Assignment 10.3

2303A51247

K NagaSri Reddy

Batch – 18

Task Description #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).

Prompt :

```
# def factorial(n):
#     result = 1
```

```
#     for i in range(1, n):
#         result = result * i
#     return result
# print(factorial(5))
# Identify the logical bug in this factorial function.
# Explain why the error occurs.
# Provide a corrected version.
# Also handle edge cases like zero and negative numbers.
```

assignment10_3 > ...

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

# print(factorial(5))

# Identify the logical bug in this factorial function.
# Explain why the error occurs.
# Provide a corrected version.
# Also handle edge cases like zero and negative numbers.
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0:
        return 1

    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(5))
print(factorial(0))
# print(factorial(-3))  # This will raise an error
```

Output :

```
PS C:\Users\ANIL\Desktop\AI Assisant> & \\?\C:\Users\ANIL\AppData\Local\Python\pythoncore-3.14-64\python.exe "c:/Users/ANIL/Desktop/AI Assisant/assignment10_3"
120
1
```

Analysis

1. The original bug was an **off-by-one error** caused by incorrect use of range().
2. AI correctly identified the logical mistake.
3. AI also improved the function by adding proper exception handling.

Task Description #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.

Prompt :

##Task2:

Original poorly written

function # 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

Critique the readability of this function.

Improve the function by:

- Using descriptive function and parameter names

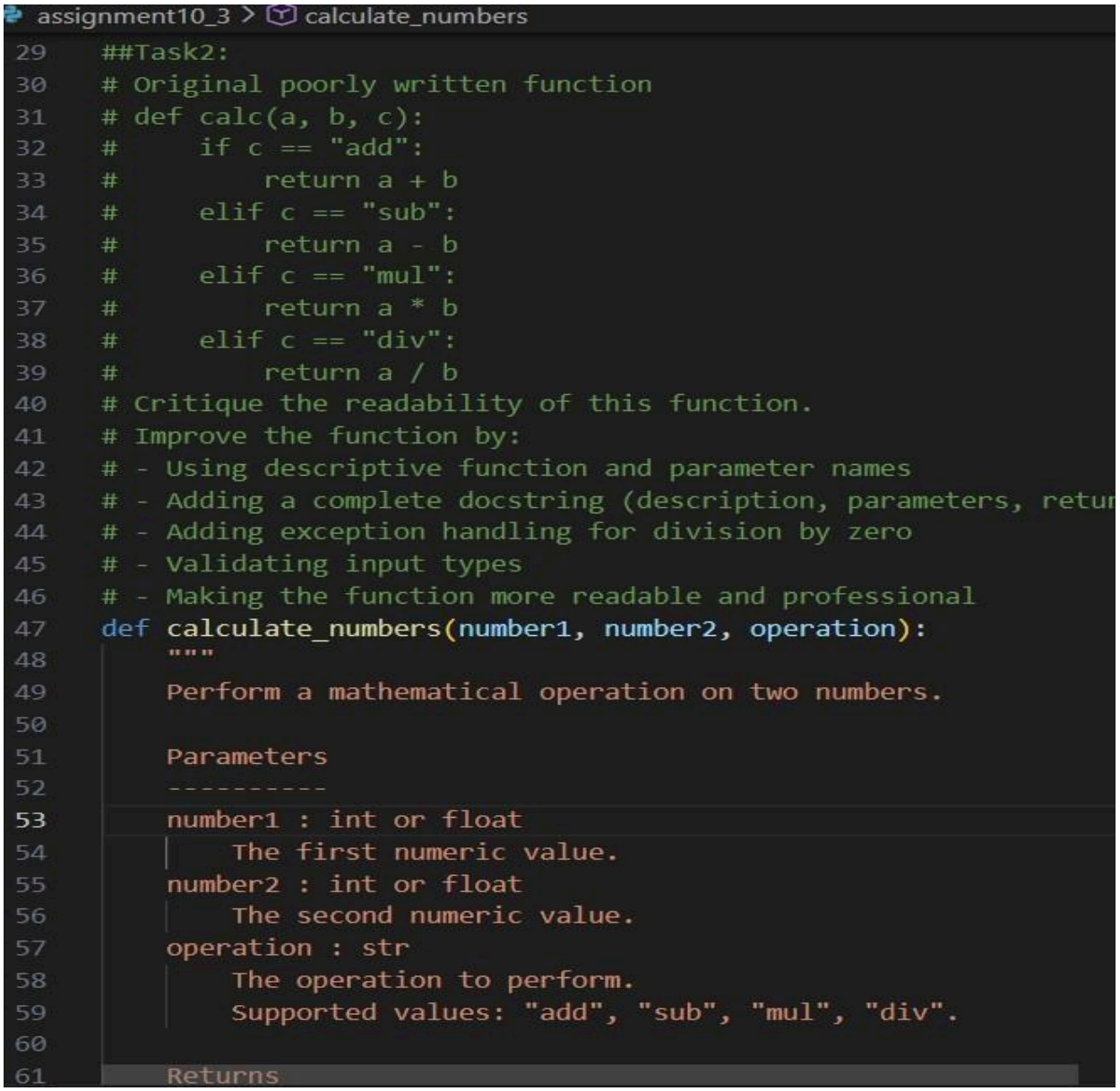
- Adding a complete docstring (description, parameters, return value, examples)

- Adding exception handling for division by zero

- Validating input types

- Making the function more readable and professional

Code:



```
assignment10_3 > calculate_numbers
29  ##Task2:
30  # Original poorly written function
31  # def calc(a, b, c):
32  #     if c == "add":
33  #         return a + b
34  #     elif c == "sub":
35  #         return a - b
36  #     elif c == "mul":
37  #         return a * b
38  #     elif c == "div":
39  #         return a / b
40  # Critique the readability of this function.
41  # Improve the function by:
42  # - Using descriptive function and parameter names
43  # - Adding a complete docstring (description, parameters, retur
44  # - Adding exception handling for division by zero
45  # - Validating input types
46  # - Making the function more readable and professional
47  def calculate_numbers(number1, number2, operation):
48      """
49      Perform a mathematical operation on two numbers.
50
51      Parameters
52      ----------
53      number1 : int or float
54          The first numeric value.
55      number2 : int or float
56          The second numeric value.
57      operation : str
58          The operation to perform.
59          Supported values: "add", "sub", "mul", "div".
60
61      Returns
```

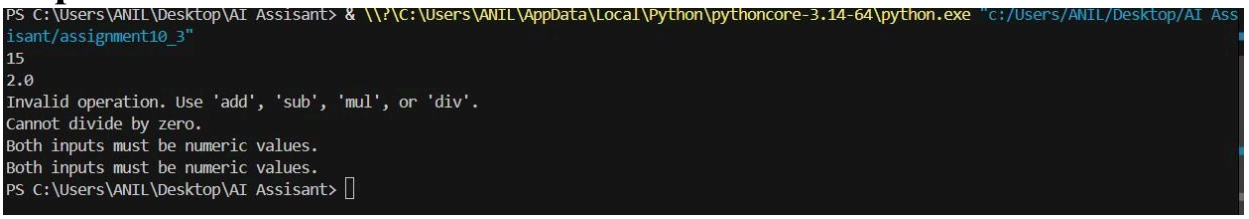
assignment10_3 > calculate_numbers

```
47    def calculate_numbers(number1, number2, operation):
61        Returns
62        -------
63        int or float
64            The result of the mathematical operation.
65
66        Raises
67        ------
68        ValueError
69            If an invalid operation is provided.
70        TypeError
71            If inputs are not numeric.
72        ZeroDivisionError
73            If division by zero is attempted.
74
75        Examples
76        --------
77        >>> calculate_numbers(10, 5, "add")
78        15
79        >>> calculate_numbers(10, 5, "div")
80        2.0
81        """
82
83        # Input validation
84        if not isinstance(number1, (int, float)) or not isinstance(number2, (int, float)):
85            raise TypeError("Both inputs must be numeric values.")
86
87        if not isinstance(operation, str):
88            raise TypeError("Operation must be a string.")
89
90        operation = operation.lower()
91
92        if operation == "add":
```

assignment10_3 > calculate_numbers

```
7     def calculate_numbers(number1, number2, operation):
2         if operation == "add":
3             return number1 + number2
4         elif operation == "sub":
5             return number1 - number2
6         elif operation == "mul":
7             return number1 * number2
8         elif operation == "div":
9             if number2 == 0:
0                 raise ZeroDivisionError("Cannot divide by zero.")
1             return number1 / number2
2         else:
3             raise ValueError("Invalid operation. Use 'add', 'sub', 'mul', or 'div'.")
4         # Testing Original Function
5     # print(calc(10, 5, "add"))
6
7     #testing both valid and invalid inputs for the improved function
8     print(calculate_numbers(10, 5, "add"))  # Valid input
9     print(calculate_numbers(10, 5, "div"))  # Valid input
0     #invalid operation
1     try:
2         print(calculate_numbers(10, 5, "mod"))  # Invalid operation
3     except ValueError as e:
4         print(e)
5     #division by zero
6     try:
7         print(calculate_numbers(10, 0, "div"))  # Division by zero
8     except ZeroDivisionError as e:
9         print(e)
0     #invalid input types
1     try:
2         print(calculate_numbers("10", 5, "add"))  # Invalid input type
```

Output:



Analysis:

1. Handling division by zero.

2.Validating input types.

3.Raising meaningful exceptions.

Task Description #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:

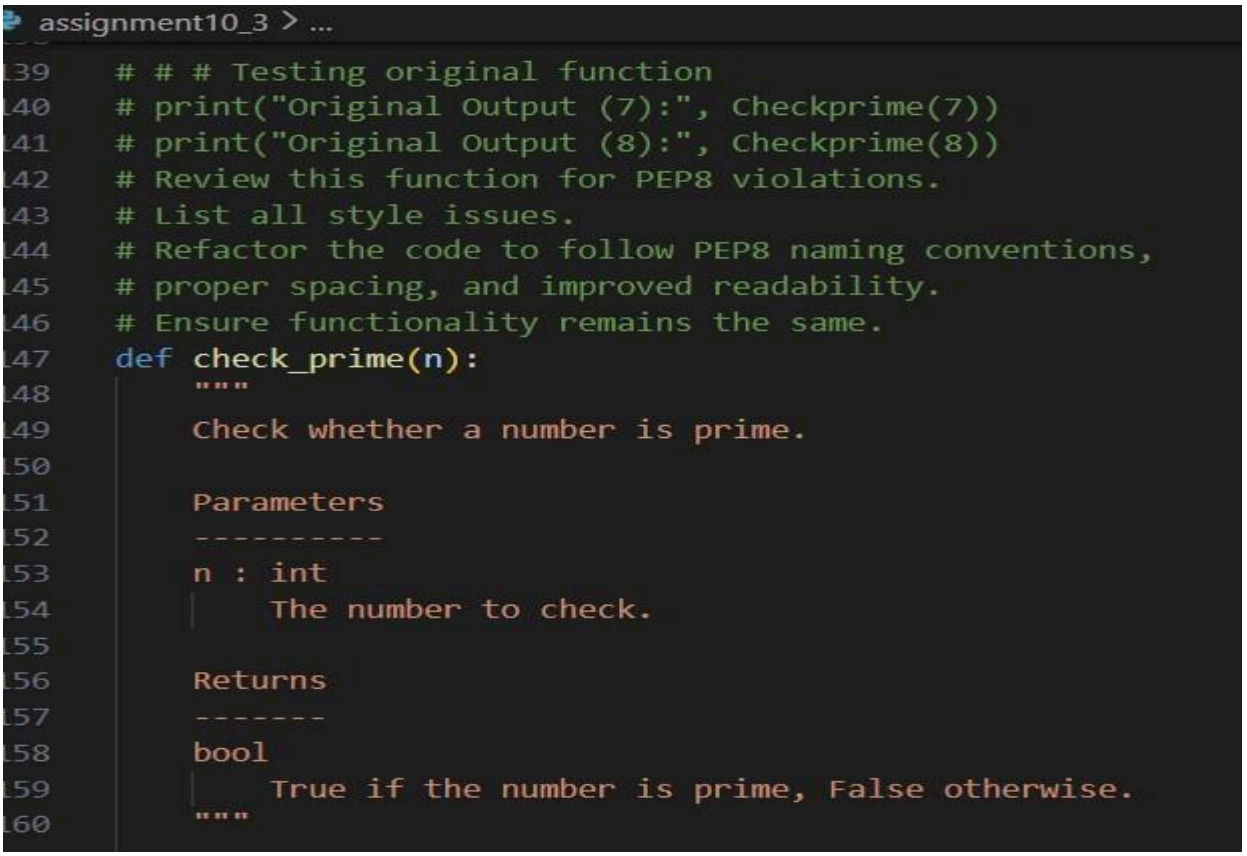
A PEP8-compliant version of the function,

e.g.: def check_prime(n): for i in range(2, n):

if n % i == 0: return

False return True

Prompt :



```
assignment10_3 > ...
147    def check_prime(n):
161
162        if not isinstance(n, int):
163            raise TypeError("Input must be an integer.")
164
165        if n <= 1:
166            return False
167
168        for i in range(2, n):
169            if n % i == 0:
170                return False
171
172        return True
173
174
175    # Testing improved function
176    print("Improved Output (7):", check_prime(7))
177    print("Improved Output (8):", check_prime(8))
```

Output:

```
PS C:\Users\ANIL\Desktop\AI Assisant> & \\?\C:\Users\ANIL\AppData\Local\Python\pythoncore-3.14-64\py
isant/assignment10_3"
Improved Output (7): True
Improved Output (8): False
PS C:\Users\ANIL\Desktop\AI Assisant>
```

Analysis:

AI also provides quick suggestions for improvement, allowing developers to focus more on logic and performance rather than style issues

Task Description #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]
```

Prompt :

##task4:

```
# def processData(d):
#     return [x * 2 for x in d if x % 2 == 0]
# Review this function for readability, naming clarity, and edge cases.
# Suggest improvements including:
# - Better function name
# - Type hints
# - Input validation
# - Configurable multiplier
# - Improved documentation
```

Code :

assignment10_3 > ...

```
    # def processData(d):
    #     return [x * 2 for x in d if x % 2 == 0]
    # Review this function for readability, naming clarity, and edge cases.
    # Suggest improvements including:
    # - Better function name
    # - Type hints
    # - Input validation
    # - Configurable multiplier
    # - Improved documentation
    from typing import List, Union
    def multiply_even_numbers(
        numbers: List[Union[int, float]],
        multiplier: Union[int, float] = 2
    ) -> List[Union[int, float]]:
        """
        Multiply even numbers in a list by a given multiplier.

        Parameters
        ----------
        numbers : List[int | float]
            A list of numeric values.
        multiplier : int | float, optional
            The value to multiply even numbers by (default is 2).
```

assignment10_3 > ...

```
    def multiply_even_numbers(
        Returns
        -------
        List[int | float]
            A list containing multiplied even numbers.

        Raises
        ------
        TypeError
            If input is not a list or contains non-numeric values.
        """

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

        result = []

        for num in numbers:
            if not isinstance(num, (int, float)):
                raise TypeError("All elements must be numeric.")

            if num % 2 == 0:
                result.append(num * multiplier)
```

```
assignment10_3 > ...
191    def multiply_even_numbers(
226                result.append(num * multiplier)
227
228        return result
229    print(multiply_even_numbers([1, 2, 3, 4]))
230    # Output: [4, 8]
231
232    print(multiply_even_numbers([2, 4, 6], 3))
233    # Output: [6, 12, 18]
234
235    print(multiply_even_numbers([]))
236    # Output: []
237
238    try:
239        print(multiply_even_numbers("123"))
240    except Exception as e:
241        print("Error:", e)
242
243    try:
244        print(multiply_even_numbers([2, "a", 4]))
245    except Exception as e:
246        print("Error:", e)
```

Output

```
PS C:\Users\ANIL\Desktop\AI Assisant> & \\?\C:\Users\ANIL\AppData\Local\Python\pythoncore-3.14-64\python.exe
isant/assignment10_3"
[4, 8]
[6, 12, 18]
[]
Error: Input must be a list.
Error: All elements must be numeric.
PS C:\Users\ANIL\Desktop\AI Assisant>
```

Analysis:

1. Quickly identified readability and naming issues.

2. Suggested better structure and documentation.

Task Description #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)

Prompt :

##task5:

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

return total

Test the function with a large list (e.g., range(1000000)).

Use AI to:

Analyze time complexity.

Suggest performance improvements (e.g., using built-in # functions, vectorization with NumPy if applicable).

Provide an optimized version.

Compare execution time before and after optimization.

4Discuss trade-offs between readability and performance.

Code :

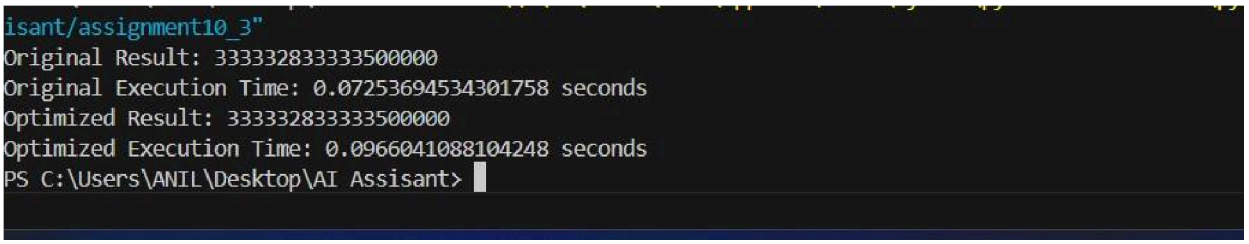
assignment10_3 > ...

```
# def sum_of_squares(numbers):
# total = 0
# for num in numbers:
# total += num ** 2
# return total
# Test the function with a large list (e.g., range(1000000)).
#  Use AI to:
#  Analyze time complexity.
# Suggest performance improvements (e.g., using built-in
# functions, vectorization with NumPy if applicable).
# Provide an optimized version.
#  Compare execution time before and after optimization.
# 4Discuss trade-offs between readability and performance.
import time
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
# Testing original function
start_time = time.time()
result = sum_of_squares(range(1000000))
end_time = time.time()
```

assignment10_3 > ...

```
print("Original Result:", result)
print("Original Execution Time:", end_time - start_time, "seconds")
# Optimized version using built-in sum and generator expression
def optimized_sum_of_squares(numbers):
    return sum(num ** 2 for num in numbers)
# Testing optimized function
start_time = time.time()
optimized_result = optimized_sum_of_squares(range(1000000))
end_time = time.time()
print("Optimized Result:", optimized_result)
print("Optimized Execution Time:", end_time - start_time, "seconds")
```

Output :



Analysis:

1.It suggested using Python's built-in sum() with a generator expression for cleaner and slightly faster execution.
2. It also recommended NumPy vectorization for better performance on very large datasets.