AI ASSISTED CODING WEEK-10.3

2303A51642

B-28

Problem Statement 1: AI-Assisted Bug Detection

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

Logic Error Code:

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

```
\Python\Python314\python.exe "c:/Users/HARSHAVARDHAN/Desktop/AI Assitant Coding/fact.py"
24
```

Corrected Code:

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

```
\Python\Python314\python.exe "c:/Users/HARSHAVARDHAN/Desktop/AI Assitant Coding/fact.py"
120
```

Error Explanation:

- Bug: The loop used range(1, n) so n was excluded; the function computed $(n-1)!$ instead of $n!$.
- Why: In Python, range(a, b) generates integers a..b-1 (the stop is exclusive), so range(1, n) multiplies 1 through n-1.
- Fix applied: Use for i in range(1, n + 1): so the product includes n.
- Effect: In fact.py, factorial(5) now returns 120 (correct) instead of 24.

Problem Statement 2: Task 2 — Improving Readability & Documentation

Scenario: The following code works but is poorly written:

Logic Error 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":

print(calc(1, 2, "add"))
print(calc(5, 3, "sub"))
print(calc(4, 6, "mul"))
print(calc(10, 2, "div"))
```

```
  File "c:\Users\HARSHAVARDHAN\Desktop\AI Assitant Coding\fact.py", line 10
    print(calc(1, 2, "add"))
IndentationError: expected an indented block after 'elif' statement on line 8
```

Corrected 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

print(calc(1, 2, "add"))
print(calc(5, 3, "sub"))
print(calc(4, 6, "mul"))
print(calc(10, 2, "div"))
```

```
C:\Users\HARSHAVARDHAN\Desktop\AI Assitant Coding>C:\Users\HARSHAVARDHAN\AppData\Local\Programs
\Python\Python314\python.exe "c:/Users/HARSHAVARDHAN/Desktop/AI Assitant Coding/fact.py"
3
2
24
5.0
```

Error Explanation:

The error in the code is in the elif c == "div": block (line 8). **It's incomplete** — there's no return statement for the division operation.

When the "div" case is executed, the function doesn't return anything, so it implicitly returns None. This causes the last print statement: print(calc(10, 2, "div")) to output None instead of the division result.

Fix: Add the division operation to the div case: elif c == "div": return a / b

This will complete the function so all four operations (add, subtract, multiply, divide) work correctly.

Problem Statement 3: Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer submits:

Logic Error Code:

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

# AI-generated test cases
assert Checkprime(2) == True  # 2 is a prime number
assert Checkprime(1) == True  # 1 is not a prime number
print("All test cases passed!")
```

```
C:\Users\HARSHAVARDHAN\Desktop\AI Assitant Coding>C:\Users\HARSHAVARDHAN\AppData\Local\Programs
\Python\Python314\python.exe "c:/Users/HARSHAVARDHAN/Desktop/AI Assitant Coding/fact.py"
All test cases passed!
```

Corrected Code:

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


# AI-generated test cases
assert check_prime(2) == True  # 2 is a prime number
assert check_prime(1) == False  # 1 is not a prime number
print("All test cases passed!")
```

```
C:\Users\HARSHAVARDHAN\Desktop\AI Assitant Coding>C:\Users\HARSHAVARDHAN\AppData\Local\Programs
\Python\Python314\python.exe "c:/Users/HARSHAVARDHAN/Desktop/AI Assitant Coding/fact.py"
All test cases passed!
```

Error Explanation:

PEP8 issues in the submitted code:

1. Function name uses PascalCase instead of snake_case.

2. Missing indentation inside the function body.
3. Missing spaces around operators (%).
4. No blank line before function definition (in a larger file).
5. Function name does not describe behavior clearly per PEP8 conventions.
6. No handling of invalid inputs like numbers less than 2.

Problem Statement 4: AI as a Code Reviewer in Real Projects

Scenario: In a GitHub project, a teammate submits:

Original Code:

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

Refactored Function Based on Feedback:

```
from typing import List, Union

def multiply_even_numbers(
    numbers: List[Union[int, float]],
    multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:
    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
    ]
```

Explanation:

The function fails because it assumes the input is a list of integers without any validation. If the input is non-iterable or contains non-numeric elements, the % 2 operation raises a TypeError. The unclear function and variable names also make the intended behavior hard to understand.

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:

Given Code:

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

print(sum_of_squares([1, 2, 3, 4]))  # Output: 30
```

Optimized version:

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

print(sum_of_squares_optimized([1, 2, 3, 4, 5]))
```

Explanation: The function has a time complexity of O(n) since it processes each element once, which makes it slow for very large lists. Using Python's built-in sum with a generator expression reduces loop overhead and improves performance. Both versions do the same work, but the optimized one runs faster due to better internal implementation. The trade-off here improves performance without hurting readability.