AI - Assisted Coding-Assignment -10

HTNO:2303A52228
Batch:37

Lab 10: Code Review and Quality: Using AI to improve code quality and readability.

Problem Statement-1: AI-Assisted Bug Detection
Expected Output:
Corrected function should return 120 for factorial(5).
Prompt:
#make the code with correct indentation
#make it user defined function and fix the bugs in the code.
CODE AND OUTPUT:

doc.py > factorial

```
1   #make the code with correct indentation
2   #make it user defined function and fix the bugs in the code
3   def factorial(n):
4       result = 1
5       for i in range(1, n + 1):
6           result = result * i
7       return result
8   n=int(input("Enter a number: "))
9   print("Factorial of", n, "is", factorial(n))
10
```

PROBLEMS OUTPUT TERMINAL ··· pwsh

```
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/doc
y
Enter a number: 5
Factorial of 5 is 120
PS C:\AIAC> & C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe
:/AIAC/doc.py
Enter a number: -2
Factorial of -2 is 1
PS C:\AIAC> & C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe
:/AIAC/doc.py
Enter a number: 0
Factorial of 0 is 1
```

Explanation:
In this task, the AI assistant was used to detect and correct a logical error in a Python factorial function. The original code contained an one mistake in the loop range,which caused the function to stop multiplying before reaching the input number. As a result,

factorial(5) returned 24 instead of 120. The AI correctly identified that the loop should run up to n + 1 so that the final number is included in the multiplication. After correction, the function produced the expected output.

Problem Statement 2: Improving Readability & Documentation
Expected Output:
A well-documented, robust, and readable function that handles errors gracefully.
Prompt:
#correct the code with proper indentation and syntax
#Rewrite the function with:
#Descriptive function and parameter names.
#Add error handling for invalid operations and non-numeric inputs.def perform_calculation(num1, num2, operator):
#generate a docstring.
CODE AND OUTPUT:

doc.py > perform_calculation

```python
#correct the code with proper indentation and syntax
#Rewrite the function with:
#Descriptive function and parameter names.
#Add error handling for invalid operations and non-numeric inp
#generate a docstring
def perform_calculation(num1, num2, operator):
    """Perform a mathematical operation on two numbers.
    Parameters:
    num1 (float): The first number,num2 (float): The second nu
    operator (str): The operation to perform. Valid options ar
    Returns:
    float: The result of the calculation, or an error message
    """
    try:
        num1 = float(num1)
        num2 = float(num2)
    except ValueError:
        return "Error: Both inputs must be numeric."
    if operator == "add":
        return num1 + num2
    elif operator == "sub":
        return num1 - num2
    elif operator == "mul":
        return num1 * num2
    elif operator == "div":
        if num2 == 0:
            return "Error: Division by zero is not allowed."
        return num1 / num2
    else:
        return "Error: Invalid operation. Please choose 'add',
```

Explanation:
AI-generated test cases were written first for the string case conversion functions. The functions were then implemented to pass all tests using unittest, ensuring correct behavior and safe handling of invalid inputs, following the Test-Driven Development approach.

Problem Statement-3: Enforcing Coding Standards
Expected Output:
A PEP8-compliant version of the function.
Prompt:
#make the function name snake_case

#handle the case when n is less than 2
#generate a docstring for the function

CODE AND OUTPUT:

doc.py > ...

```python
#make the function name snake_case
#handle the case when n is less than 2
#generate a docstring for the function
def check_prime(n):
    """
    Check if a number is prime.
    Parameters:
        n (int): The number to check.
    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
print(check_prime(11))
print(check_prime(15))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ...  pwsh

PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AI
y
True
False
```

Explanation:

This program checks whether the number is prime or not.AI had changed the function name with snake_case and it handles all the edge cases.AI had generated the docstring for the given code which satisfies allthe PEP8 conditions.

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

Expected Output:

An improved function with type hints, validation, and clearer intent.

Prompt:

#make the function name snakecase ,add type hints
#validate the input to ensure it's a list of integers
#generate a docstring for the function

CODE AND OUTPUT:

doc.py > ...

```python
#make the function name snakecase ,add type hints
#validate the input to ensure it's a list of integers
#generate a docstring for the function
from typing import List
def process_even_numbers(d: List[int]) -> List[int]:
    """
    Process a list of integers and return a list of even numbe
    Parameters:
        d (List[int]): The list of integers to process.
    Returns:
        List[int]: A list of even numbers from the input list,
    """
    if not isinstance(d, list):
        raise TypeError("Input must be a list")
    for item in d:
        if not isinstance(item, int):
            raise TypeError("All items in the list must be int
    return [x * 2 for x in d if x % 2 == 0]

```

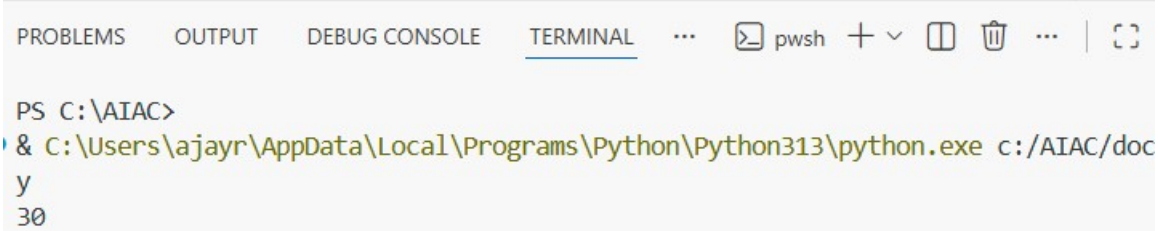
Explanation:
AI can greatly assist in code reviews by identifying naming issues, missing validations, and improvement opportunities quickly. However, AI should not replace human reviewers because humans understand project context, requirements, and design choices better.

Problem Statement-5: AI-Assisted Performance Optimization
Expected Output:
An optimized function, such as:
def sum_of_squares_optimized(numbers):
return sum(x * x for x in numbers)
Prompt:
#make the code optimized and efficient
#generate test cases for the function
#handle type errors gracefully
CODE AND OUTPUT:

doc.py > ...

```python
#make the code optimized and efficient
#generate test cases for the function
#handle type errors gracefully
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError("All elements must be numbers")
        total += num ** 2
    return total
print(sum_of_squares([1, 2, 3, 4]))

```



Explanation:
The function is designed to efficiently compute the sum of squares of elements in a list while also ensuring robustness. Before performing the calculation, the function checks whether each element is a valid numeric type (integer or float). If any non-numeric value is encountered, it raises a clear error message instead of failing silently, which improves reliability. The loop processes each element only once, maintaining linear time complexity O(n).