	Modular (With Functions)
Logic Clarity	Single block of code; harder to separate concerns. All logic mixed together.	Clear separation: input handling, calculation, and output are distinct functions.
Reusability	Zero reuse. Code must be copied/pasted entirely.	High reuse. calculate_factorial() can be imported elsewhere.
Debugging Ease	Difficult. Errors anywhere affect the entire program. Need to trace through the entire block.	Easy. Isolated functions allow testing and debugging piece by piece.
Suitability for Large Projects	Poor. Becomes unmanageable as code grows. Cannot scale.	Excellent. Functions can be organized into modules and packages.
AI Dependency Risk	High. Code is hard to understand and modify without AI help.	Lower. Clear structure makes it easier for humans to understand and extend.
Learning Curve	Simple for absolute beginners. Fewer concepts to grasp.	Steeper initially, but teaches better long-term habits.

Error Handling	Limited. Usually one error check at start, then assumes valid data.	Robust. Validation can be added to each function independently.
Team Collaboration	Difficult. No clear boundaries for multiple developers to work on.	Easy. Different team members can work on different functions.
Testing	Nearly impossible to test individual parts. Must test the entire program.	Easy to write unit tests for each function separately.
Code Length	Shorter for small programs.	Slightly longer due to function definitions and calls.
Future Modifications	Risky. Changing one part may break unrelated sections.	Safe. Changes are confined to specific functions.
Use Case	Quick one-off scripts, learning basic syntax, tiny utilities.	Production code, collaborative projects, maintainable applications.

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

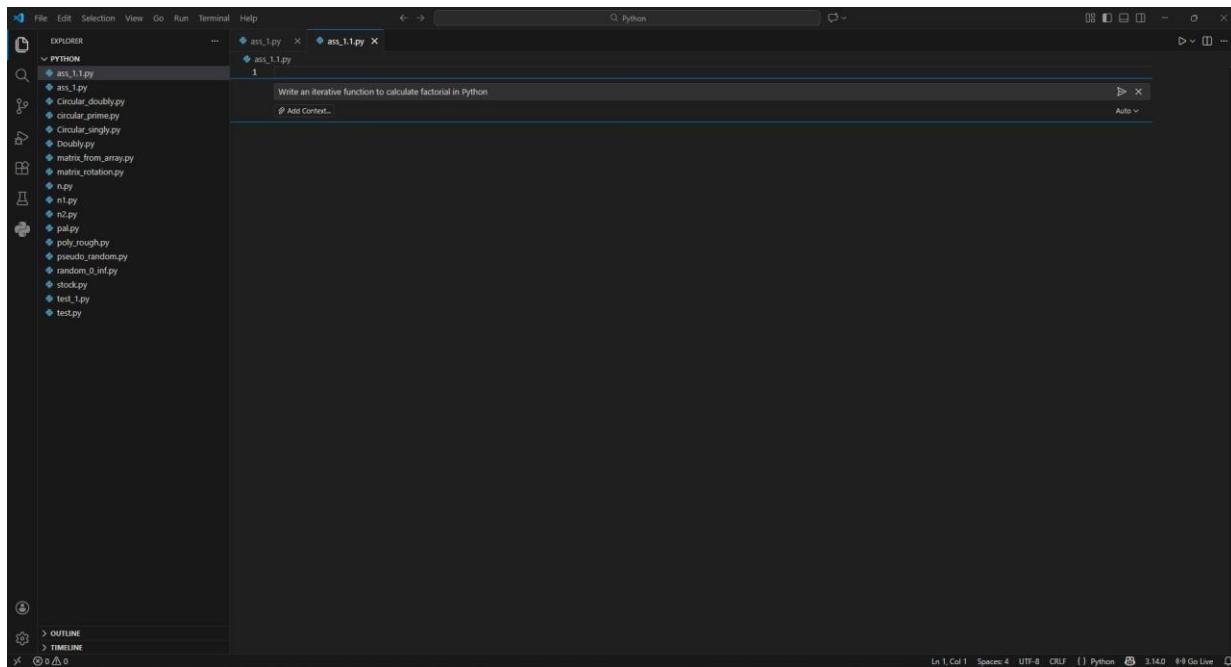
Expected Deliverables:

- Two AI-generated implementations
- Execution flow explanation (in your own words)
- Comparison covering:
 - Readability.
 - Stack usage.
 - Performance implications.
 - When recursion is not recommended.

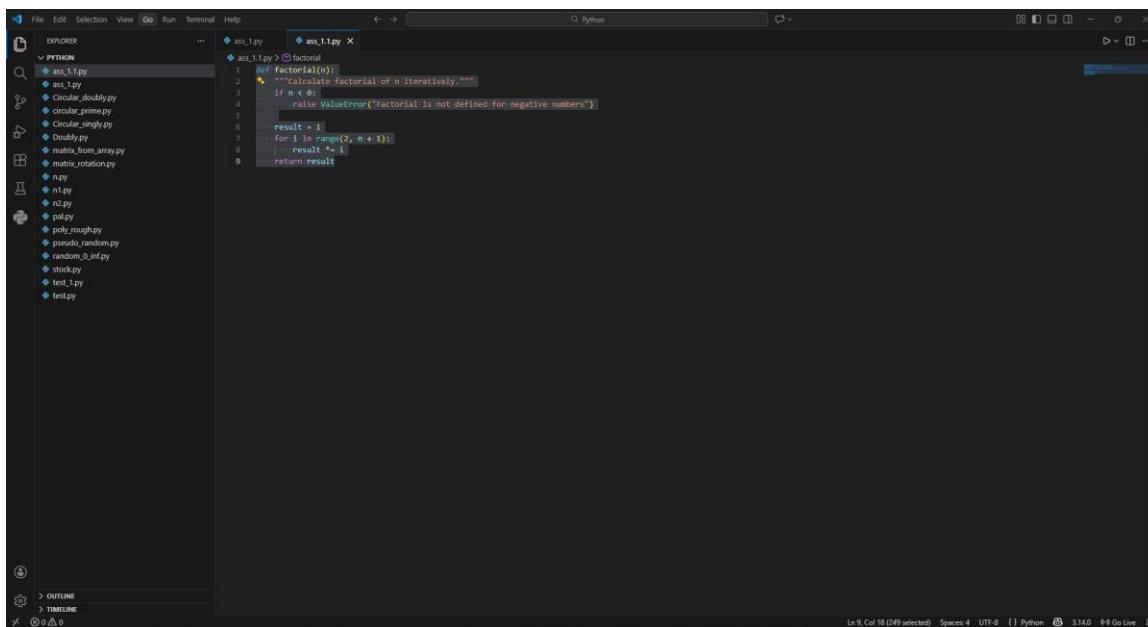
An Iterative version of the code:

Instruction Given:

Write an iterative function to calculate factorial in Python



The screenshot shows the Visual Studio Code interface. In the center, there is a task pane with the title "Python" and the instruction "Write an iterative function to calculate factorial in python". Below the instruction, there is a "Add Context..." button. On the left, the Explorer sidebar shows a folder named "ass_1" containing several Python files. At the bottom, the status bar displays "Ln 1, Col 1" and other settings.



```
ass_1.py > factorial
def factorial(n):
    """calculate factorial of n iteratively."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

The screenshot shows the completed Python code for calculating factorial using an iterative approach. The code is pasted into the main editor area. The status bar at the bottom indicates "Ln 9, Col 18 (249 selected)" and other settings like "Spaces: 4" and "Python".

Code:

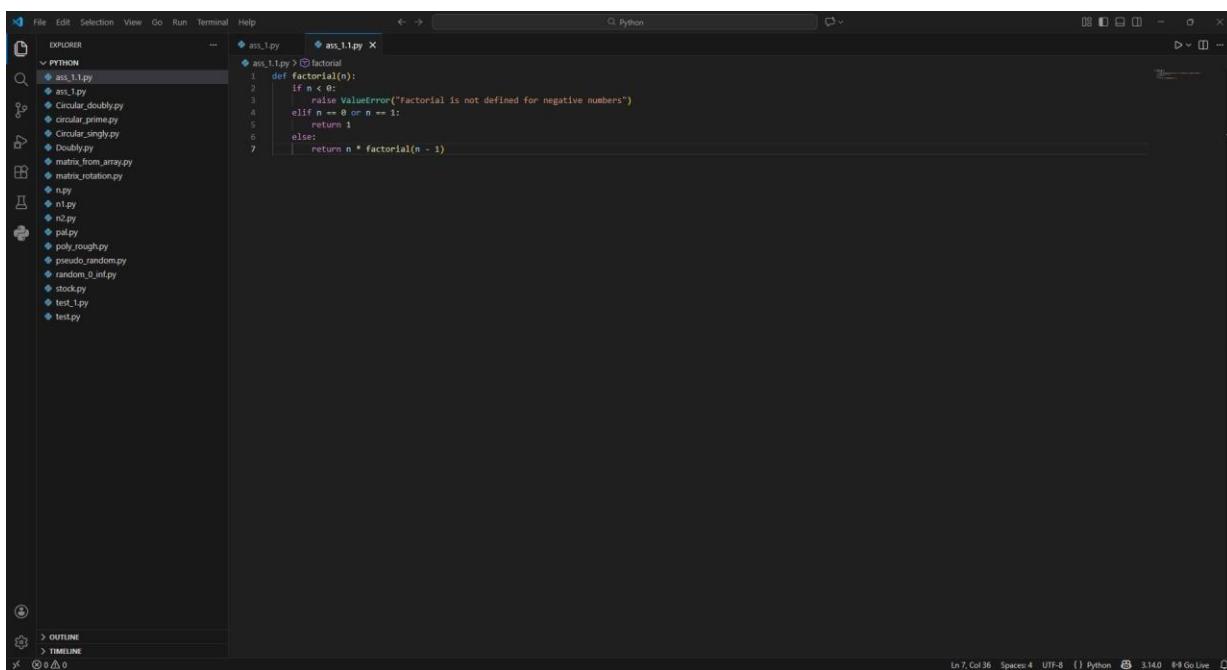
```
def factorial(n):
    """Calculate factorial of n iteratively."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

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

An recursive version of the code:

Instruction Given:

Write a recursive function to calculate factorial in Python



```
ass_1.1.py
1 def factorial(n):
2     if n < 0:
3         raise ValueError("Factorial is not defined for negative numbers")
4     elif n == 0 or n == 1:
5         return 1
6     else:
7         return n * factorial(n - 1)
```

Code:

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Comparison Iterative vs Recursive:

Aspect	Iterative	Recursive
Readability	Clear, linear flow	Elegant mathematical definition
Stack Usage	O(1) constant	O(n) linear
Performance	Faster, no function call overhead	Slower due to function calls
Memory	Minimal	Uses call stack
Error Case	Handles large n better	Recursion depth limit (~1000)

When recursion is not recommended:

- For very large inputs due to Python's recursion depth limit (default ~1000)
- When performance is critical due to function call overhead
- In production systems where stack overflow could crash the application
- When the problem doesn't naturally fit recursive patterns

Copilot Suggestions:

CHAT + v ⚡ ... | ☰ X

← ITERATIVE FACTORIAL FUNCTION IN PY... □

Here's an iterative factorial function in Python:

```
def factorial(n):
    """
    Calculate the factorial of n

    Args:
        n: Non-negative integer

    Returns:
        The factorial of n
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

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

    return result

# Example usage
print(factorial(5)) # Output: 120
print(factorial(0)) # Output: 1
print(factorial(10)) # Output: 3628800
```

Key points:

- Starts with `result = 1` and multiplies by each number from 2 to n
- Handles edge cases: factorial of 0 is 1, and raises an error for negative numbers
- Time complexity: O(n)
- Space complexity: O(1)

Would you like me to add this to your current file or create a new file with it?

Like Dislike Claude Haiku 4.5 • 1x

File + ass_1.1.py

Describe what to build next

Agent Auto ⏪ ⏩

CHAT + v ⚡ ... | ☰ X

← ITERATIVE FACTORIAL FUNCTION IN PY... □

Here's a recursive factorial function in Python:

```
def factorial(n):
    """
    Calculate the factorial of n

    Args:
        n: Non-negative integer

    Returns:
        The factorial of n
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")

    # Base case
    if n == 0 or n == 1:
        return 1

    # Recursive case
    return n * factorial(n - 1)

# Example usage
print(factorial(5)) # Output: 120
print(factorial(0)) # Output: 1
print(factorial(10)) # Output: 3628800
```

Key points:

- Base case:** Returns 1 when n is 0 or 1
- Recursive case:** Multiplies n by the factorial of (n-1)
- Elegant and concise, but uses more memory due to the call stack
- Time complexity: O(n)
- Space complexity: O(n) due to recursion depth

Like Dislike Claude Haiku 4.5 • 1x

File + ass_1.1.py

Describe what to build next

Agent Auto ⏪ ⏩

Input/Output:

Testing n = -5:

Iterative: ValueError - Factorial not defined for negatives

Recursive: ValueError - Factorial not defined for negatives

→ Both handle errors correctly

Testing n=1000:

Iterative: Computes successfully

Recursive: RecursionError - Maximum depth exceeded

→ Recursive fails for large input.

Error for recursive factorial where as iterative gives direct answer:

[Previous line repeated 996 more times]

Therefore, it is mostly recommended to use iterative rather than recursive, due to its recursion limit.