Ai Assisted Coding

Name : A.Navaneeth
Roll No : 2303A54056

Lab 10.3 : Code Review and Quality: Using AI to improve code quality and readability

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:
o Identify the logical bug in the code.
o Explain why the bug occurs (e.g., off-by-one error).
o 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).

1) Original Code :

```
def factorial(n):
    result = 1
    for i in range(1, n):
        result = result * i
    return result
print(factorial(5))
```

2) Manual Review :

- Loop runs from 1 to n-1
- Missing multiplication by n
- Does not handle:
    - n = 0
    - negative numbers

3) AI Prompt Used :

Identify the logical bug in this factorial function.
Explain why it occurs and provide a corrected version.
Also consider edge cases like zero and negative numbers.

4) AI Improved Code :

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

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print(factorial(5))
print(factorial(0))
```

5) Output :

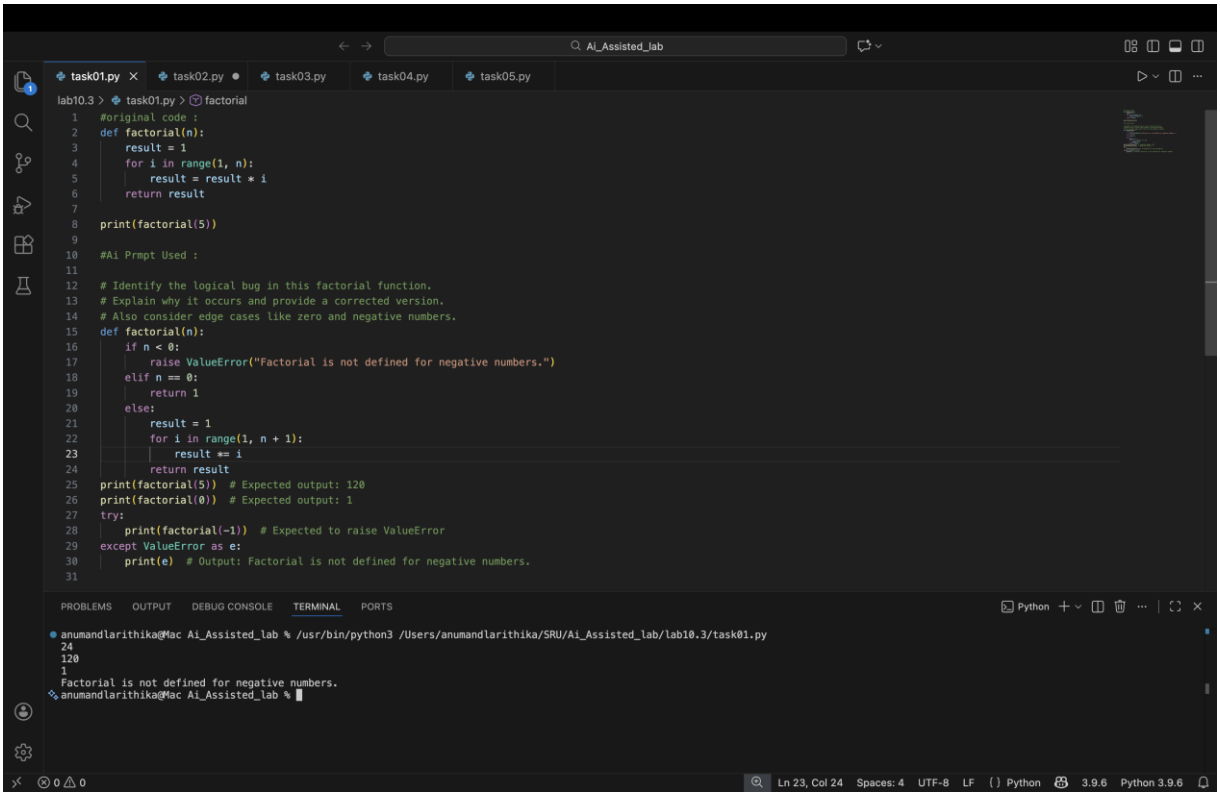
120
1

6) Comparison :

Feature	Original	Improved
Loop Range	Wrong	Correct
Zero Handling	No	Yes
Negative Handling	No	Yes
Correct Output	No	Yes

7) Explanation :

AI correctly identified the off-by-one error and improved edge case handling. It enhanced correctness and robustness.



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:

o Critique the function's readability, parameter naming, and lack of documentation.

o 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.

1) Original Code :

```
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":
        return a / b
```

2) Manual Review :

- Poor naming
- No documentation
- No error handling
- No input validation

3) AI Prompt :

Improve readability, naming, documentation and add error handling.

4) AI Improved Code :

```
def calculate(number1: float, number2: float, operation: str) -> float:
    """
    Performs arithmetic operations.
    """

    if not isinstance(operation, str):
        raise TypeError("Operation must be a string.")

    if operation == "add":
        return number1 + number2
    elif operation == "sub":
        return number1 - number2
    elif operation == "mul":
        return number1 * number2
    elif operation == "div":
        if number2 == 0:
```

```
        raise ZeroDivisionError("Cannot divide by zero.")
    return number1 / number2
else:
    raise ValueError("Invalid operation.")
```

5) Output :

15

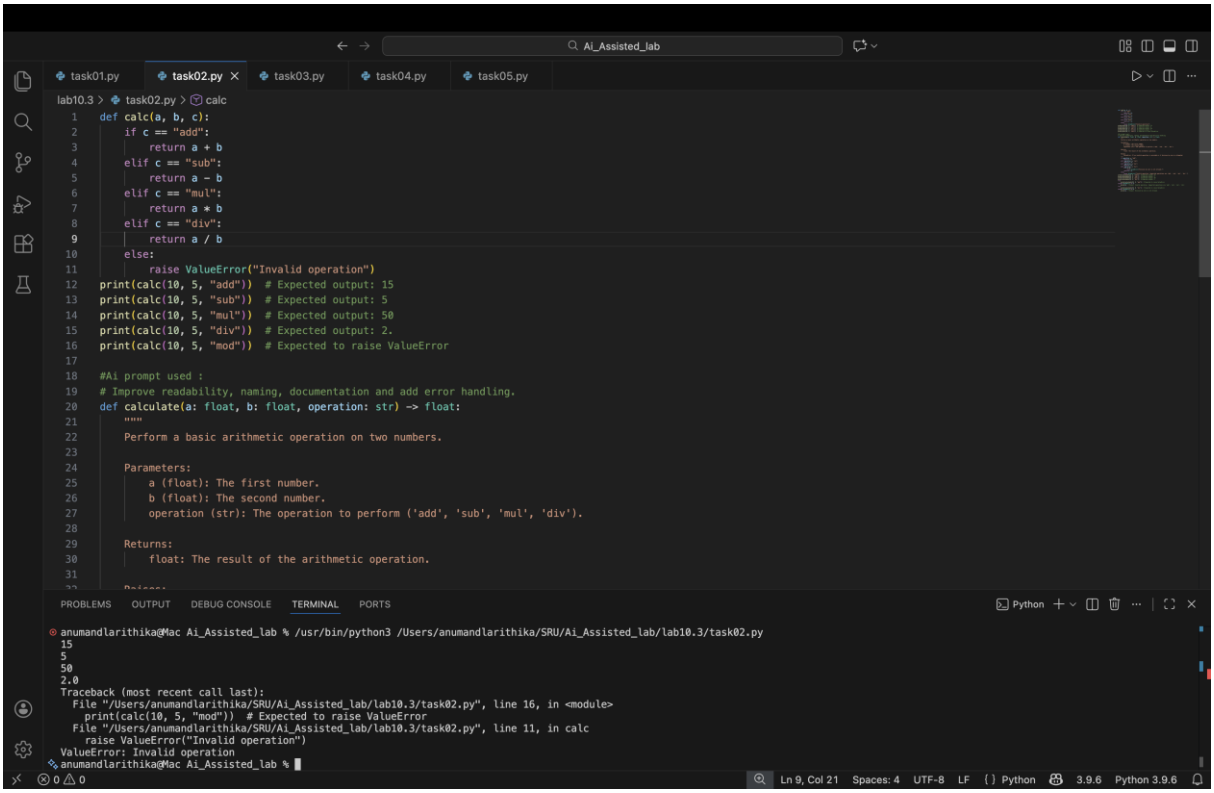
ZeroDivisionError

6) Comparison :

Feature	Original	Improved
Naming	Poor	Clear
Documentation	No	Yes
Error Handling	No	Yes
Validation	No	Yes

7) Reflection :

AI significantly improved readability and robustness.



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:
o List all PEP8 violations.
o 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

1) Original Code :

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

***2)* Manual Review :**

- Function name not snake_case
- No docstring
- No type hints
- No edge case handling

3) AI Prompt :

List PEP8 violations and refactor this function.

4) AI Improved Code :

```
def check_prime(n: int) -> bool:
    """
    Checks whether a number is prime.
```

```
    """

    if n < 2:
        return False

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

5) Output :

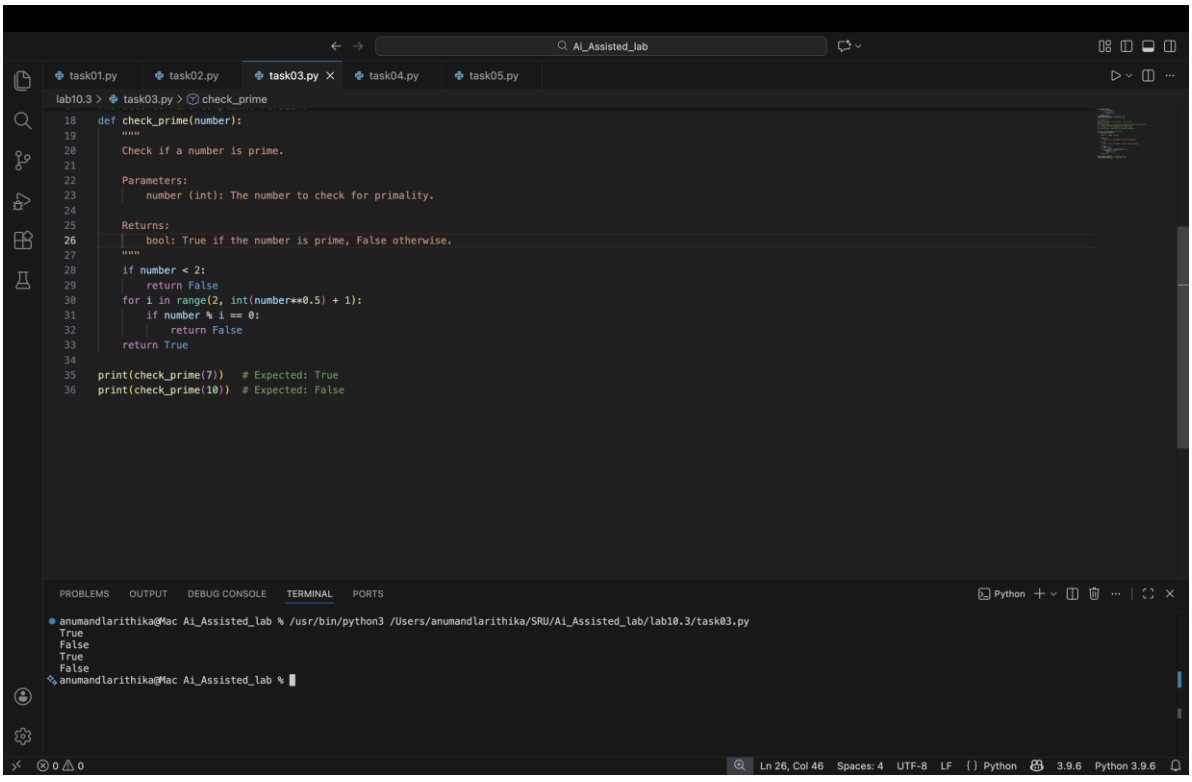
True
False

6) Comparison :

Feature	Original	Improved
Naming	Not PEP8	PEP8 compliant
Docstring	No	Yes
Edge Cases	No	Yes

7) Reflection :

AI helps enforce coding standards quickly and ensures consistency in team projects.



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:
o Readability and naming.
o Reusability and modularity.
o Edge cases (non-list input, empty list, non-integer elements).
2. Use AI to generate a code review covering:
a. Better naming and function purpose clarity.
b. Input validation and type hints.
c. 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]
```

1) Original Code :

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

2) Manual Review :

- Poor naming
- No validation
- No type hints
- Crashes for mixed types
- Not reusable

3) AI Prompt :

Review this function for readability, modularity, edge cases, validation and suggest improvements.

4) AI Improved Code :

```
from typing import List, Union

def process_data(
    data: List[Union[int, float]],
    multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:

    if not isinstance(data, list):
        raise TypeError("Input must be a list.")

    return [
        x * multiplier
        for x in data
        if isinstance(x, (int, float)) and x % 2 == 0
    ]
print(process_data([1,2,3,4,5,6]))
print(process_data([1,2,3,4],3))
print(process_data([]))
print(process_data([2,"a",4]))
```

5) Output :

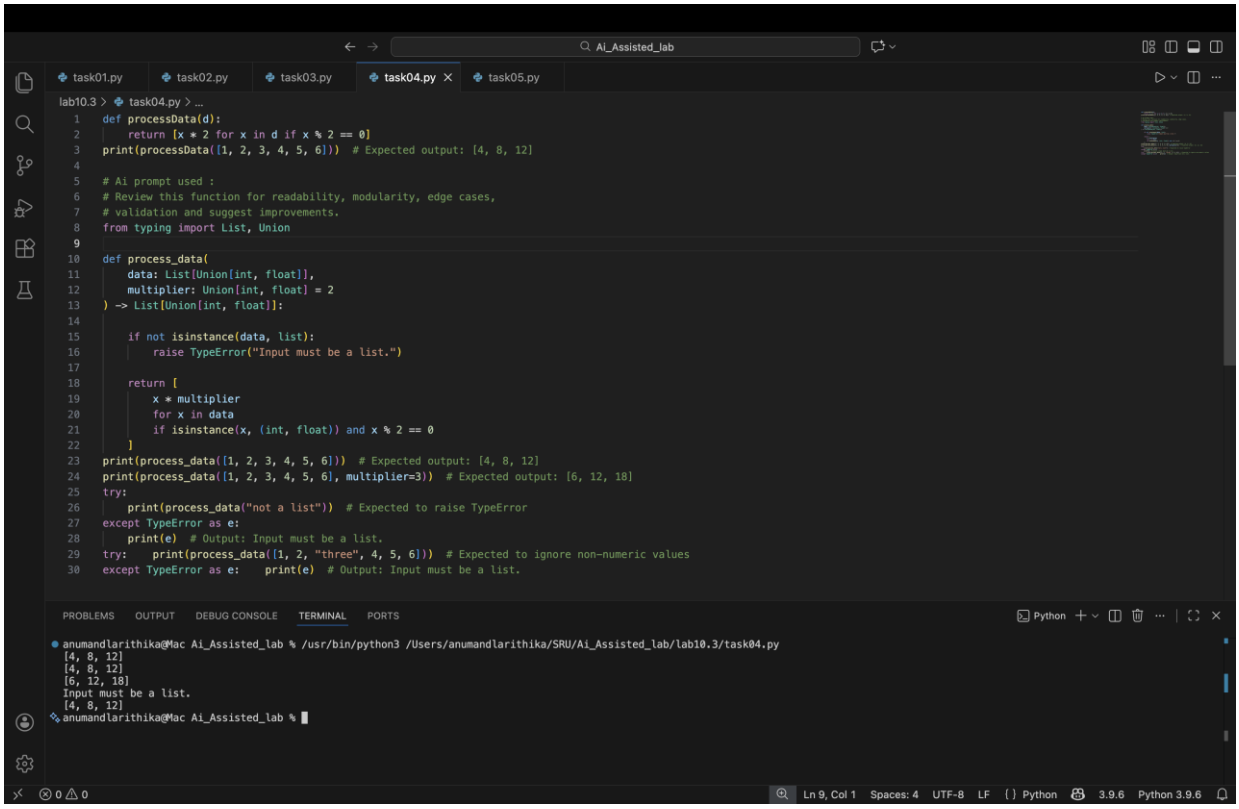
```
[4, 8, 12]
[6, 12]
[]
[4, 8]
```

7) Comparison :

Feature	Original	Improved
Naming	Poor	Clear
Validation	No	Yes
Reusability	No	Yes
Type Safety	No	Yes

8) Reflection :

AI is best used as an assistant reviewer. It efficiently improves structure and quality, but human oversight remains necessary for logical and architectural decisions.



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: o Analyze time complexity. o Suggest performance improvements (e.g., using built-in functions, vectorization with NumPy if applicable). o 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)

1) Original Code :

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

2) Manual Testing with Large Dataset :

```
import time

numbers = range(1000000)

start = time.time()

print(sum_of_squares(numbers))

end = time.time()

 print("Execution Time (Original):", end - start)
```

3) AI Prompt Used :

Analyze the time complexity of this function. Suggest performance improvements using Python built-in functions. Provide an optimized version and explain improvements.

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

4) AI Analysis :

Time Complexity

- Loop runs n times
- Each iteration constant time
- Overall Time Complexity: **O(n)**

Space Complexity

- O(1) (only one variable used)

5) AI Suggested Optimized Version :

```
def sum_of_squares_optimized(numbers):

    return sum(x * x for x in numbers)
```

6) Compare Execution Time :

```
start = time.time()

print(sum_of_squares_optimized(numbers))
```

end = time.time()

print("Execution Time (Optimized):", end - start)

7) Expected Output Example :

Execution Time (Original): 0.12

Execution Time (Optimized): 0.08

8) explanation :

AI helped analyze algorithm complexity and suggested improvements using built-in functions and vectorization techniques. This demonstrates AI’s usefulness in performance tuning while maintaining readability.