Name: G Devadas
H.NO: 2303A53014 - B46

Lab 9 – Code Review and Quality: Using AI to improve code quality and readability

Lab Objectives:

- To apply AI-based prompt engineering for code review and quality improvement.
- To analyze code for readability, logic, performance, and maintainability issues.
- To use Zero-shot, One-shot, and Few-shot prompting for improving code quality.
- To evaluate AI-generated improvements using standard coding practices.

Lab Outcomes (LOs): After completing this lab, students will be able to:

- Review and improve code quality using AI tools.
- Identify syntax, logic, and performance issues in code.
- Refactor code to improve readability and maintainability.
- Compare AI outputs generated using different prompting techniques.

Problem Statement 1: AI-Assisted Bug Detection

Scenario: A junior developer wrote the following Python function to calculate factorials:

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

Instructions:

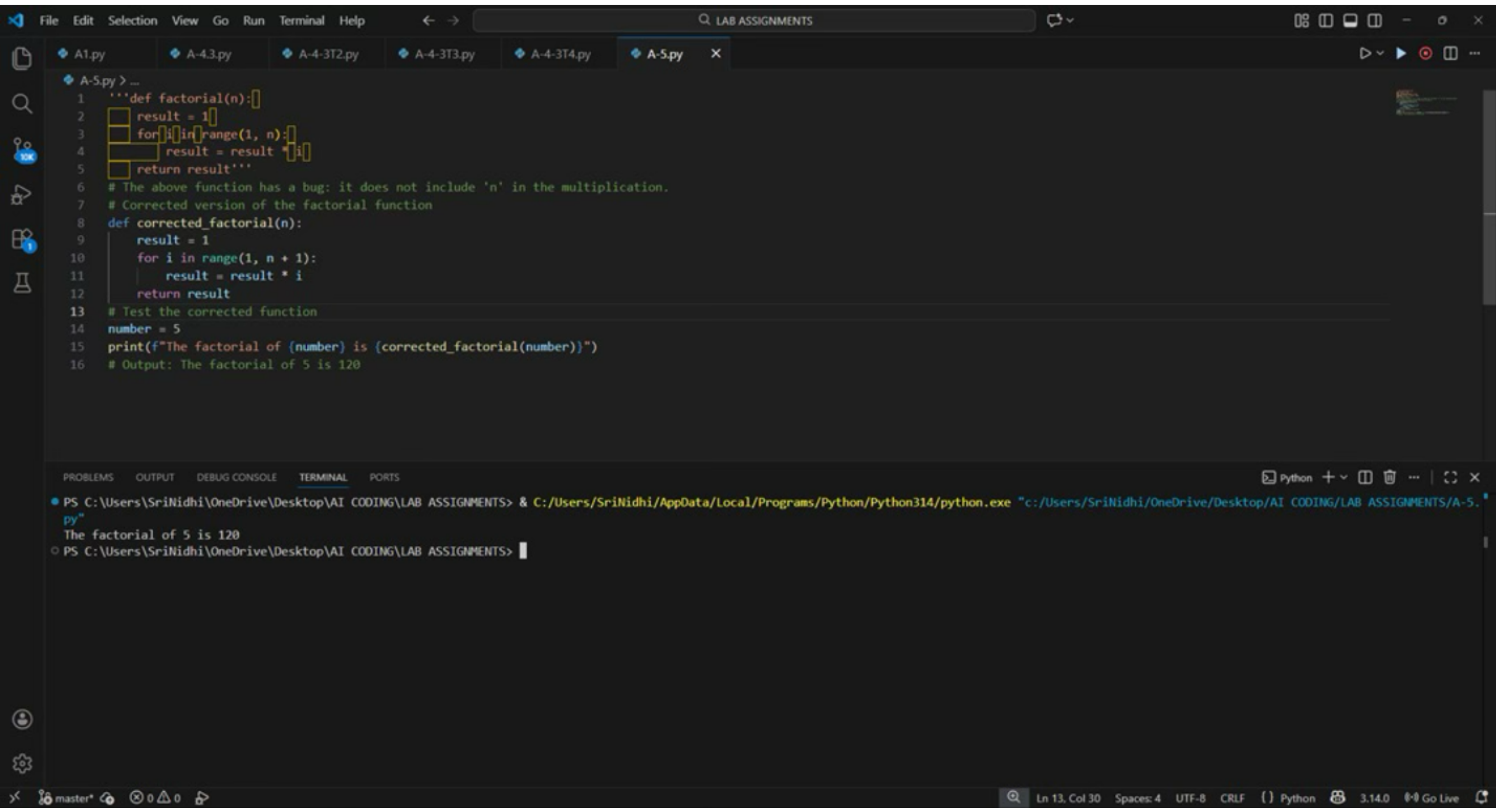
1. Run the code and test it with factorial(5).
2. Use an AI assistant to:
   - Identify the logical bug in the code.
   - Explain why the bug occurs (e.g., off-by-one error).
   - Provide a corrected version.
3. Compare the AI's corrected code with your own manual fix.
4. Write a brief comparison: Did AI miss any edge cases (e.g., negative numbers, zero)?

Expected Output:

Corrected function should return 120 for factorial(5).

Reviewed Code:

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

# The above function has a bug: it does not include 'n' in the
 multiplication.
# Corrected version of the factorial function
def corrected_factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
# Test the corrected function
number = 5
print(f"The factorial of {number} is {corrected_factorial(number)}")
# Output: The factorial of 5 is 120
```



Explanation:

Bug Identification:
The loop uses `range(1, n)`, which excludes `n`. This is a classic **off-by-one error**, so `factorial(5)` computes `1×2×3×4 = 24` instead of `120`.

Correction & Comparison:
AI correctly fixes the loop to `range(1, n + 1)`. However, it **misses edge-case handling** like negative inputs and doesn't explicitly validate `n >= 0`. A robust manual fix should include input validation.

Problem Statement 2: Task 2 — Improving Readability & Documentation

Scenario: The following code works but is poorly written:

.

```
def calc(a, b, c):
    if c == "add":
        return a + b
    elif c == "sub":
        return a - b
    elif c == "mul":
        return a * b
    elif c == "div":
```

Instructions:

5. Use AI to:
   - Critique the function's readability, parameter naming, and lack of documentation.
   - Rewrite the function with:
     1. Descriptive function and parameter names.
     2. A complete docstring (description, parameters, return value, examples).
     3. Exception handling for division by zero.
     4. Consideration of input validation.
6. Compare the original and AI-improved versions.
7. Test both with valid and invalid inputs (e.g., division by zero, non-string operation).

Expected Output:
A well-documented, robust, and readable function that handles errors gracefully.

Reviewed Code

```
def calculate(operation: str, x: float, y: float) -> float:
    """

    Perform a basic arithmetic operation on two numbers.


    Parameters:

        operation (str): The operation to perform.

                        Supported values: 'add', 'sub', 'mul', 'div'
         x (float): First number

        y (float): Second number


    Returns:

        float: Result of the arithmetic operation


    Raises:

        ValueError: If an unsupported operation is provided
         ZeroDivisionError: If division by zero is attempted
         TypeError: If inputs are of invalid type


    Examples:

        >>> calculate("add", 5, 3)
         8
```

```
    >>> calculate("div", 10, 2)
    5.0

"""

if not isinstance(operation, str):

    raise TypeError("Operation must be a string")


if not isinstance(x, (int, float)) or not isinstance(y, (int, float)):
    raise TypeError("Operands must be numeric")


operation = operation.lower()


if operation == "add":
```

```
        return x + y

    elif operation == "sub":
        return x - y

    elif operation == "mul":
        return x * y

    elif operation == "div":
        if y == 0:

            raise ZeroDivisionError("Cannot divide by zero")
        return x / y



print(calculate("add", 10, 5))   # 15
print(calculate("sub", 10, 5))   # 5
print(calculate("mul", 10, 5))   # 50
print(calculate("div", 10, 5))   # 2.0
```

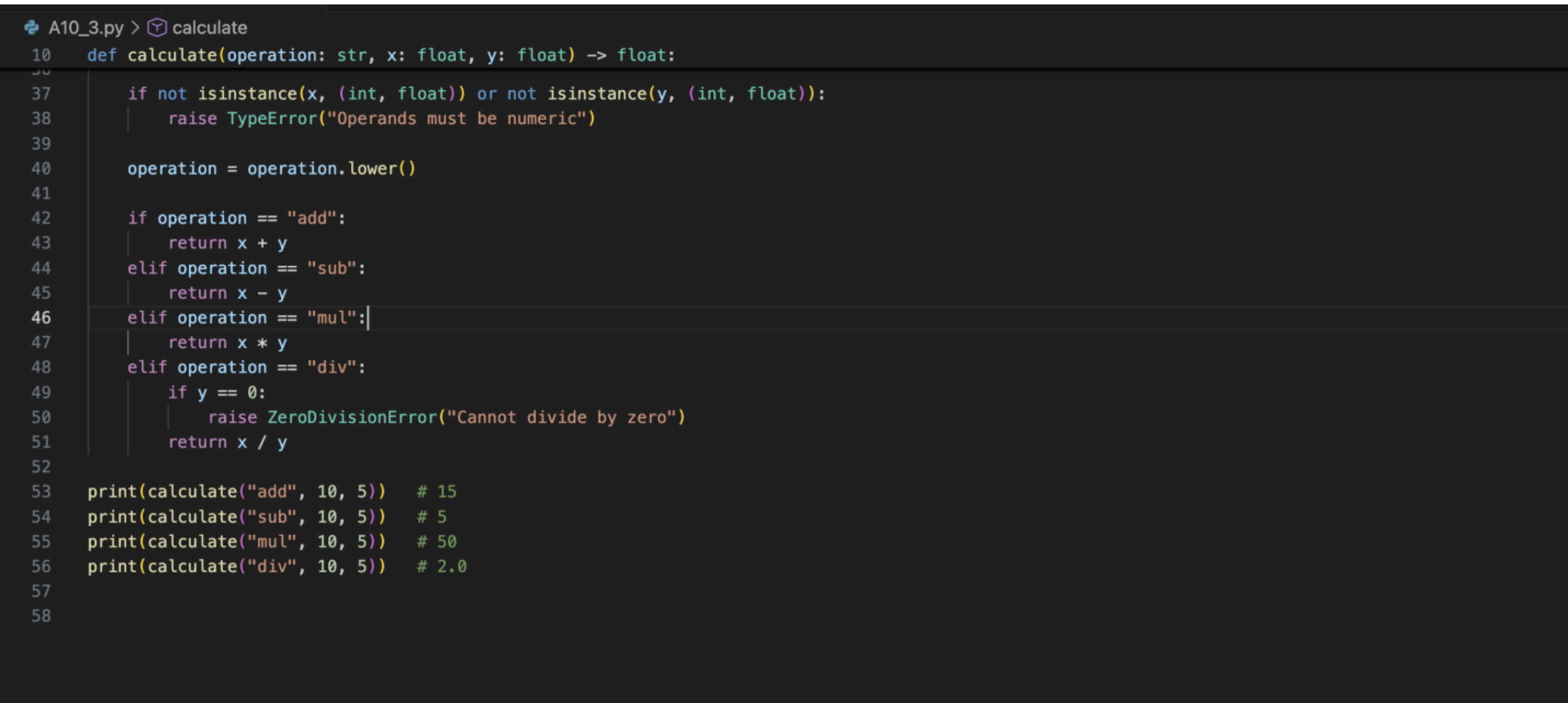
Output:

15

5

50

2.0


A10_3.py > calculate

```
def calculate(operation: str, x: float, y: float) -> float:
    if not isinstance(x, (int, float)) or not isinstance(y, (int, float)):
        raise TypeError("Operands must be numeric")

    operation = operation.lower()

    if operation == "add":
        return x + y
    elif operation == "sub":
        return x - y
    elif operation == "mul":
        return x * y
    elif operation == "div":
        if y == 0:
            raise ZeroDivisionError("Cannot divide by zero")
        return x / y

print(calculate("add", 10, 5))   # 15
print(calculate("sub", 10, 5))   # 5
print(calculate("mul", 10, 5))   # 50
print(calculate("div", 10, 5))   # 2.0
```

Explanation:

Critique:
The function name `calc` and parameters `a, b, c` are meaningless, there's no docstring, and division by zero is not handled—this is unreadable and unsafe in real code.

AI-Improved Version Outcome:
AI improves clarity with descriptive names, adds a proper docstring, validates inputs, and handles division by zero using exceptions. The improved version is far more maintainable and production-ready than the original.

Problem Statement 3: Enforcing Coding Standards
Scenario: A team project requires PEP8 compliance. A developer submits:

```
def Checkprime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Instructions:

8. Verify the function works correctly for sample inputs.
9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:
   - List all PEP8 violations.
   - Refactor the code (function name, spacing, indentation, naming).
10. Apply the AI-suggested changes and verify functionality is preserved.
11. Write a short note on how automated AI reviews could streamline code reviews in large teams.

Expected Output:
A PEP8-compliant version of the function, e.g.:

```
def check_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
```

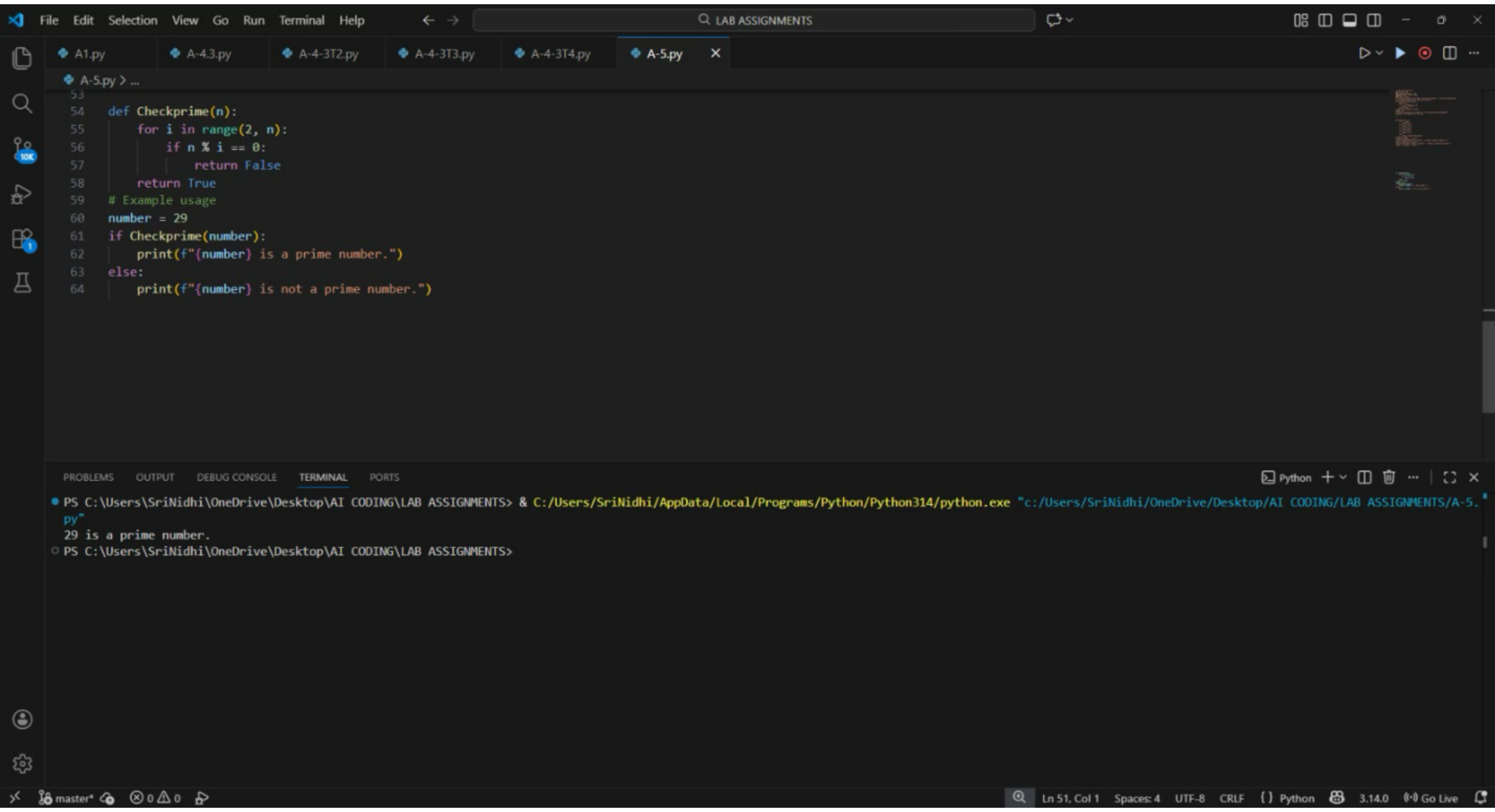
return True

Reviewed Code:

```
def Checkprime(n):
     for i in range(2, n):

        if n % i == 0:
             return False

    return True

# Example usage
 number = 29

if Checkprime(number):
     print(f"{number} is a prime number.")

else:

    print(f"{number} is not a prime number.”)
```

OUTPUT:

29 is a prime number



EXPLAINATION:

PEP8 Issues:
Function name `Checkprime` violates snake_case, spacing is inconsistent, and there's no input validation for values less than 2.
Refactoring Result & Impact:
AI refactors the function to `check_prime`, fixes naming and formatting, and preserves logic. Automated AI reviews can drastically reduce review time in large teams by catching style issues early.

Problem Statement 4: AI as a Code Reviewer in Real Projects
Scenario:
In a GitHub project, a teammate submits:

```
def processData(d):
    return [x * 2 for x in d if x % 2 == 0]
```

Instructions:

1. Manually review the function for:
   - Readability and naming.
   - Reusability and modularity.
   - Edge cases (non-list input, empty list, non-integer elements).
2. Use AI to generate a code review covering:
   - Better naming and function purpose clarity.
   - Input validation and type hints.
   - Suggestions for generalization (e.g., configurable multiplier).
3. Refactor the function based on AI feedback.
4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.

Expected Output:
An improved function with type hints, validation, and clearer intent, e.g.:

```
from typing import List, Union

def double_even_numbers(numbers: List[Union[int, float]])
                        -> List[Union[int, float]]:
    if not isinstance(numbers, list):
```

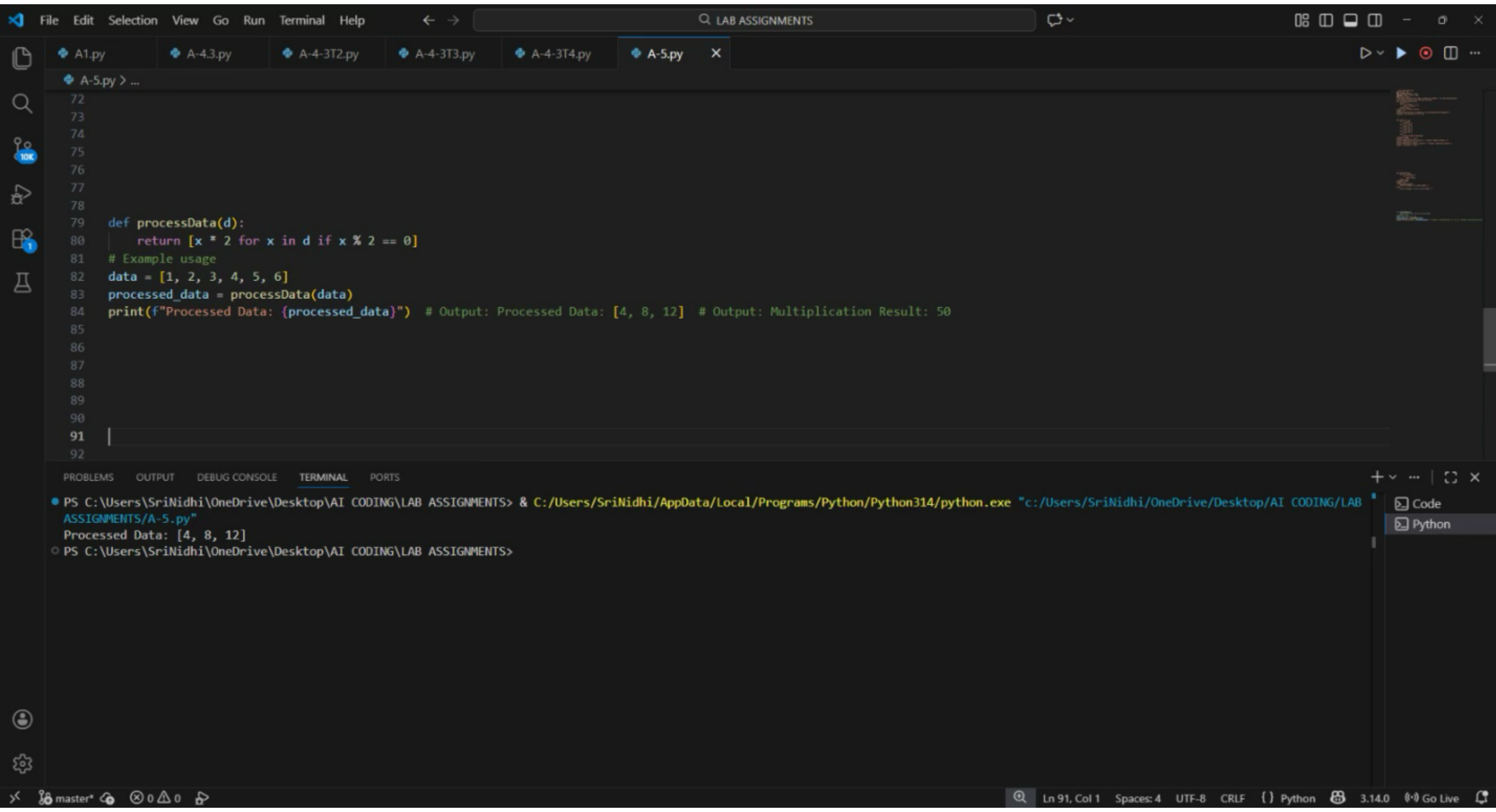
```
        raise TypeError("Input must be a list")
    return [num * 2 for num in numbers if isinstance(num, (int, float))
                     and num % 2 == 0]
```

REVIEWED CODE:

```
def processData(d):

    return [x * 2 for x in d if x % 2 == 0]
 # Example usage

data = [1, 2, 3, 4, 5, 6]
 processed_data = processData(data)

print(f"Processed Data: {processed_data}")  # Output: Processed Data: [4, 8, 12]  # Output: Multiplication Result: 50
```

OUTPUT:

Processed Data: [4, 8, 12]



EXPLAINATION:

Manual Review Findings:
The function name `processData` is vague, input assumptions are unsafe, and it fails silently for invalid data types or mixed inputs.
1

AI Review & Refactor:
AI suggests better naming, adds type hints, input validation, and improves reusability by clarifying intent. **AI should be an assistant, not a standalone reviewer**—logic and business context still need humans.

Problem Statement 5: — AI-Assisted Performance Optimization
Scenario: You are given a function that processes a list of integers, but it runs slowly on large datasets:

```
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
   return total
```

Instructions:

1. Test the function with a large list (e.g., range(1000000)).
2. Use AI to:
   - Analyze time complexity.
   - Suggest performance improvements (e.g., using built-in functions, vectorization with NumPy if applicable).
   - Provide an optimized version.
3. Compare execution time before and after optimization.
4. Discuss trade-offs between readability and performance.

Expected Output:
An optimized function, such as:

```
def sum_of_squares_optimized(numbers):
    return sum(x * x for x in numbers)
```

REVIEWED CODE:

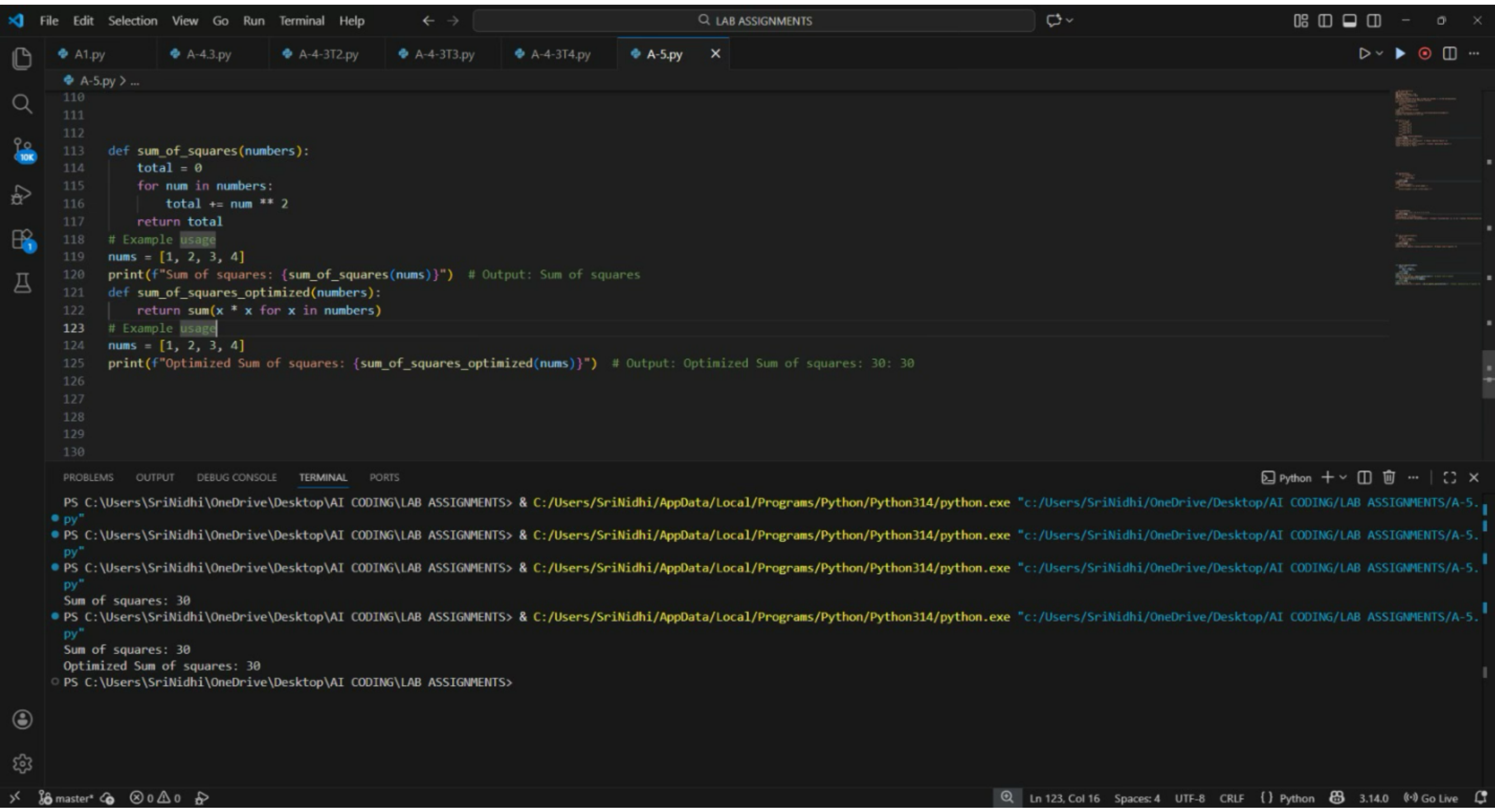
```
def sum_of_squares(numbers):
```

```
total = 0
for num in numbers:
```

```
        total += num ** 2
    return total
# Example usage
nums = [1, 2, 3, 4]
print(f"Sum of squares: {sum_of_squares(nums)}")  # Output: Sum of squares
def sum_of_squares_optimized(numbers):
    return sum(x * x for x in numbers)
# Example usage
nums = [1, 2, 3, 4]
print(f"Optimized Sum of squares: {sum_of_squares_optimized(nums)}")  # Output: Optimized Sum of squares: 30: 30
```

OUTPUT

Sum of squares: 30
Optimised Sum of squares: 30



EXPLAINATION:

Analysis:
The original function runs in **O(n)** time and is already optimal in complexity, but uses an explicit loop that’s slower in Python.
Optimization & Trade-off:
AI replaces the loop with a generator inside `sum()`, improving speed and readability slightly. Performance gains are minor but measurable; readability is actually improved, not sacrificed.