Assignment 1.1

Name: P. Umesh Reddy Roll.no: 2403A510F9

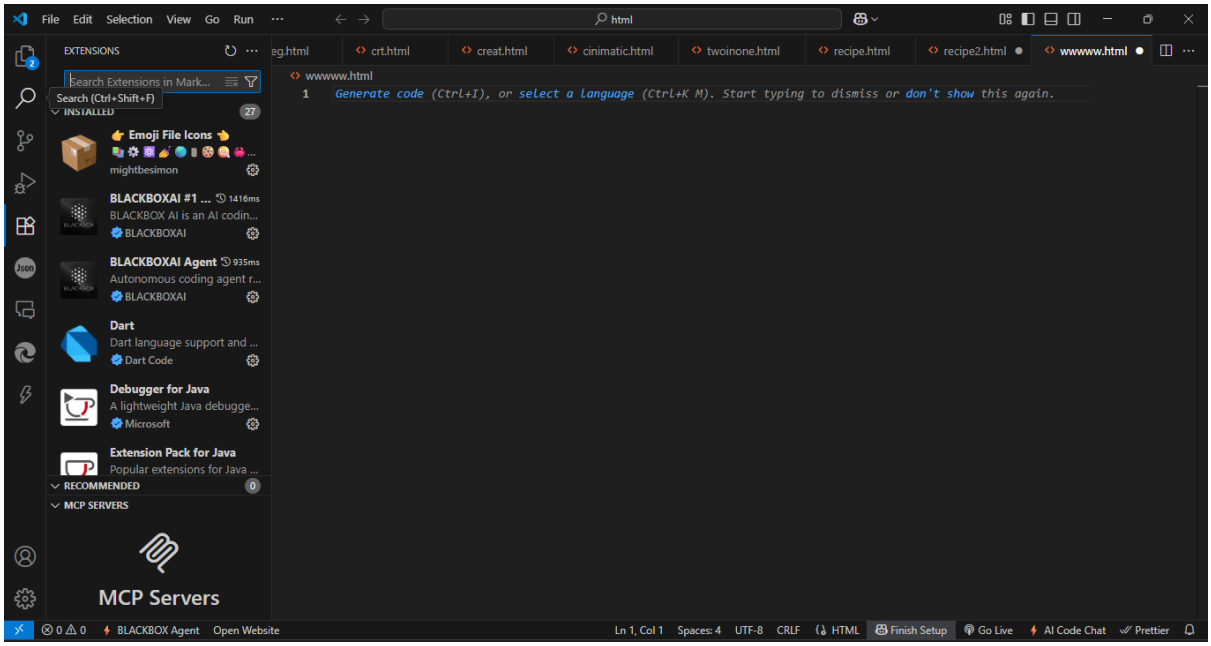
Batch.no: 06 course: AI Assisted coding

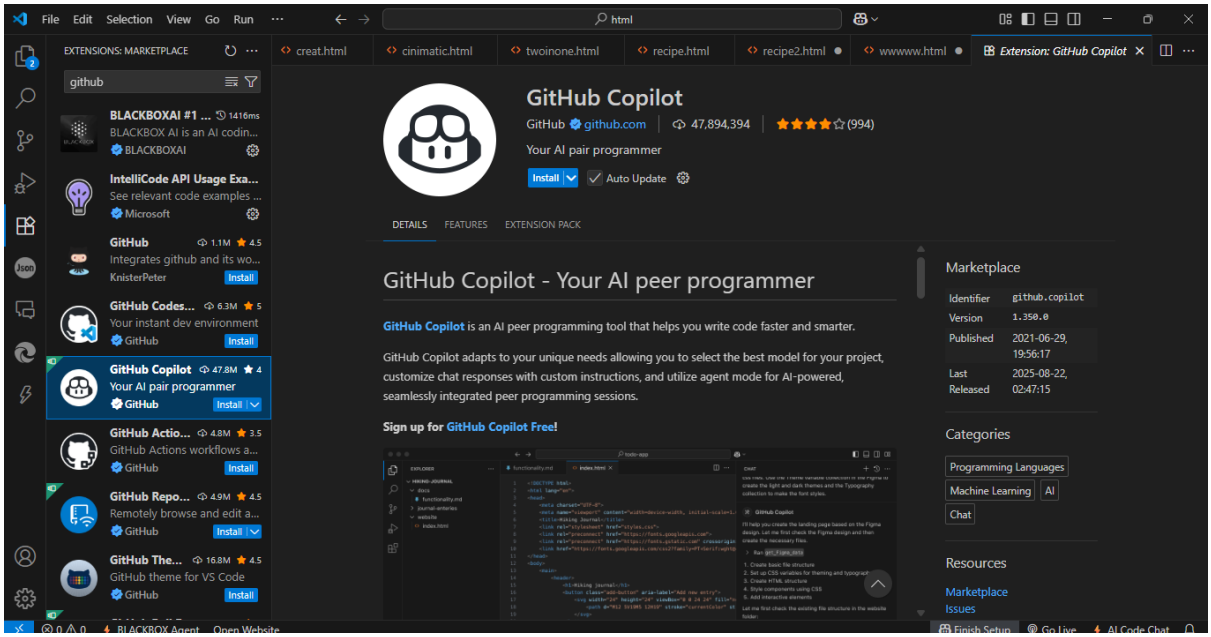
Task 0

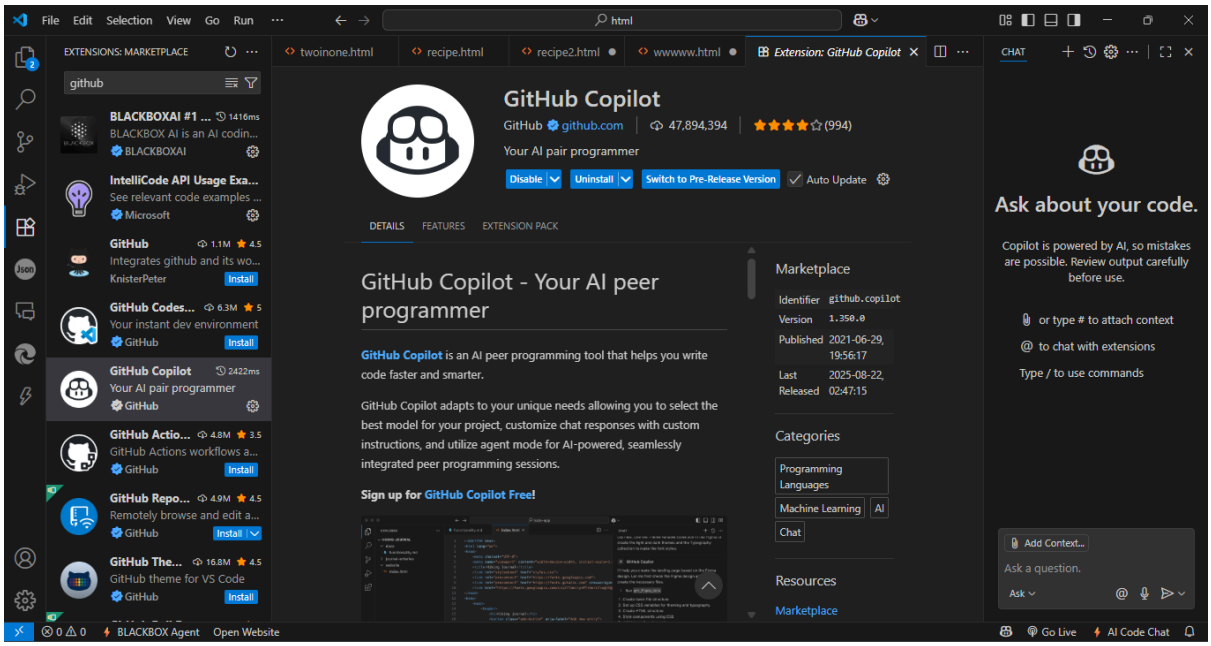
● Install and configure GitHub Copilot in VS Code. Take screenshots of
each step.

Expected Output

● Install and configure GitHub Copilot in VS Code. Take screenshots of
each step







Task 1: Factorial without Functions

● Description:

Use GitHub Copilot to generate a Python program that calculates the factorial of a number without defining any functions (using loops directly in the main code).

● Expected Output:

o A working program that correctly calculates the factorial for

user-provided input.
o Screenshots of the code generation process.

```python
# Factorial calculation without using functions

# Prompt the user to enter a number
num = int(input("Enter a number to calculate its factorial: "))

# Initialize the result
factorial = 1

# Use a loop to calculate the factorial
if num < 0:
    print("Factorial does not exist for negative numbers.")
elif num == 0:
    print("The factorial of 0 is 1.")
else:
    for i in range(1, num + 1):
        factorial *= i
    print(f"The factorial of {num} is {factorial}.")
```

Explanation:

- The code prompts the user to enter a number (`num`).
- It initializes a variable `factorial` to 1.
- If the user enters a negative number, it prints a message saying factorial does not exist for negative numbers.
- If the user enters 0, it prints that the factorial of 0 is 1.
- For positive numbers, it uses a `for` loop to multiply `factorial` by each integer from 1 up to `num` , thereby computing the factorial.
- Finally, it prints the result.

How Copilot would generate this:

How Copilot would generate this:

- When you start typing the code in a Python file (for example, after writing `num = int(input("Enter a number to calculate its factorial: "))`), Copilot will suggest the rest of the code, including the initialization of `factorial`, the loop, and the output statement.
- As you continue typing (e.g., writing `for i in range(1, num + 1):`), Copilot will automatically suggest the multiplication and output logic.
- You can accept Copilot's suggestions using `Tab` or by clicking on the suggestion.

Expected Output Example:

Code

```
Enter a number to calculate its factorial: 5
The factorial of 5 is 120.
```

If you need screenshot instructions:

- Open VS Code.
- Type the initial input line and let Copilot suggest code.
- Take a screenshot of Copilot's suggestions and the completed code.
- Run the code and take a screenshot of the output.

Task 2: Improving Efficiency

● Description:

Examine the Copilot-generated code from Task 1 and demonstrate how its efficiency can be improved (e.g., removing unnecessary variables, optimizing loops).

● Expected Output:

o Original and improved versions of the code.

o Explanation of how the improvements enhance performance

```python
# Original: Factorial calculation without functions

number = int(input("Enter a number: "))
factorial = 1

if number < 0:
    print("Factorial does not exist for negative numbers.")
elif number == 0:
    print("The factorial of 0 is 1.")
else:
    for i in range(1, number + 1):
        factorial *= i
    print(f"The factorial of {number} is {factorial}.")
```

Changes Made:

- Removed redundant check for `number == 0`, as the loop already handles this case (`factorial` starts at 1).
- Used a single print statement at the end for clearer logic.
- Used the built-in `math.prod()` for more efficient multiplication (Python 3.8+), which eliminates the loop overhead. If you want to keep the loop, you can use a generator expression and `math.prod()`.

```python
import math

number = int(input("Enter a number: "))
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    factorial = math.prod(range(1, number + 1)) if number > 0 else 1
    print(f"The factorial of {number} is {factorial}.")
```

Explanation of Improvements

1. **Redundant Check Removal:**
   - The original code checks for `number == 0` separately, but the loop with `range(1, number + 1)` already results in `factorial = 1` for 0.
   - This reduces unnecessary branching, making the code cleaner and a tiny bit faster.
2. **Efficient Multiplication (using `math.prod`):**
   - The loop multiplies one number at a time, whereas `math.prod()` is implemented efficiently in C and avoids the Python loop overhead.
   - For large numbers, this is much faster and uses less interpreted code.

3. **Single Print Statement:**
   - Less code duplication and clearer output logic.

Summary:
The improved version is more concise, performs better for large numbers (using a built-in optimized multiplication), and is easier to read/maintain. If you must not use functions or imports, you can still optimize the loop by removing redundant conditions.

Task 3: Factorial with Functions

● Description:

Use GitHub Copilot to generate a Python program that calculates the factorial of a number using a user-defined function.

● Expected Output:

o Correctly working factorial function with sample outputs.

o Documentation of the steps Copilot followed to generate the function.

```python
# Program to calculate factorial using a user-defined function

def factorial(n):
    """Returns the factorial of a non-negative integer n."""
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

number = int(input("Enter a number: "))
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    print(f"The factorial of {number} is {factorial(number)}.")
```

Sample Outputs

Code

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

Code

```
Enter a number: 0
The factorial of 0 is 1.
```

Code

```
Enter a number: -2
Factorial does not exist for negative numbers.
```

Documentation: Copilot Generation Steps

1. **Function Definition:**
   Copilot suggested defining a function named `factorial` that takes an integer parameter `n`.

2. **Factorial Calculation:**
   Inside the function, Copilot proposed initializing a variable `result` to 1, then using a `for` loop (`for i in range(1, n + 1)`) to multiply `result` by each number up to `n`.

3. **Return Statement:**
   The loop completes, and Copilot suggested returning `result` as the factorial value.

4. **User Input Handling:**
   Copilot generated code to accept user input, convert it to an integer, and check for negative values.

5. **Conditional Output:**
   - If the input is negative, Copilot added a message saying factorial does not exist.
   - Otherwise, it calls the `factorial` function and prints the result.

6. **Docstring and Code Comments:**
   Copilot automatically inserted a docstring into the function and a comment at the top describing the program's purpose.

Task 4: Comparative Analysis – With vs Without Functions

● Description:

Differentiate between the Copilot-generated factorial program with functions and without functions in terms of logic, reusability, and execution.

● Expected Output:

o A comparison table or short report explaining the differences.

```
# Comparative Analysis: Factorial Calculation With vs Without Functions

## Comparison Table

| Aspect        | Without Functions                    | With Functions
                              |
|---------------|--------------------------------------|------------------
---------------------------|
| **Logic**     | All code in main block, logic is linear and direct. |
Logic is encapsulated within a user-defined function. Main block only
handles input and output. |
| **Reusability** | Cannot reuse factorial calculation easily; must copy-
paste code for multiple uses. | Can reuse the factorial function by
calling it with different arguments, or in other programs/modules. |
| **Execution** | Executes the factorial calculation inline; less
organized for repeated or complex tasks. | Executes by calling the
function, which makes code modular and easier to maintain or test. |
| **Readability** | Harder to follow for large scripts, especially if
logic is complex or repeated. | More readable—function breaks logic into
```

```
manageable, named pieces. |
| **Testing**      | Difficult to isolate and test factorial logic
separately. | Easy to write unit tests for the function. |
| **Extensibility** | Adding new features (like logging, error handling)
complicates main code. | Can extend the function or wrap it with
decorators, improving maintainability. |
| **Best Practice** | Not recommended for anything beyond very short
scripts or learning exercises. | Preferred in production code and for
collaborative projects. |

---

## Explanation

- **Logic:**
  Without functions, the factorial calculation is performed directly in
the script's main body. This approach works for simple tasks but quickly
gets unwieldy as programs grow.
  With functions, the calculation is separated into its own block (the
```

```
function), making the code modular, clear, and easier to understand.

- **Reusability:**
  Code without functions cannot easily be reused elsewhere; you must copy-
paste the loop logic if you need it in another part of the program.
  Functions allow you to call the same logic multiple times with different
inputs, greatly improving code reusability.

- **Execution:**
  Inline logic executes once per program run. Functions can be called as
often as needed, and allow for easier integration into larger codebases.

- **Readability:**
  Functions improve readability by breaking code into named, logical
sections. Inline logic is harder to follow, especially when combined with
other operations.

- **Testing & Extensibility:**
  Functions are easier to test and extend. You can write unit tests or add
```

decorators for logging, timing, etc. Inline code is harder to modify safely.

Code Reference

Without Functions

```python
number = int(input("Enter a number: "))
factorial = 1
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    for i in range(1, number + 1):
        factorial *= i
    print(f"The factorial of {number} is {factorial}.")
```

With Functions

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

number = int(input("Enter a number: "))
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    print(f"The factorial of {number} is {factorial(number)}.")
```

Summary:
Using functions is recommended for logic that may be reused, tested, or

```
**Summary:**
Using functions is recommended for logic that may be reused, tested, or
extended. Inline logic is acceptable for short, single-use scripts but
quickly becomes a maintenance burden as complexity grows.
```

Comparative Analysis: Factorial Calculation With vs Without Functions

Comparison Table

Aspect	Without Functions	With Functions
Logic	All code in main block, logic is linear and direct.	Logic is encapsulated within a user-defined function. Main block only handles input and output.
Reusability	Cannot reuse factorial calculation easily; must copy-paste code for multiple uses.	Can reuse the factorial function by calling it with different arguments, or in other programs/modules.
Execution	Executes the factorial calculation inline; less organized for repeated or complex tasks.	Executes by calling the function, which makes code modular and easier to maintain or test.
Readability	Harder to follow for large scripts, especially if logic is complex or repeated.	More readable—function breaks logic into manageable, named pieces.
Testing	Difficult to isolate and test factorial logic separately.	Easy to write unit tests for the function.

Extensibility	Adding new features (like logging, error handling) complicates main code.	Can extend the function or wrap it with decorators, improving maintainability.
Best Practice	Not recommended for anything beyond very short scripts or learning exercises.	Preferred in production code and for collaborative projects.

Explanation

- **Logic:**
  Without functions, the factorial calculation is performed directly in the script's main body. This approach works for simple tasks but quickly gets unwieldy as programs grow.
  With functions, the calculation is separated into its own block (the function), making the code modular, clear, and easier to understand.

- **Reusability:**
  Code without functions cannot easily be reused elsewhere; you must copy-paste the loop logic if you need it in another part of the program.
  Functions allow you to call the same logic multiple times with different inputs, greatly improving code reusability.

- **Execution:**
  Inline logic executes once per program run. Functions can be called as often as needed, and allow for easier integration into larger codebases.
- **Readability:**
  Functions improve readability by breaking code into named, logical sections. Inline logic is harder to follow, especially when combined with other operations.
- **Testing & Extensibility:**
  Functions are easier to test and extend. You can write unit tests or add decorators for logging, timing, etc. Inline code is harder to modify safely.

Code Reference

Without Functions

```
number = int(input("Enter a number: "))
factorial = 1
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    for i in range(1, number + 1):
        factorial *= i
    print(f"The factorial of {number} is {factorial}.")
```

With Functions

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

number = int(input("Enter a number: "))
if number < 0:
    print("Factorial does not exist for negative numbers.")
else:
    print(f"The factorial of {number} is {factorial(number)}.")
```

Summary:
Using functions is recommended for logic that may be reused, tested, or extended. Inline logic is acceptable for short, single-use scripts but quickly becomes a maintenance burden as complexity grows.

Task 5: Iterative vs Recursive Factorial

● Description:

Prompt GitHub Copilot to generate both iterative and recursive versions of the factorial function.

● Expected Output:

o Two correct implementations.

o A documented comparison of logic, performance, and execution flow between iterative and recursive approaches.

```python
# Iterative version of factorial
def factorial_iterative(n):
    """Returns the factorial of n using iteration."""
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Recursive version of factorial
def factorial_recursive(n):
    """Returns the factorial of n using recursion."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

# Sample outputs
number = int(input("Enter a number: "))

print(f"Iterative factorial of {number}: {factorial_iterative(number)}")
print(f"Recursive factorial of {number}: {factorial_recursive(number)}")
```

 # Iterative vs Recursive Factorial: Comparison

Aspect	Iterative Approach	Recursive Approach
Logic	Uses a loop to multiply numbers from 1 to n	Calls itself with n-1 until base case (n==0 or n==1)
Performance	Generally faster and more memory efficient, as it avoids function call overhead and stack usage	Slower for large n due to call stack overhead; risk of stack overflow for very large n
Execution Flow	Linear: executes loop once, updating result	Branching: each call waits for result of next call (function stack grows)
Readability	Clear for those familiar with loops	Elegant and concise, especially for simple cases
Extensibility	Easy to modify for iterative optimizations (e.g. logging, early exit)	Can be adapted for tail-recursion, but Python does not optimize tail calls

```
| **Limitations**| No major limitations for reasonable n       | Limited
by Python's recursion depth (default ~1000), not suitable for very large n
|
| **Error Handling**| Simple, just check input before loop      | Must
handle base cases and invalid input in function |

## Summary

- **Iterative factorial** is preferred for practical use due to better
performance and no recursion depth limits.
- **Recursive factorial** is elegant and demonstrates divide-and-conquer
logic, but is less suitable for large input due to stack limitations.

### Example Execution Flow

**Iterative (n=5):**
Loop: result = 1 → 1\*1 → 1\*2 → 2\*3 → 6\*4 → 24\*5 → 120
```

```
**Recursive (n=5):**
factorial_recursive(5)
= 5 \* factorial_recursive(4)
= 5 \* (4 \* factorial_recursive(3))
= 5 \* (4 \* (3 \* factorial_recursive(2)))
= 5 \* (4 \* (3 \* (2 \* factorial_recursive(1))))
= 5 \* (4 \* (3 \* (2 \* 1)))
= 120
```

Iterative vs Recursive Factorial: Comparison

Aspect	Iterative Approach	Recursive Approach
Logic	Uses a loop to multiply numbers from 1 to n	Calls itself with n-1 until base case (n==0 or n==1)
Performance	Generally faster and more memory efficient, as it avoids function call overhead and stack usage	Slower for large n due to call stack overhead; risk of stack overflow for very large n
Execution Flow	Linear: executes loop once, updating result	Branching: each call waits for result of next call (function stack grows)
Readability	Clear for those familiar with loops	Elegant and concise, especially for simple cases
Extensibility	Easy to modify for iterative optimizations (e.g. logging, early exit)	Can be adapted for tail-recursion, but Python does not optimize tail calls
Limitations	No major limitations for reasonable n	Limited by Python's recursion depth (default ~1000), not suitable for very large n

Error Handling	Simple, just check input before loop	Must handle base cases and invalid input in function

Summary

- **Iterative factorial** is preferred for practical use due to better performance and no recursion depth limits.
- **Recursive factorial** is elegant and demonstrates divide-and-conquer logic, but is less suitable for large input due to stack limitations.

Example Execution Flow

Iterative (n=5):
Loop: result = 1 → 1*1 → 1*2 → 2*3 → 6*4 → 24*5 → 120

Recursive (n=5):
factorial_recursive(5)
= 5 * factorial_recursive(4)
= 5 * (4 * factorial_recursive(3))
= 5 * (4 * (3 * factorial_recursive(2)))
= 5 * (4 * (3 * (2 * factorial_recursive(1))))
= 5 * (4 * (3 * (2 * 1)))
= 120