

# AI-ASSISSTANT-CODING-LAB-10.3

**Name:** V.Shiva krishna

**Batch:**41

**Roll-No:**2303A52151

## **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)?

**PROMPT:** Identify the logical bug in the factorial function. explain why it occurs, and provide a corrected version.

### **Code:**

```
# TASK 1 PROMPT (Zero-Shot)  
# Identify the logical bug in the factorial function,  
# explain why it occurs, and provide a corrected version.  
  
def factorial(n: int) -> int:  
    """  
    Calculates factorial of a number.  
    """  
    if n < 0:  
        raise ValueError("Factorial is not defined for negative numbers")  
  
    result = 1  
    for i in range(1, n + 1):    # FIX: included n (off-by-one error fixed)  
        result *= i  
    return result
```

## OUTPUT:

```
TASK 1 OUTPUT:
Bug Identified    : Off-by-one error (range(1, n) excluded n)
Fix Applied      : Changed range(1, n) to range(1, n+1)
Correct Output   : factorial(5) = 120
-----
```

### Justification:

The factorial code had an off-by-one error because the loop did not include the number  $n$ . AI helped identify the logical mistake quickly. The loop range was corrected to include  $n$ . This produced the correct output for factorial calculation.

## 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).

**PROMPT:** Critique the function for readability and documentation. then rewrite it with meaningful names, docstring, input validation, and exception handling.

## CODE:

```

# TASK 2 PROMPT (One-Shot)
# Critique the function for readability and documentation,
# then rewrite it with meaningful names, docstring,
# input validation, and exception handling.

def calculate(a: float, b: float, operation: str) -> float:
    """
    Performs arithmetic operations.
    """
    if not isinstance(operation, str):
        raise TypeError("Operation must be a string")

    if operation == "add":
        return a + b
    elif operation == "sub":
        return a - b
    elif operation == "mul":
        return a * b
    elif operation == "div":
        if b == 0:
            raise ZeroDivisionError("Division by zero not allowed")
        return a / b
    else:
        raise ValueError("Invalid operation")

```

## OUTPUT:

```

-----
TASK 2 OUTPUT:
Issues Identified : Poor naming, no documentation, no error handling
Fix Applied      : Added descriptive names, docstring, validation
Add Result       : 15
Multiply Result  : 50
-----

```

## Justification:

The original function had unclear variable names and no documentation. AI suggested better naming and added a proper docstring. Error handling and input validation were included. This made the function more readable and reliable.

## 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.

**PROMPT:**

Identify PEP8 violations and refactor the code. while preserving functionality.

**CODE:**

```
# TASK 3 PROMPT (Zero-Shot)
# Identify PEP8 violations and refactor the code
# while preserving functionality.

def check_prime(n: int) -> bool:
    """
    Checks whether a number is prime.
    """
    if n <= 1:
        return False

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

**OUTPUT:**

```
TASK 3 OUTPUT:
PEP8 Issues      : Function name, indentation, spacing
Fix Applied      : snake_case name, proper indentation
Is 7 Prime?      : True
Is 10 Prime?     : False
-----
```

**Justification:** The original code violated PEP8 naming and indentation rules. AI identified these style issues accurately. The function was refactored using snake\_case and proper formatting. Functionality was preserved with improved code quality.

#### **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.

**PROMPT:** Review the function for readability, reusability, edge cases, and type safety. Refactor accordingly.

##### **CODE:**

```
# TASK 4 PROMPT (Few-Shot)  
# Review the function for readability, reusability,  
# edge cases, and type safety. Refactor accordingly.  
  
from typing import List, Union  
  
def double_even_numbers(  
    numbers: List[Union[int, float]],  
    multiplier: int = 2  
) -> List[Union[int, float]]:  
    """  
    Doubles even numbers in a list.  
    """  
    if not isinstance(numbers, list):  
        raise TypeError("Input must be a list")  
  
    return [  
        num * multiplier  
        for num in numbers  
        if isinstance(num, (int, float)) and num % 2 == 0  
    ]
```

## OUTPUT:

```
TASK 4 OUTPUT:
Issues Identified : Poor naming, no validation, no type hints
TASK 4 OUTPUT:
Issues Identified : Poor naming, no validation, no type hints
Issues Identified : Poor naming, no validation, no type hints
Fix Applied      : Clear name, type hints, validation, reusability
Processed List   : [4, 8, 12]
-----
```

## Justification:

The original function lacked clarity and input validation. AI recommended meaningful names and type hints. Validation and reusability were added. This improved robustness and real-world usability.

## 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.

**PROMPT:** Analyze the time complexity and optimize the function.using Pythonic constructs.

## CODE:

```
# TASK 5 PROMPT (Zero-Shot)
# Analyze the time complexity and optimize the function
# using Pythonic constructs.
def sum_of_squares_optimized(numbers) -> int:
    """
    Returns sum of squares using optimized generator expression.
    """
    return sum(x * x for x in numbers)
```

## OUTPUT:

```
TASK 5 OUTPUT:  
Performance Issue: Loop-based accumulation (slower)  
Fix Applied      : Used generator expression with sum()  
Optimized Result : 285  
-----
```

## Justification:

The original function used a manual loop which was slower. AI analyzed the time complexity and suggested optimization. A generator expression with `sum()` was used. This improved performance while keeping the code readable.