Lab Assignment 1.1

Name : Shivanand Rama
Batch : 01
Roll No : 2303A5103

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.

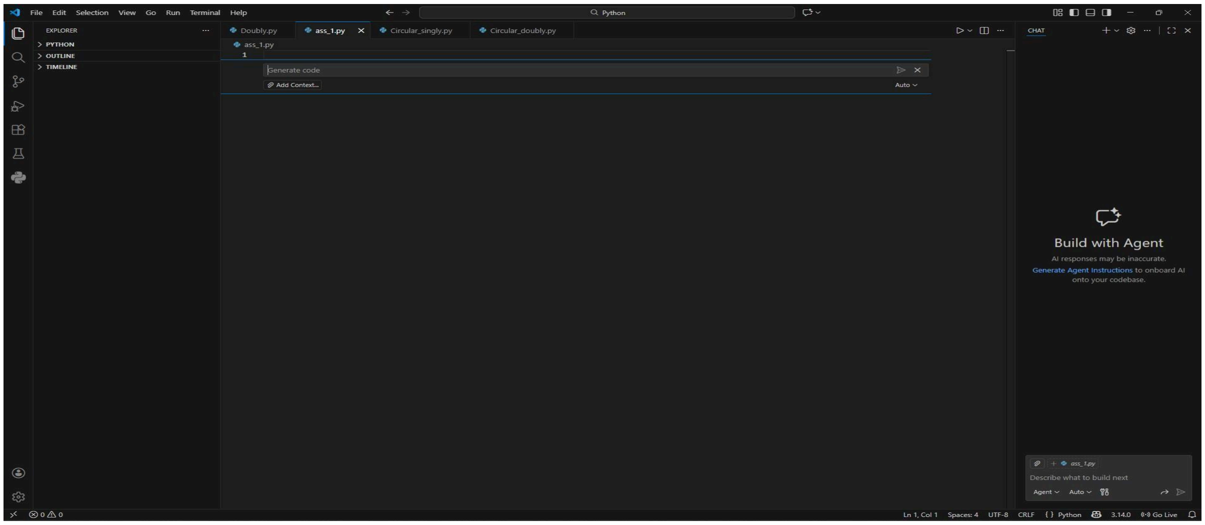
Constraint:

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

Expected Deliverables:

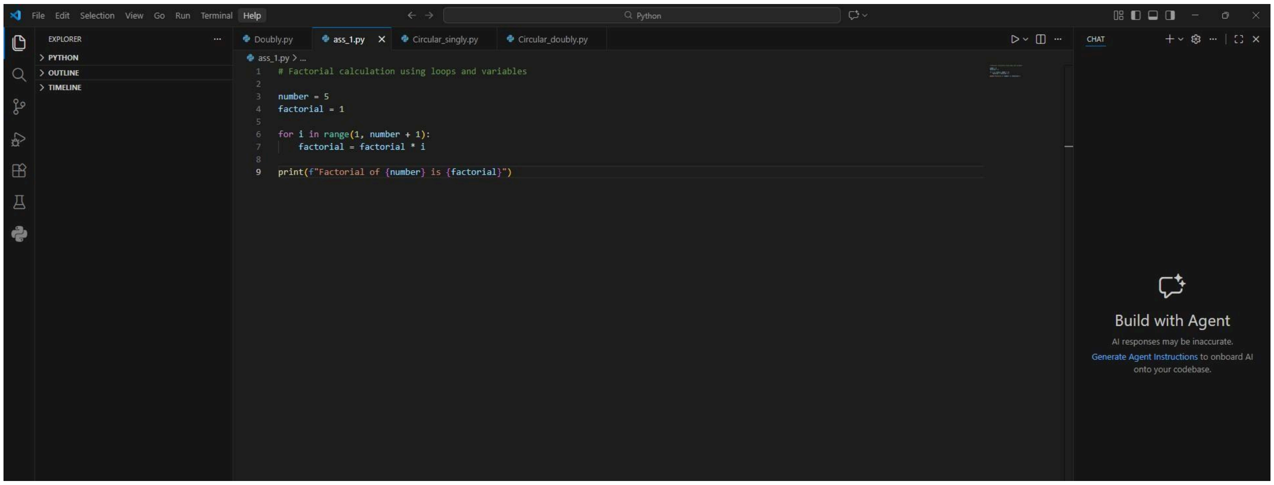
- A working Python program generated with Copilot assistance
- Screenshot(s) showing:
- The prompt you typed
- Copilot's suggestions
- Sample input/output screenshots
- Brief reflection (5–6 lines):
- How helpful was Copilot for a beginner?
- Did it follow best practices automatically?

Copilot VSCode workspace:


File Edit Selection View Go Run Terminal Help
Python
EXPLORER
PYTHON
OUTLINE
TIMELINE
Doubly.py
ass_1.py
Circular_singly.py
Circular_doubly.py
Generate code
Add Context...
Auto
CHAT
Build with Agent
AI responses may be inaccurate.
Generate Agent Instructions to onboard AI onto your codebase.
Describe what to build next
Agent
Auto
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.14.0 Go Live

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



```python
# 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}")
```

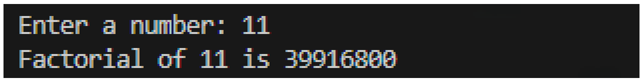
AI responses may be inaccurate.

Generate Agent Instructions to onboard AI onto your codebase.

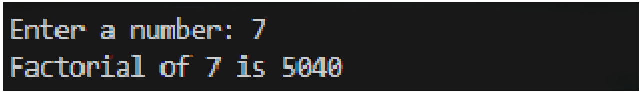
Copilot Suggestion:

```python
# 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

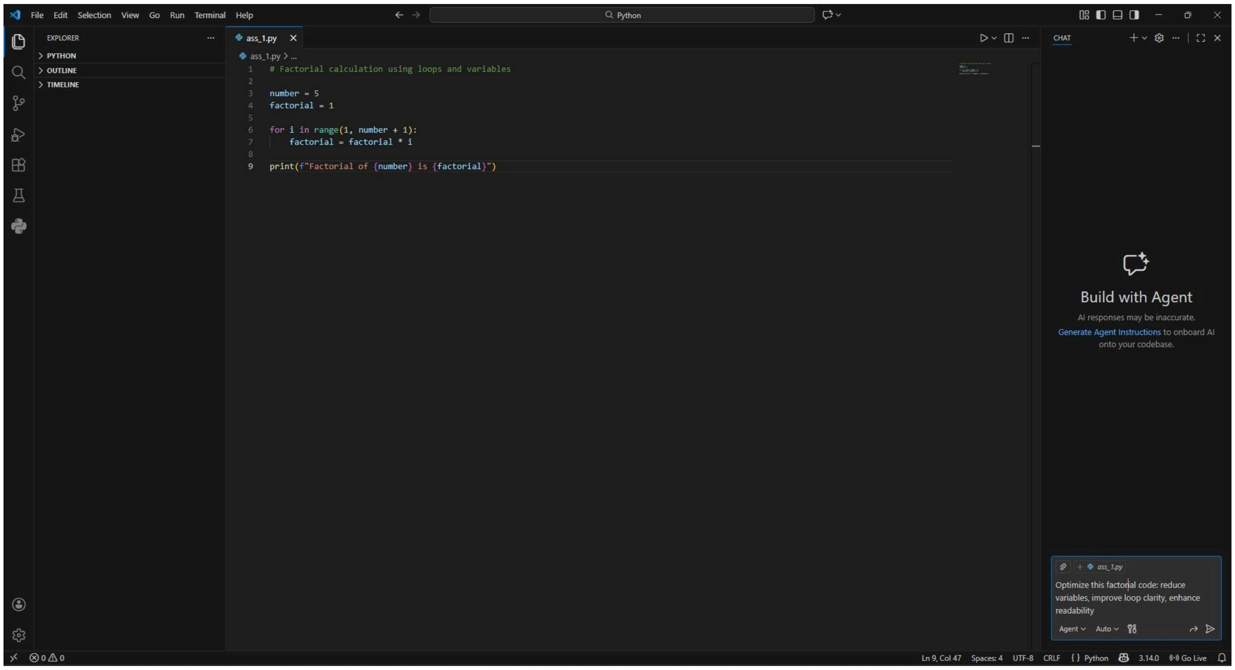
- Original AI-generated code
- Optimized version of the same code
- Side-by-side comparison
- Written explanation:
  - What has been improved?
  - Why the new version is better (readability, performance, maintainability.

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.



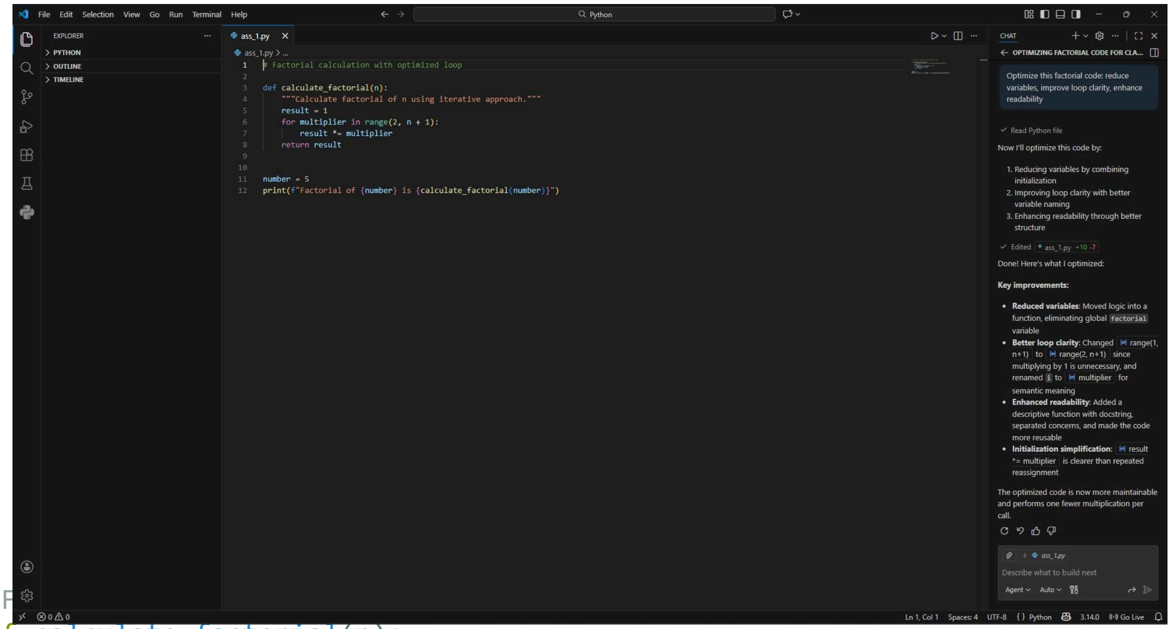
```python
# Factorial calculation using loops and variables

number = 5
factorial = 1

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

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

Build with Agent

AI responses may be inaccurate.

Generate Agent Instructions to onboard AI onto your codebase.

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



```python
# Factorial calculation with optimized loop

def calculate_factorial(n):
    """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)}")
```

OPTIMIZING FACTORIAL CODE FOR CLA...

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

✓ Read Python file

Now I'll optimize this code by:

1. Reducing variables by combining initialization
2. Improving loop clarity with better variable naming
3. Enhancing readability through better structure

✓ Edited ass_1.py +10 -7

Done! Here's what I optimized:

Key improvements:

- **Reduced variables**: Moved logic into a function, eliminating global `factorial` variable
- **Better loop clarity**: Changed `range(1, n+1)` to `range(2, n+1)` since multiplying by 1 is unnecessary, and renamed `i` to `multiplier` for semantic meaning
- **Enhanced readability**: Added a descriptive function with docstring, separated concerns, and made the code more reusable
- **Initialization simplification**: `result *= multiplier` is clearer than repeated reassignment

The optimized code is now more maintainable and performs one fewer multiplication per call.

```python
# F
def calculate_factorial(n):
    """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	factorial = factorial * i	factorial *= i
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

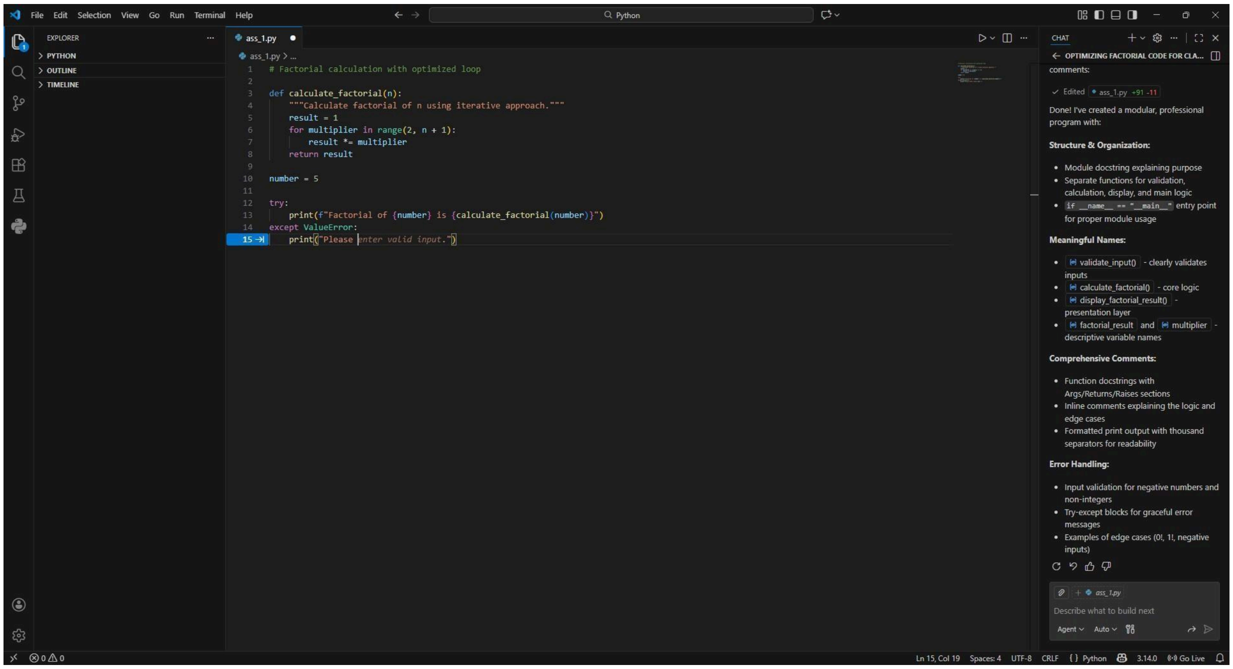
Constraints:

- 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:



```python
# Factorial calculation with optimized loop

def calculate_factorial(n):
    """Calculate factorial of n using iterative approach."""
    result = 1
    for multiplier in range(2, n + 1):
        result *= multiplier
    return result

number = 5

try:
    print(f"Factorial of {number} is {calculate_factorial(number)}")
except ValueError:
    print("Please enter valid input.")
```

comments:

Edited ass_1.py +91 -11

Done! I've created a modular, professional program with:

Structure & Organization:

- Module docstring explaining purpose
- Separate functions for validation, calculation, display, and main logic
- `if __name__ == "__main__"` 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:

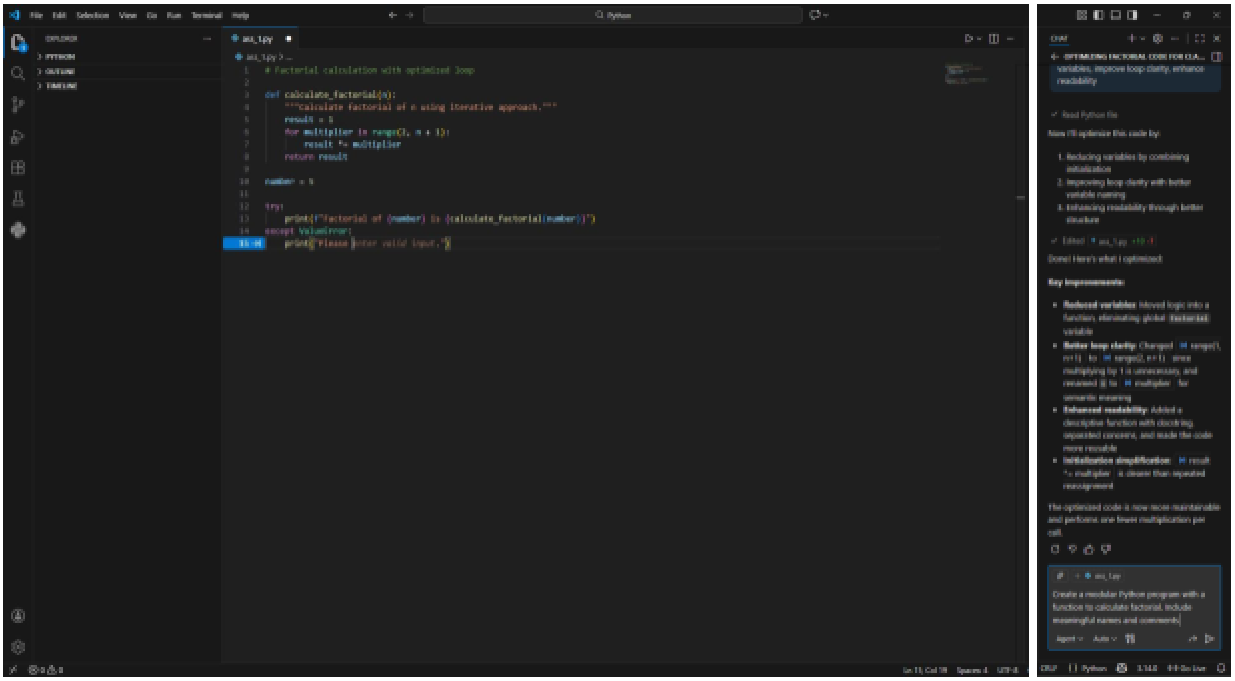
- Function docstrings with Args/Returns/Raises sections
- Inline comments explaining the logic and edge cases
- Formatted print output with thousand separators for readability

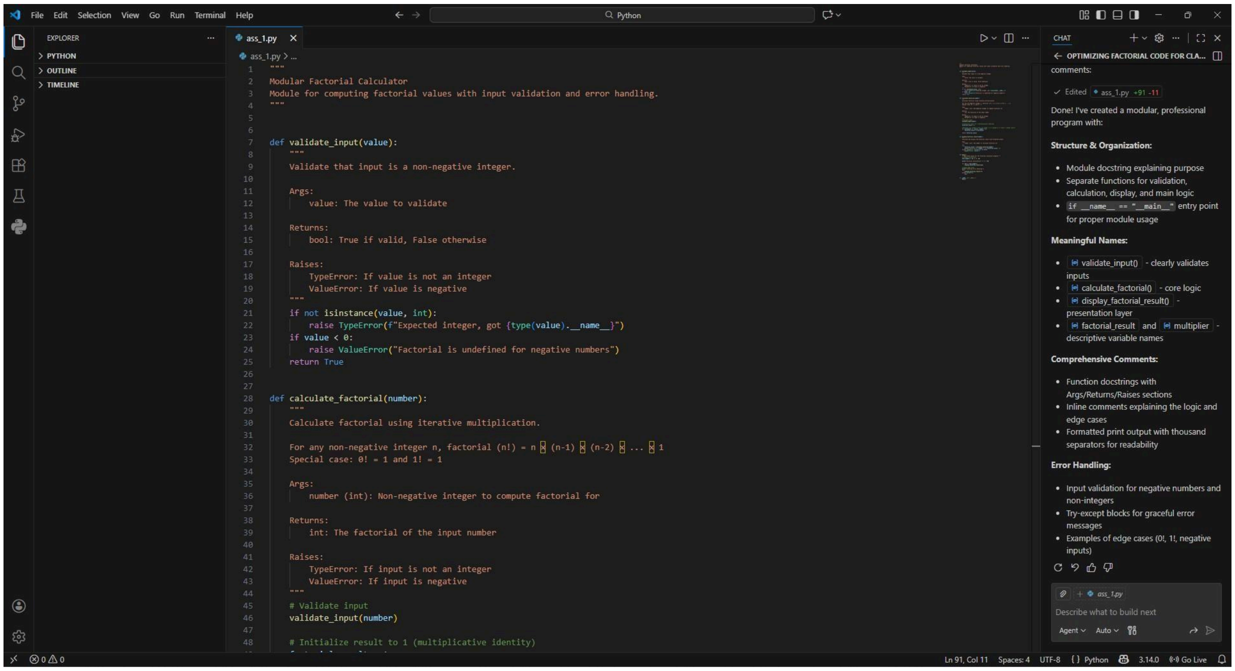
Error Handling:

- Input validation for negative numbers and non-integers
- Try-except blocks for graceful error messages
- Examples of edge cases (0!, 1!, negative inputs)

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





```
"""
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)
```

comments:

✓ Edited ass_1.py +91 -11

Done! I've created a modular, professional program with:

Structure & Organization:

- Module docstring explaining purpose
- Separate functions for validation, calculation, display, and main logic
- `if __name__ == "__main__"` 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)

Describe what to build next

Agent Auto

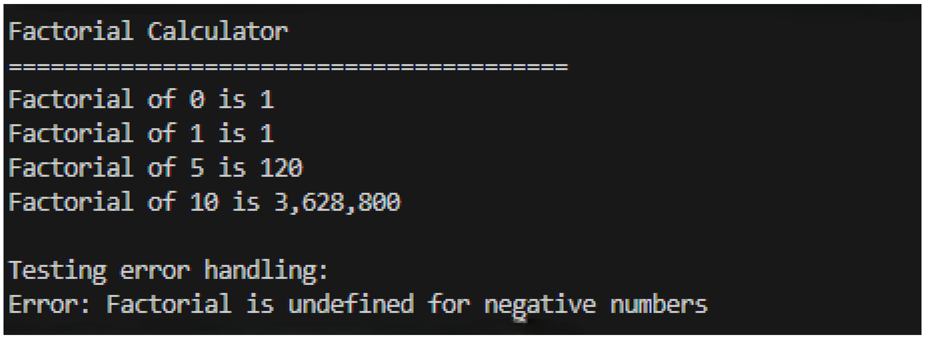
Refactored and improved code:

```
 """
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 risk 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

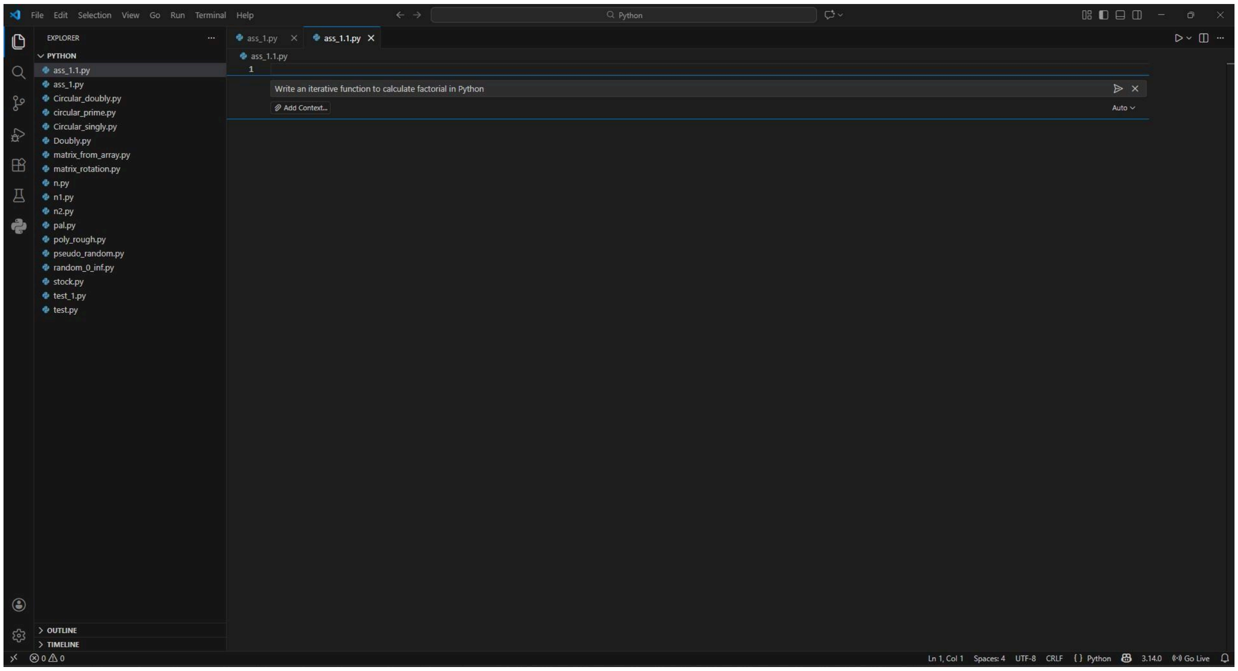
first Expected Deliverables:

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

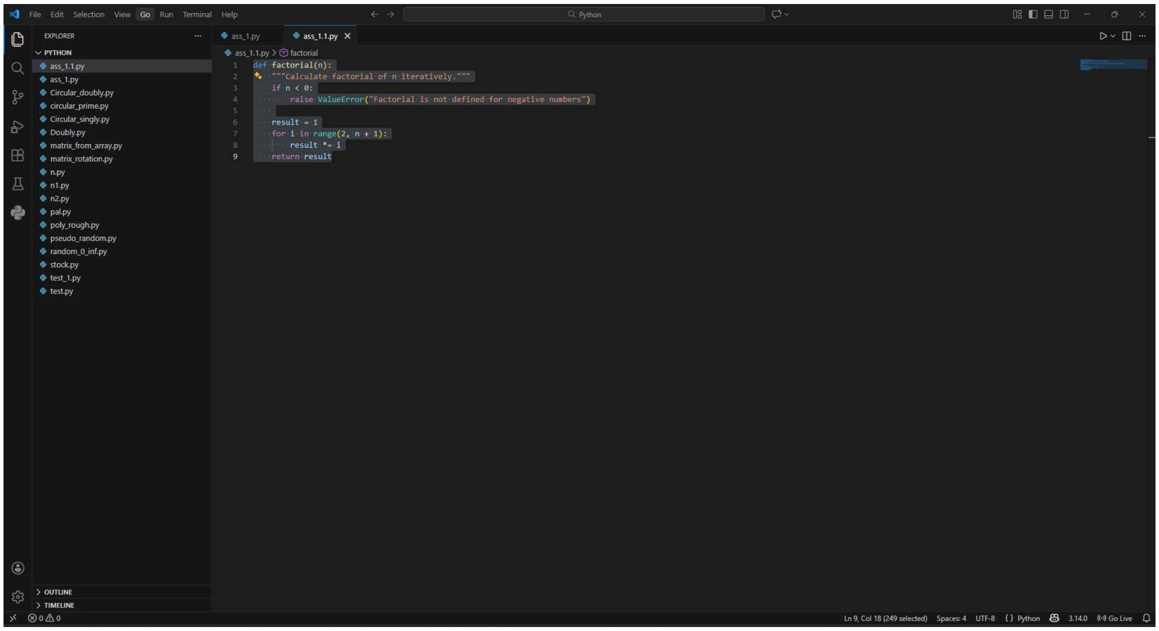
An Iterative version of the code:

Instruction Given:

Write an iterative function to calculate factorial in Python



Write an iterative function to calculate factorial in Python



```python
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
```

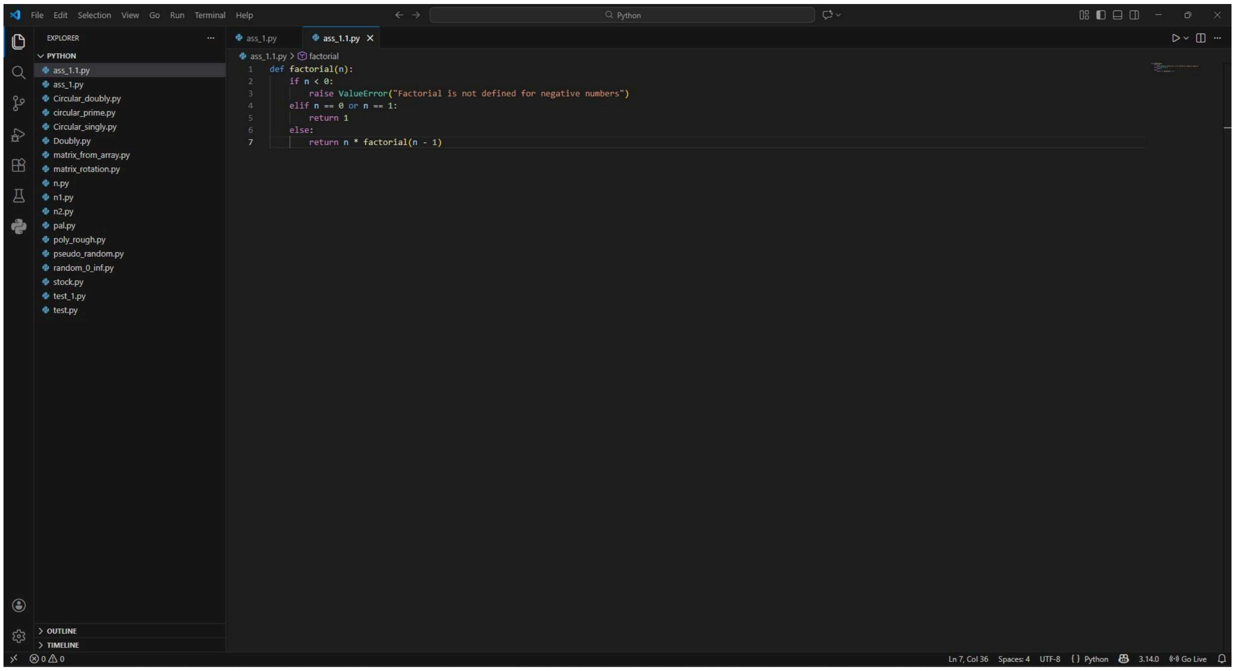
Code:

```
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



```
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)
```

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)
```

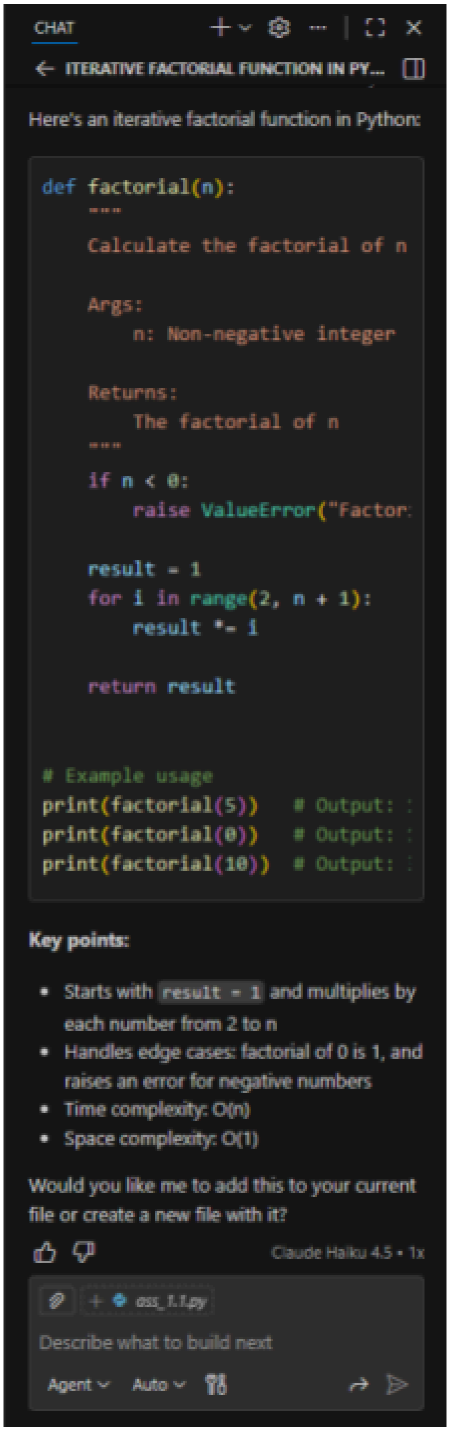
Comparison Iterative vs Recursive:

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
ITERATIVE FACTORIAL FUNCTION IN PY...
Claude Haiku 4.5 • 1x
ass_1.1.py
Describe what to build next
Agent
Auto

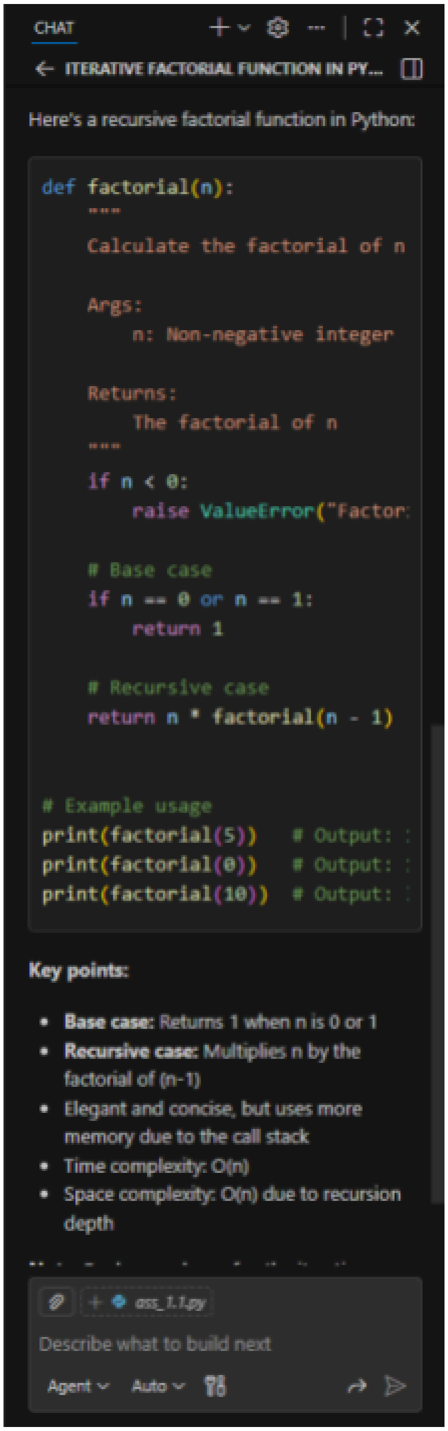
Here's an iterative factorial function in Python:

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

    Args:
        n: Non-negative integer

    Returns:
        The factorial of n
    """
    if n < 0:
        raise ValueError("Factor

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

    return result


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

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?


CHAT
ITERATIVE FACTORIAL FUNCTION IN PY...
ass_1.1.py
Describe what to build next
Agent
Auto

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("Factor

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

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


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

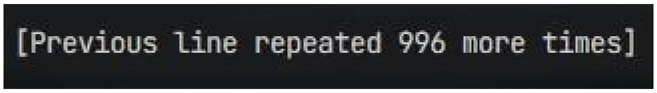
Key points:

- **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

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.