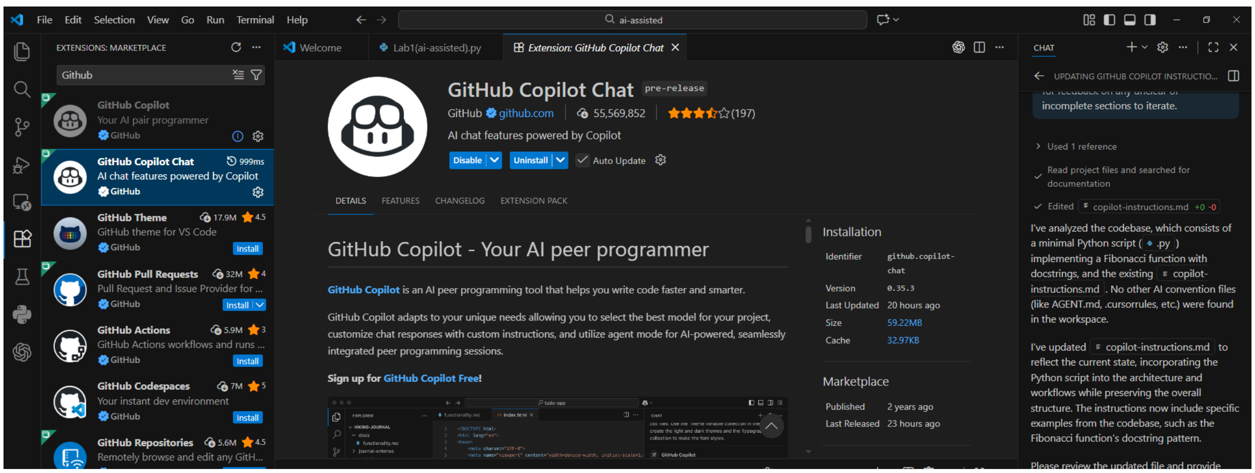
Lab - 01 Assignment

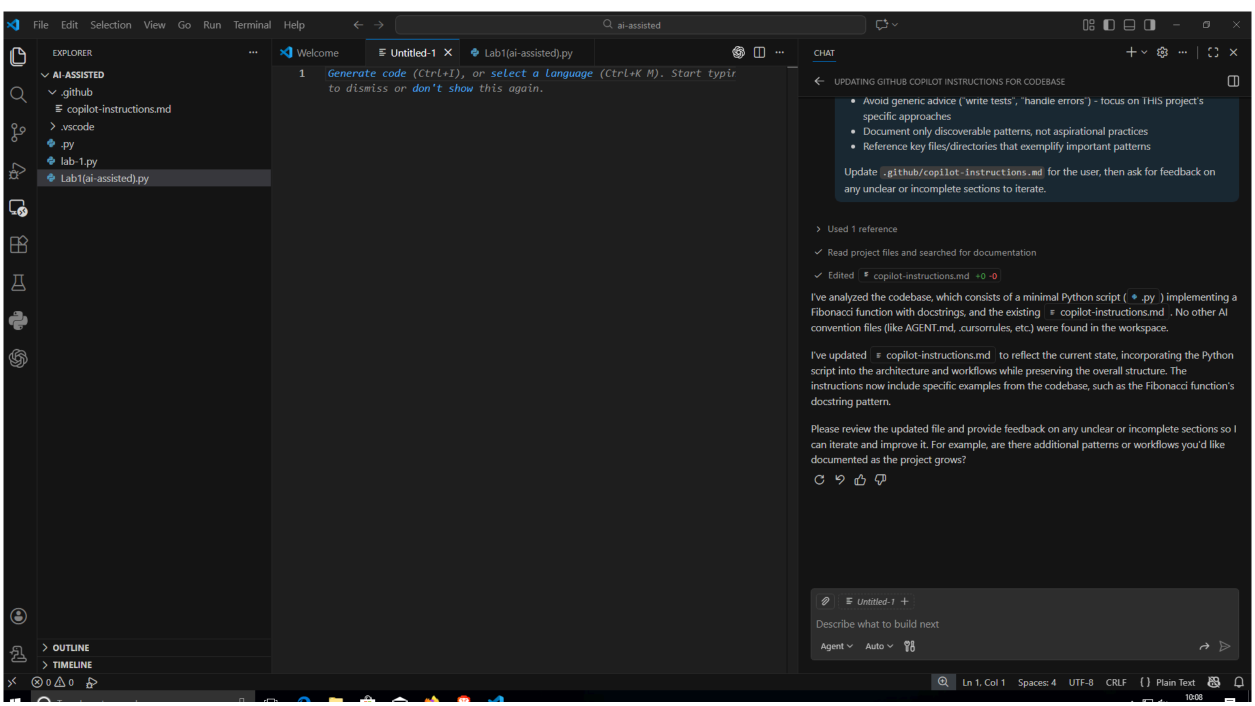
Name: Zobiya Fatima
Roll No: 2303A51879
Batch No: 14

Task 0

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



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



Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

❖ Scenario

➢ You are developing a basic validation script for a numerical learning Application.

❖ Task Description

Use GitHub Copilot to generate a Python program that:

➢ Checks whether a given number is prime

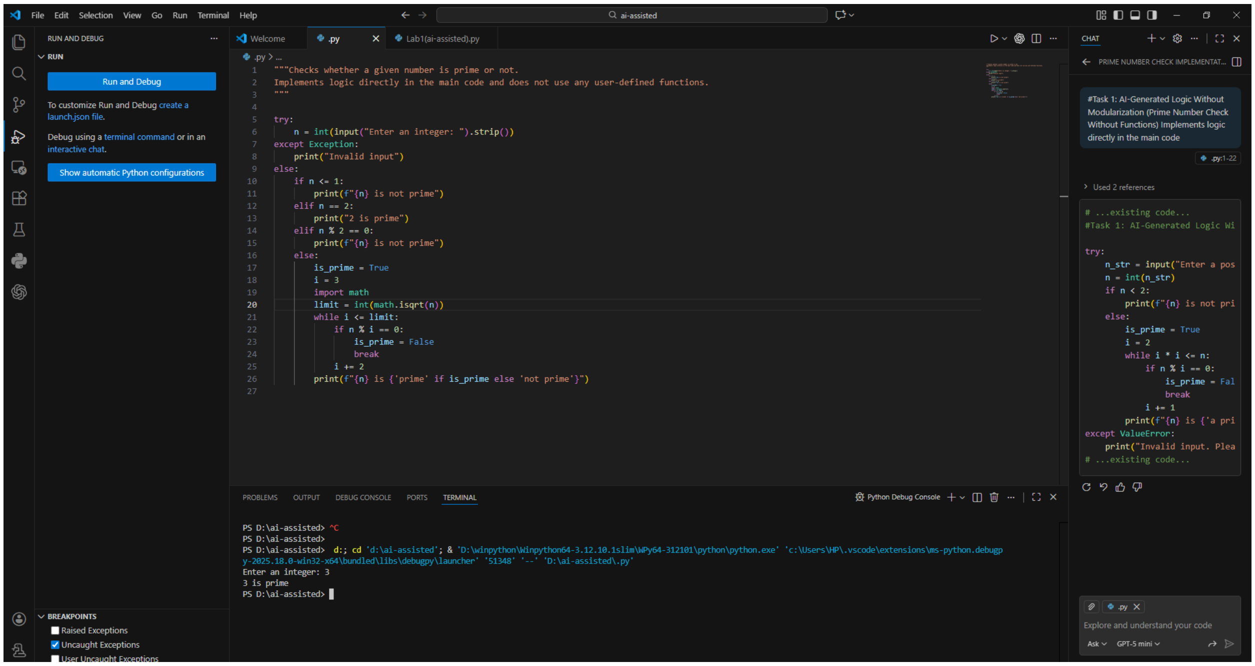
➢ Accepts user input

➢ Implements logic directly in the main code

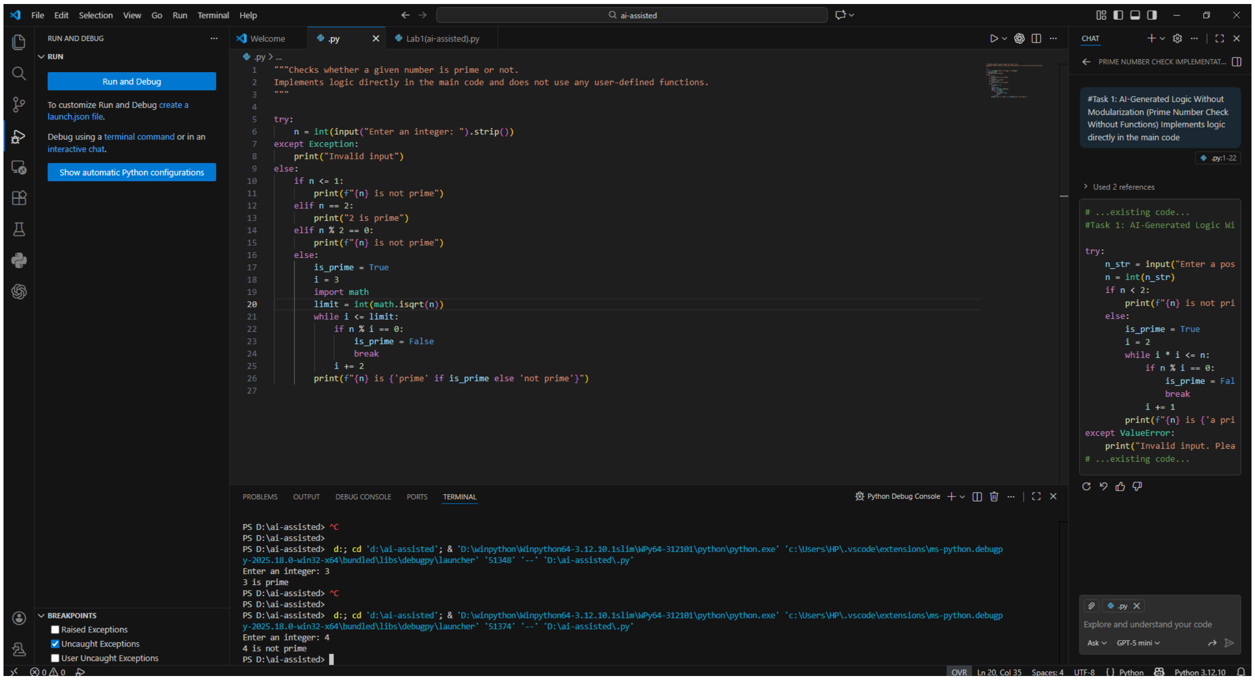
➢ Does not use any user-defined functions

❖ Expected Output

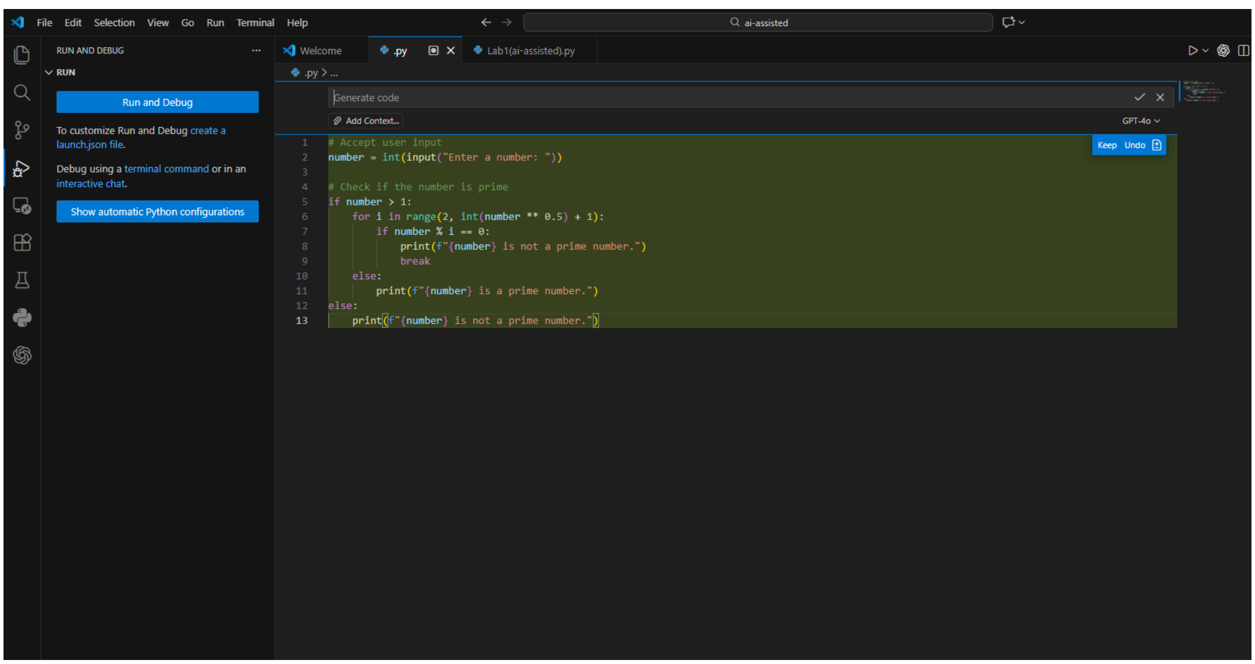
➢ Correct prime result



- ➢ Non-prime result



- ➢ Screenshots showing Copilot-generated code suggestions



Task 2: Efficiency & Logic Optimization (Cleanup)

❖ Scenario

The script must handle larger input values efficiently.

❖ Task Description

Review the Copilot-generated code from Task 1 and improve it by:

➢ Reducing unnecessary iterations

➢ Optimizing the loop range (e.g., early termination)

➢ Improving readability

➢ Use Copilot prompts like:

▪ "Optimize prime number checking logic"

▪ "Improve efficiency of this code"

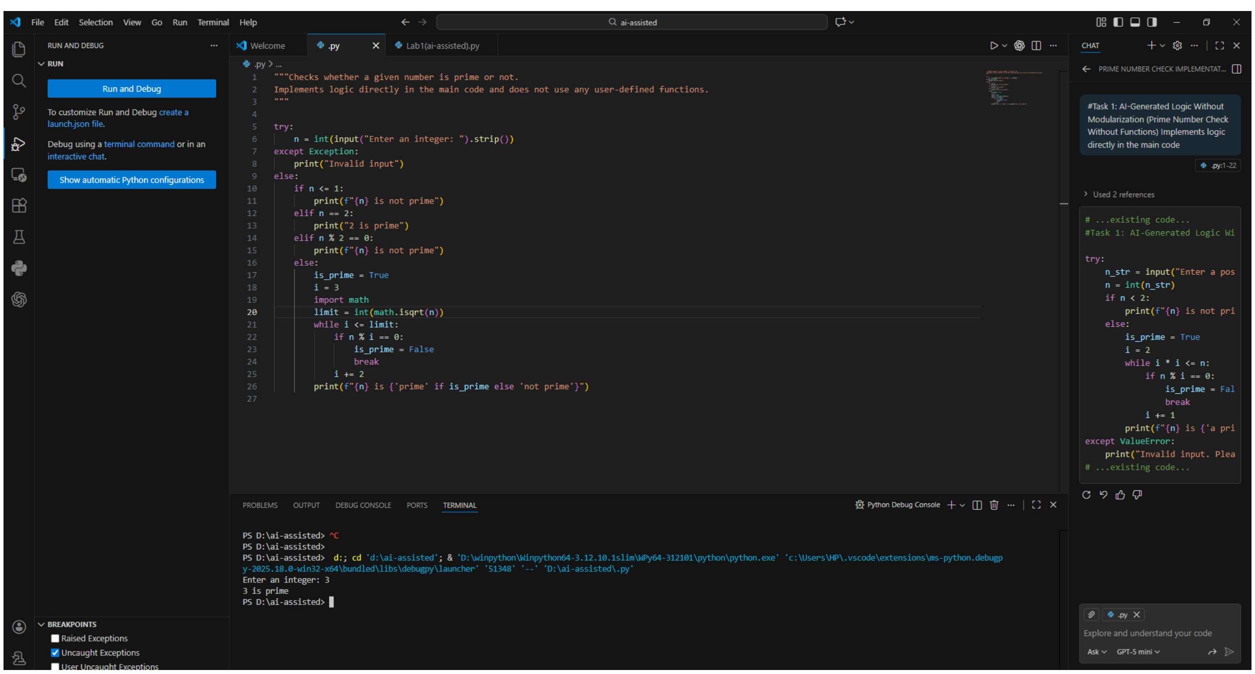
Hint:

Prompt Copilot with phrases like

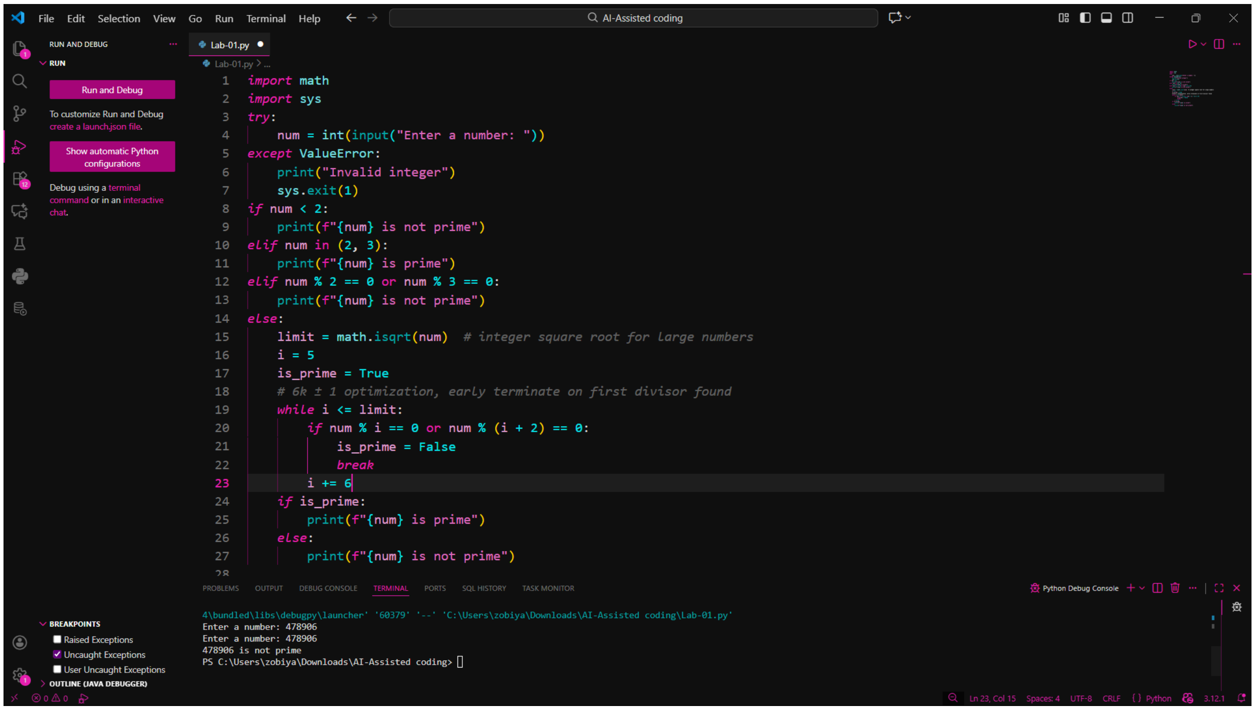
"optimize this code", "simplify logic", or "make it more readable"

❖ Expected Output

➢ Original



➢ Optimized code versions



➢ Explanation of how the improvements reduce time complexity

Time Complexity Improvements:

1. Early exit for n<2, n∈{2,3}, and multiples of 2 or 3: O(1) checks for common cases.

2. Integer square root (isqrt): test divisors only up to √n, reducing checks from O(n) to O(√n).

3. 6k±1 wheel: after removing 2 and 3, all primes have form 6k±1. Test only ~1/3 of candidates.

- Each iteration checks i and i+2, then jumps by 6, avoiding multiples of 2 and 3

4. Early termination: break on first divisor found, so composites with small factors exit quickly.

Result: Worst-case O(√n), but with constant factor ~1/3 vs. naive trial division. Efficient for moderate integers.

Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

❖ Scenario

The prime-checking logic will be reused across multiple modules.

❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

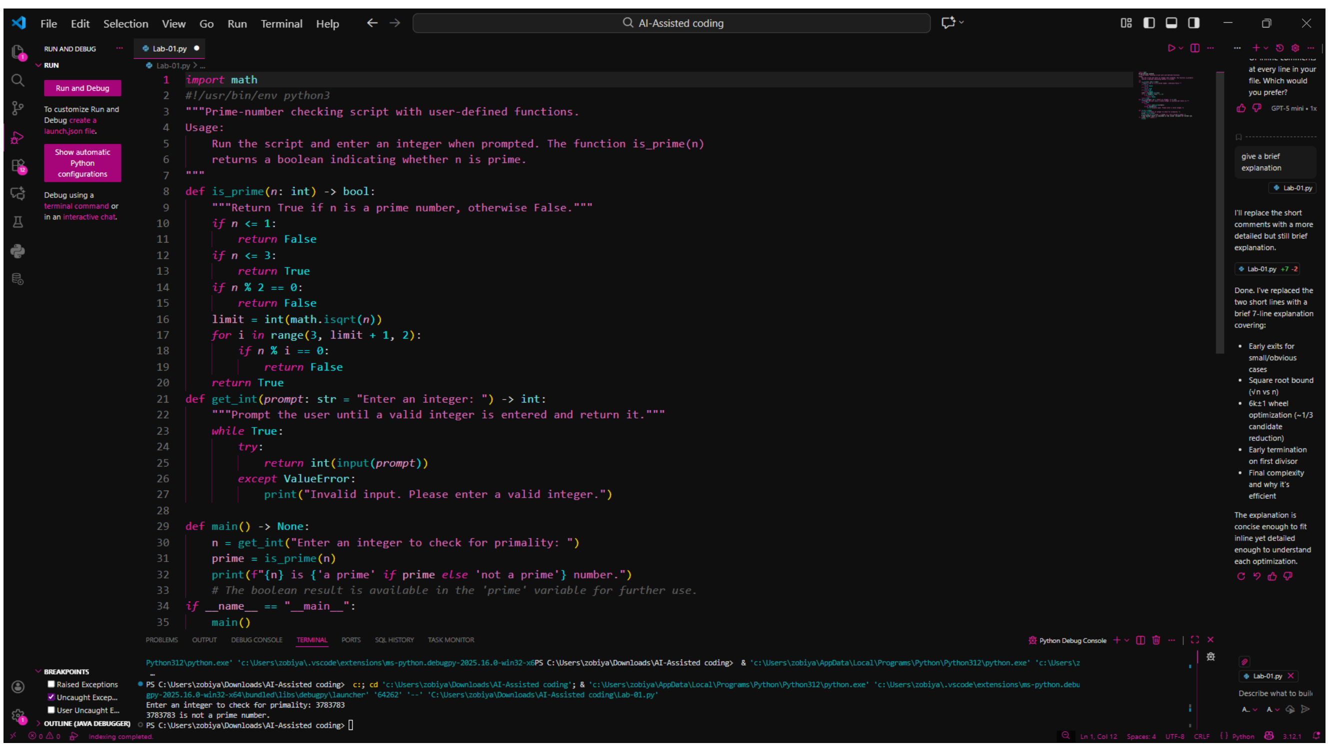
➢ Uses a user-defined function to check primality

➢ Returns a Boolean value

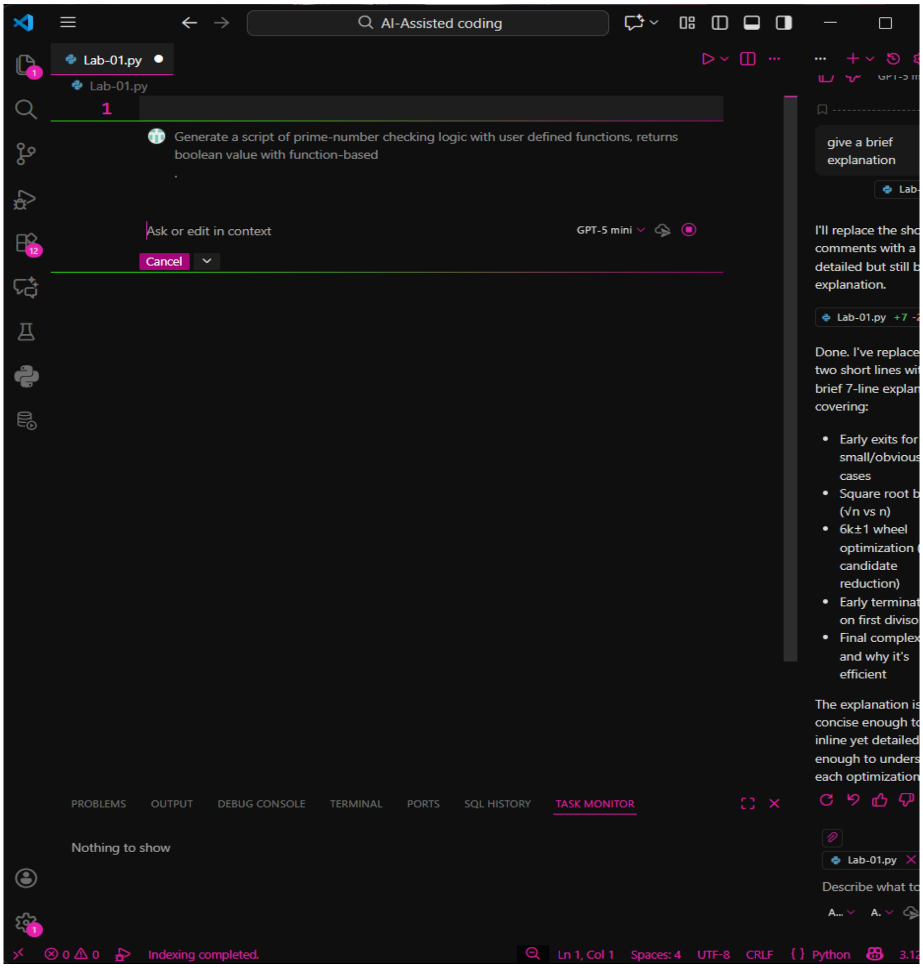
➢ Includes meaningful comments (AI-assisted)

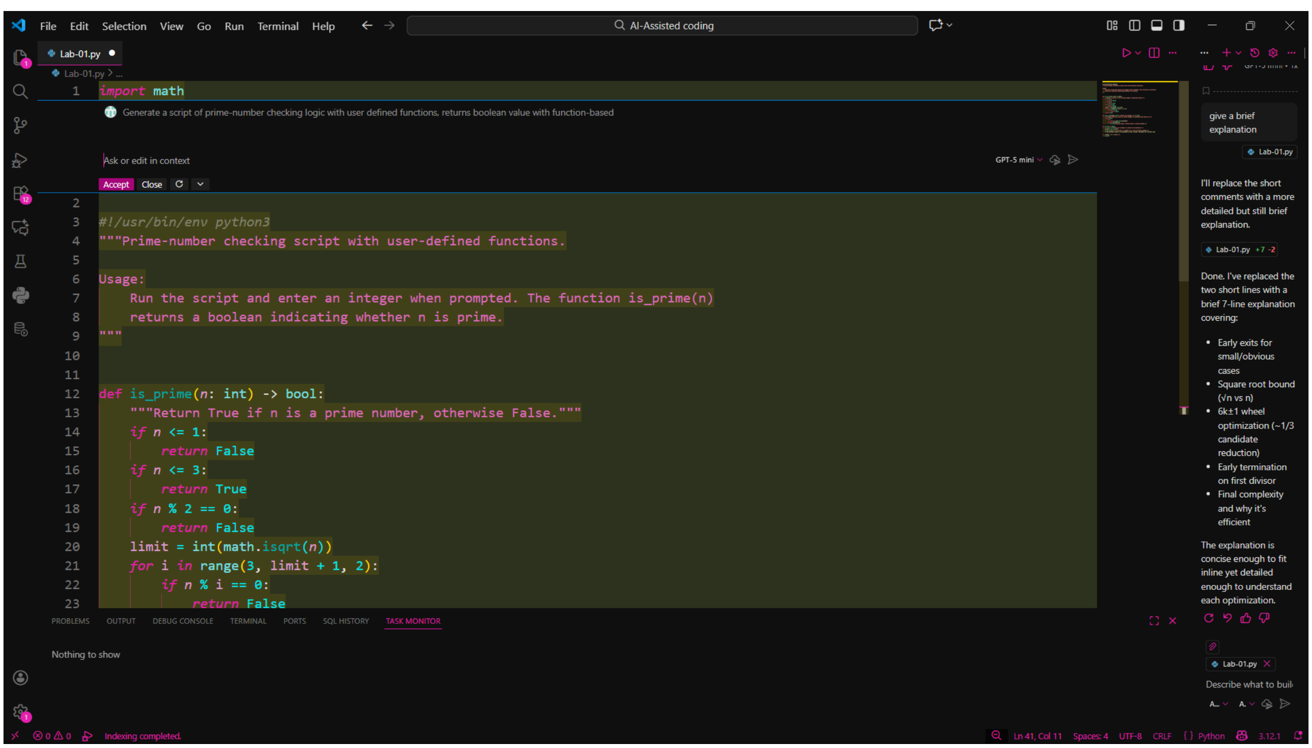
❖ Expected Output

➢ Correctly working prime-checking function

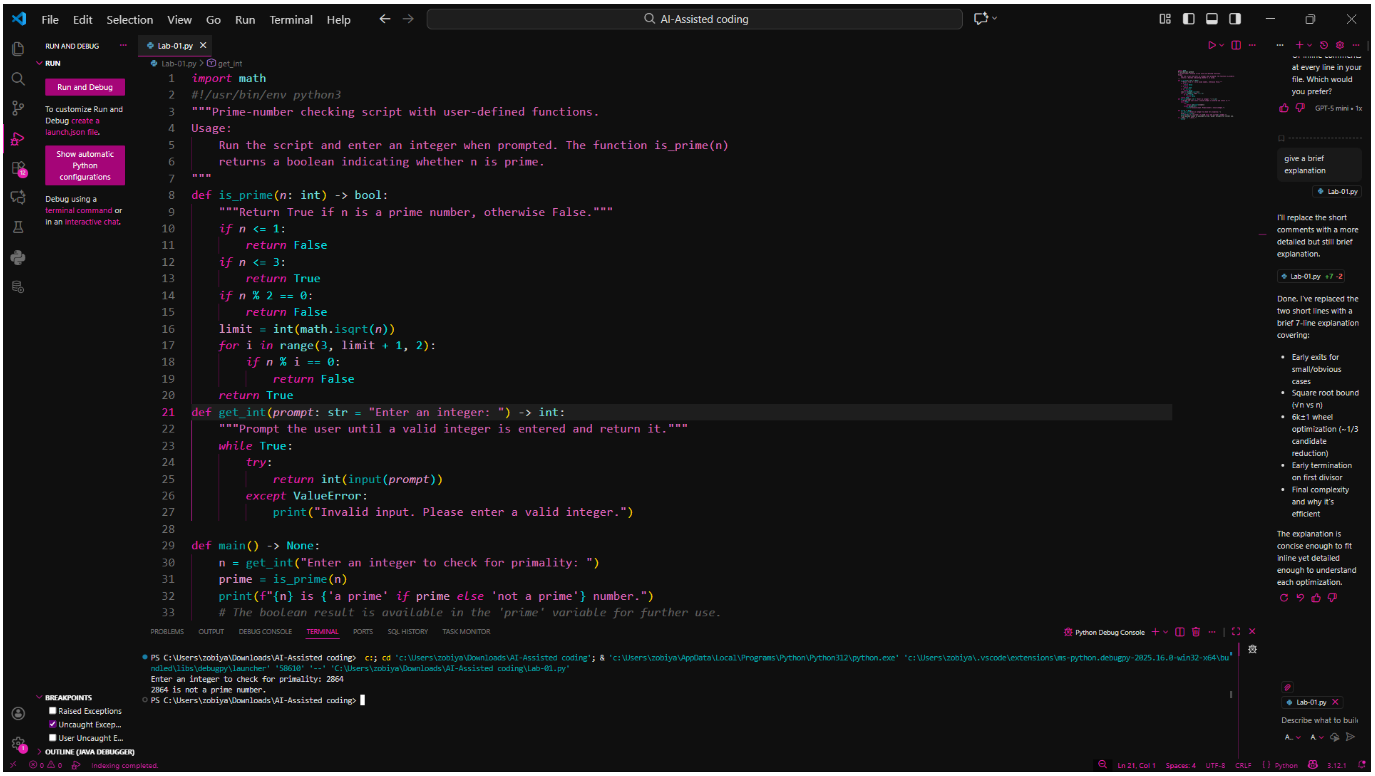


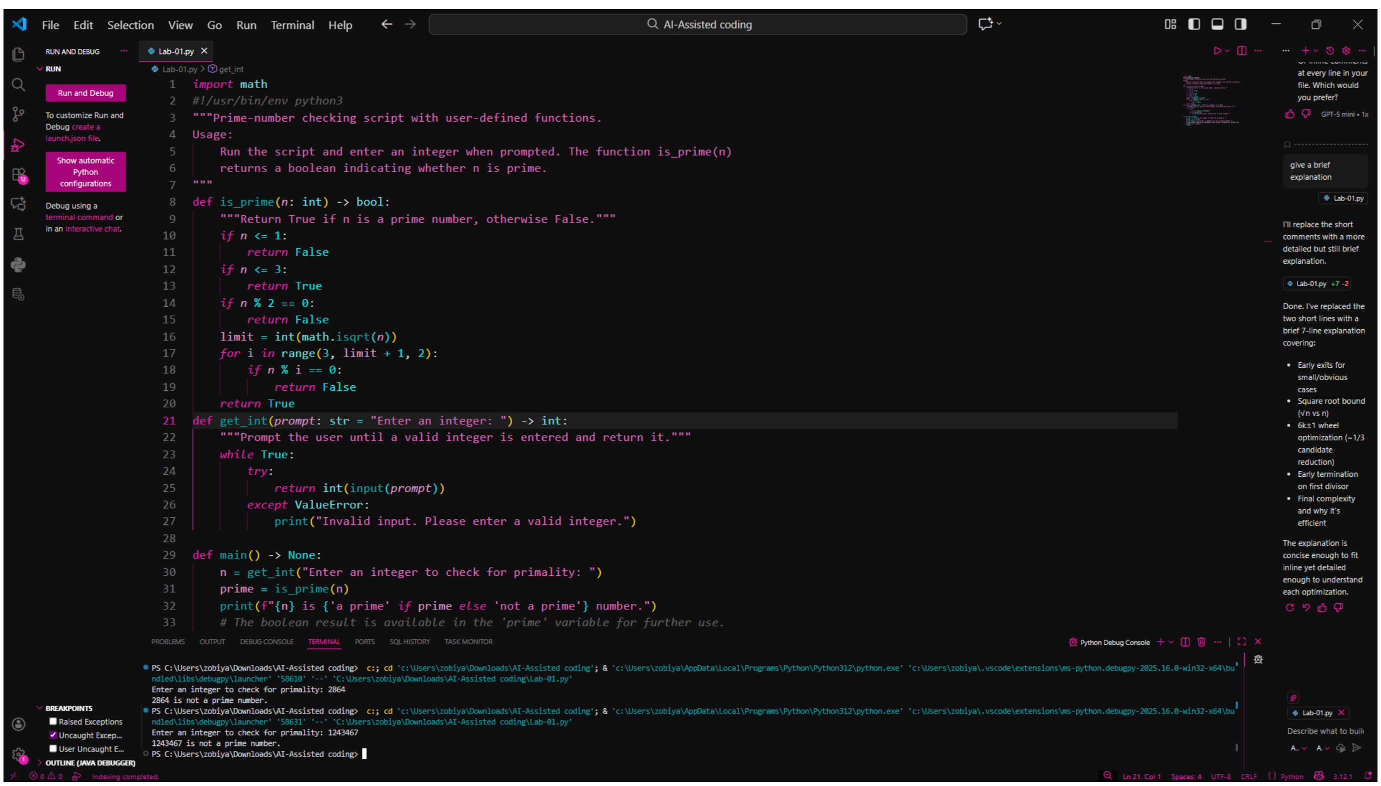
➢ Screenshots documenting Copilot's function generation





➢ Sample test cases and outputs





Task 4: Comparative Analysis –With vs Without Functions

❖ Scenario

You are participating in a technical review discussion.

❖ Task Description

Compare the Copilot-generated programs:

➢ Without functions (Task 1)

➢ With functions (Task 3)

➢ Analyze them based on:

➢ Code clarity

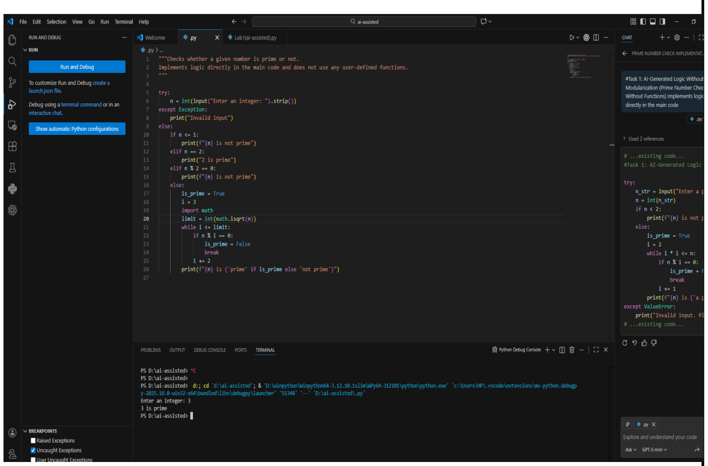
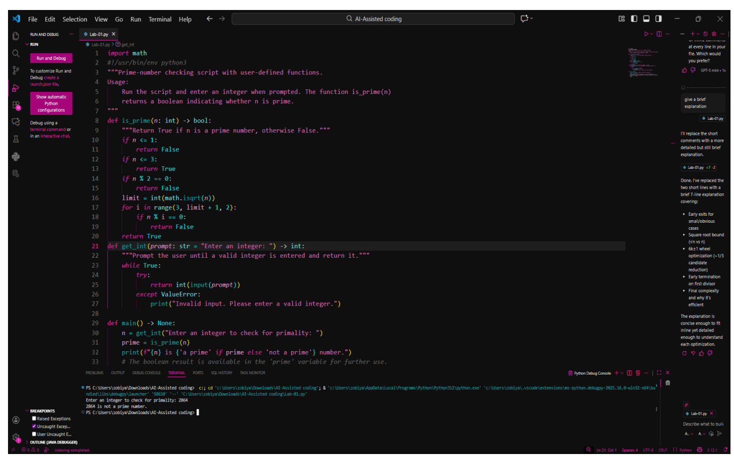
➢ Reusability

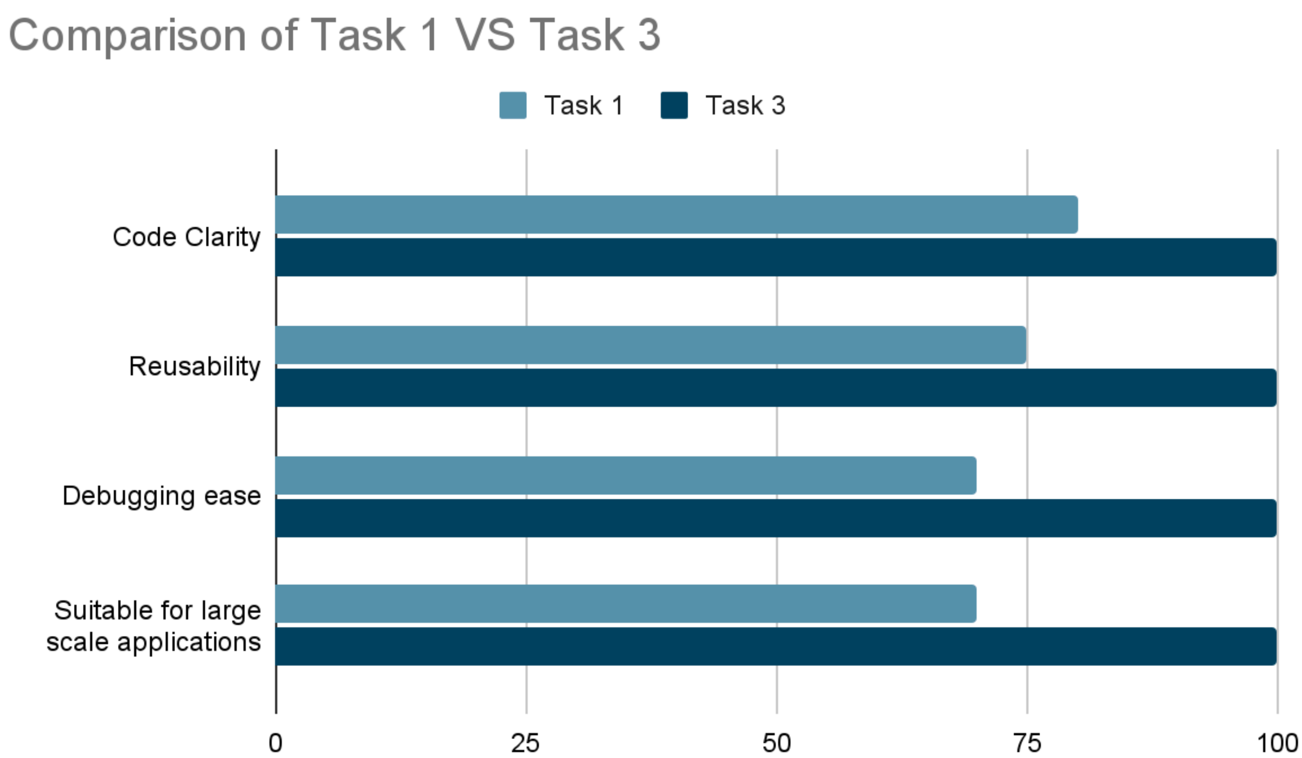
➢ Debugging ease

➢ Suitability for large-scale applications

❖ Expected Output

Comparison table or short analytical report

	
Task 1	**Task 3**
• Logic is correct, but dense • Many things happen in one file scope • Variable reuse (`is_prime` as both function concept and variable) can confuse beginners • Harder to skim	• Meaningful function names: `is_prime`, `get_int` • Type hints (`n: int -> bool`) improve readability • Docstrings explain *what* each function does • Logic is broken into small, understandable units



Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)

❖ Scenario

Your mentor wants to evaluate how AI handles alternative logical strategies.

❖ Task Description

Prompt GitHub Copilot to generate:

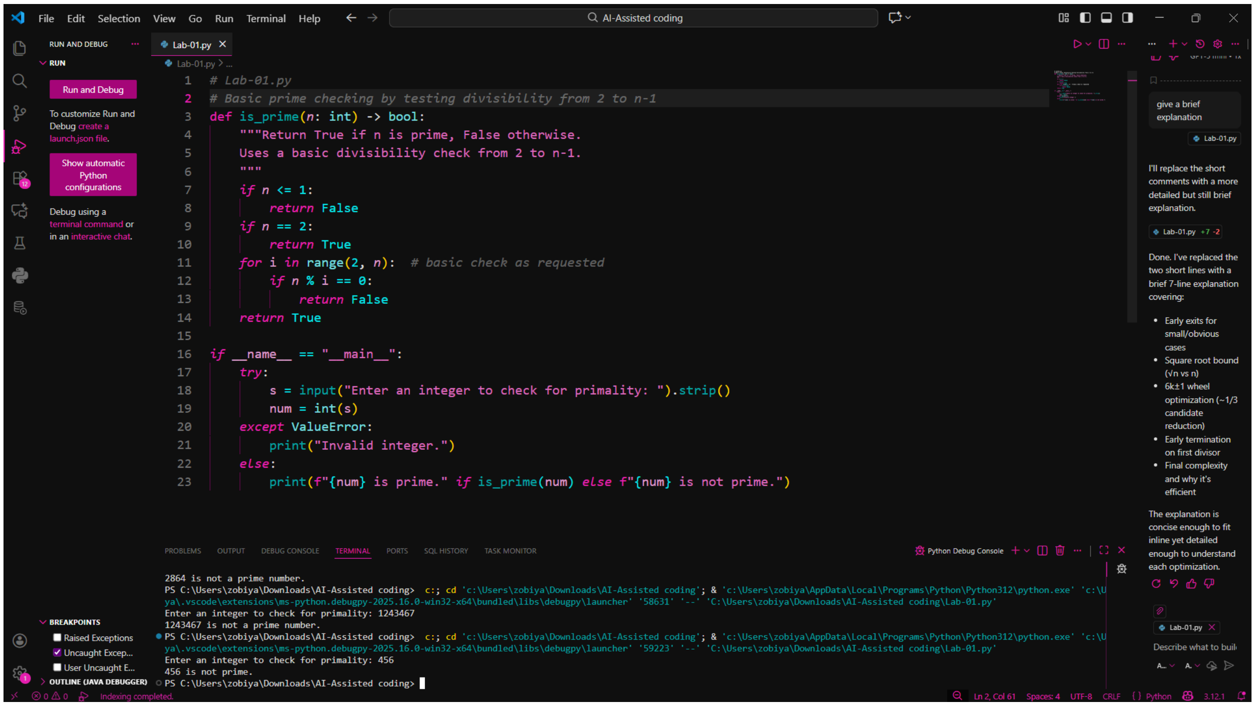
➢ A basic divisibility check approach

➢ An optimized approach (e.g., checking up to √n)

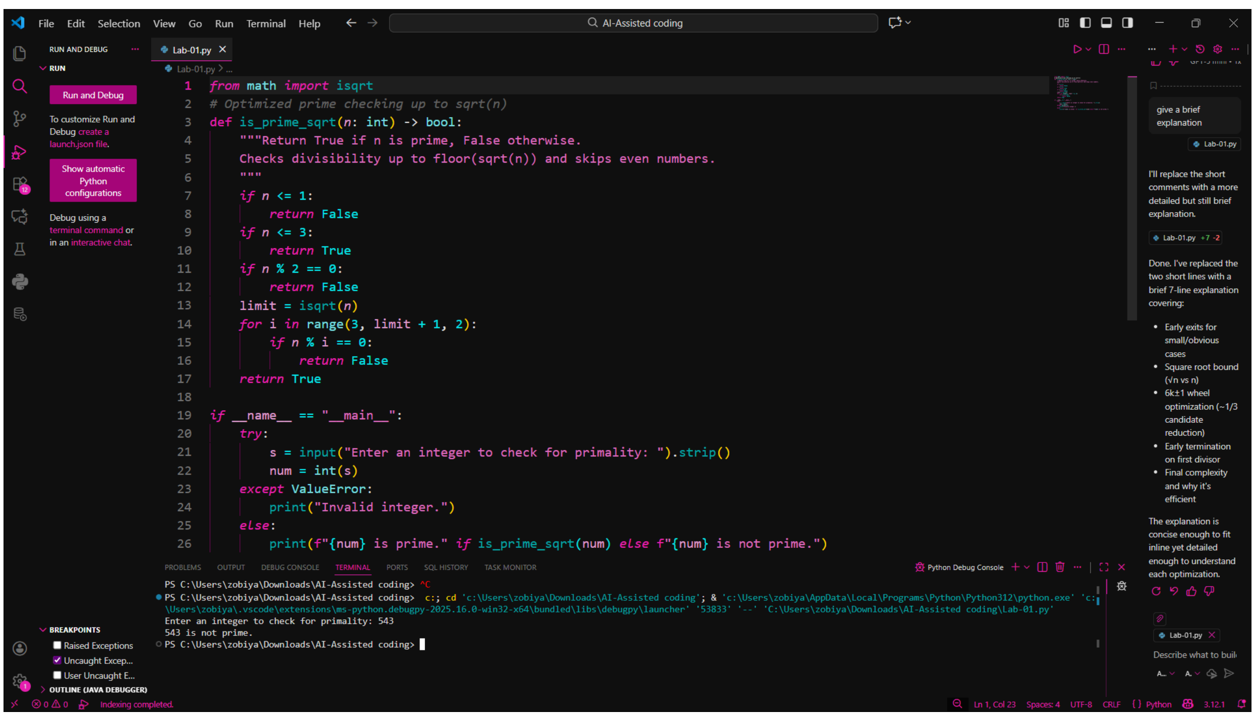
❖ Expected Output

➢ Two correct implementations

- # Approach 1:



- # Approach 2:



➢ Comparison discussing:

Basic divisibility check	Optimized Approach
▪ **Execution flow:** The algorithm checks divisibility of the number by every integer from 2 up to n−1. It stops immediately when a divisor is found.	▪ **Execution flow:** The algorithm checks divisibility only up to √n and skips even numbers. This reduces the number of iterations significantly.
▪ **Time complexity:** Time complexity is **O(n)** since it may check almost all numbers up to n.	▪ **Time complexity:** Time complexity is **O(√n)**, which is much more efficient.
▪ **Performance for large inputs:** Performance degrades significantly as input size increases. Large numbers take a long time to evaluate.	▪ **Performance for large inputs:** Performs well even for large input values. Execution time increases slowly as n grows.
▪ **When each approach is appropriate:** Suitable for learning purposes and very small input values. Useful when simplicity is more important than efficiency.	▪ **When each approach is appropriate:** Suitable for real-world applications and large datasets. Preferred when performance and scalability are important.