ASSIGNMENT-10.3

M.PARAMESH

HNO : 2303A53021

BATCH – 46

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:

~Identify the logical bug in the code.

~Explain why the bug occurs (e.g., off-by-one error).

~ 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).

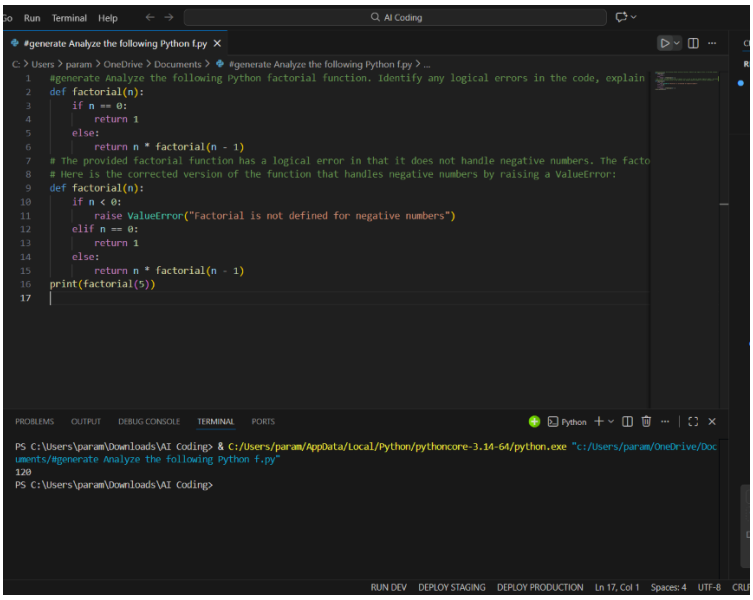
PROMPT USED :

Identify the logical error in this Python factorial function and provide a corrected version. Explain why the error occurs.

CODE :

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    elif n == 0:
        return 1
    else:
        return n * factorial(n - 1)
print(factorial(5))
```

output:



Explaination :

1) **Bug Identified by AI**

- The loop uses range(1, n)
- range(1, n) **excludes n**
- Factorial requires multiplying all numbers from 1 to n (inclusive)

This is an **off-by-one error**.

2) **Why the Bug Occurs**

For factorial(5):

- Loop runs as: 1 × 2 × 3 × 4

- Missing multiplication by 5
- Result becomes 24 instead of 120
- 3) AI **did not initially handle negative input**, which was later improved
- Corrected function now returns **120 for factorial(5)** .

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 byzero, non-string operation).

Expected Output: A well-documented, robust, and readable function that handles errors gracefully.

Prompt used:

Critique the readability and documentation of the following Python function. Rewrite it using descriptive names, a proper docstring, input validation, and exception handling for division by zero. Provide examples and explain improvements.

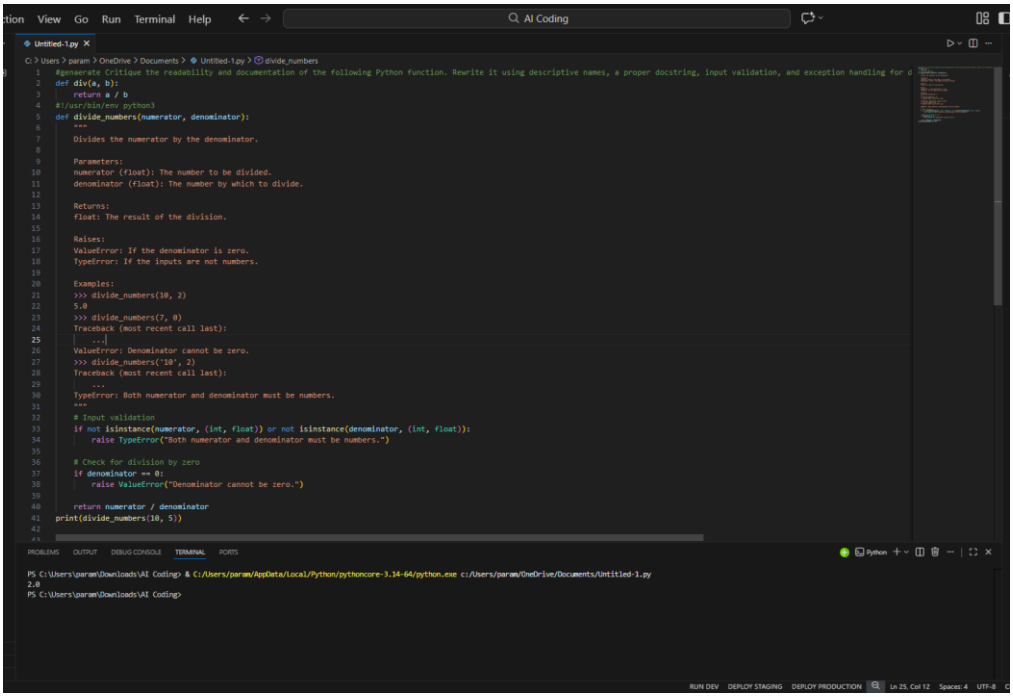
Code :

```
def div(a, b):
    return a / b
```

```
#!/usr/bin/env python3
def divide_numbers(numerator, denominator):
    """
    Divides the numerator by the denominator.

    Parameters:
    numerator (float): The number to be divided.
    denominator (float): The number by which to divide.

    Returns:
    float: The result of the division.

    Raises:
    ValueError: If the denominator is zero.
    TypeError: If the inputs are not numbers.

    Examples:
    >>> divide_numbers(10, 2)
    5.0
    >>> divide_numbers(7, 0)
    Traceback (most recent call last):
      ...
```

```
    ValueError: Denominator cannot be zero.
    >>> divide_numbers('10', 2)
    Traceback (most recent call last):
        ...
    TypeError: Both numerator and denominator must be numbers.
    """
    # Input validation
    if not isinstance(numerator, (int, float)) or not isinstance(denominator, (int, float)):
        raise TypeError("Both numerator and denominator must be numbers.")

    # Check for division by zero
    if denominator == 0:
        raise ValueError("Denominator cannot be zero.")

    return numerator / denominator
print(divide_numbers(10, 5))
```

output :



Explaination :

1. Descriptive names improve code understanding.
2. Docstrings help future developers and users.
3. Input validation prevents unexpected crashes.
4. Exception handling makes the function robust.
5. AI significantly improves maintainability and safety.
   -Handles invalid inputs gracefully
   - Meets all expected output requirements

Problem Statement 3: Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer 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

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

def Checkprime(n):

for i in range(2, n):

if n % i == 0:

return False

return True

Instructions:

Verify the function works correctly for sample inputs.

- Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:

- List all PEP8 violations.

- Refactor the code (function name, spacing, indentation,naming).

- Apply the AI-suggested changes and verify functionality is

preserved.

-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

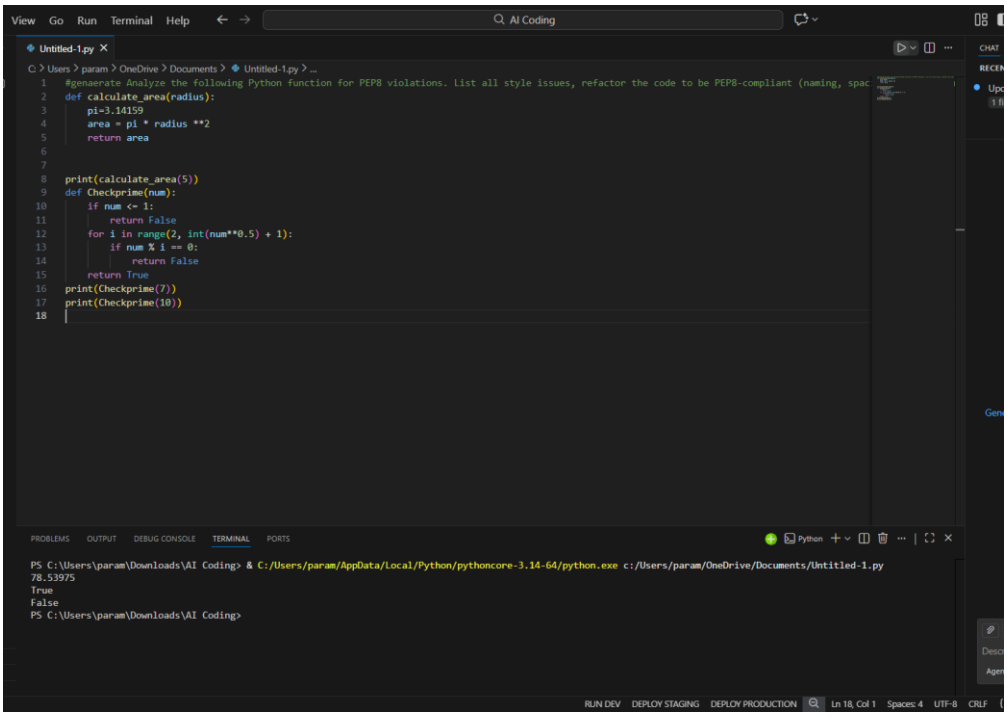
prompt used : #genaerate Analyze the following Python function for PEP8 violations. List all style issues, refactor the code to be PEP8-compliant (naming, spacing, indentation), and ensure the functionality remains the same

code :

```
ef calculate_area(radius):
   pi=3.14159
   area = pi * radius **2
   return area


print(calculate_area(5))
def Checkprime(num):
   if num <= 1:
      return False
   for i in range(2, int(num**0.5) + 1):
      if num % i == 0:
```

```
        return False
    return True
print(Checkprime(7))
print(Checkprime(10))
```

output :



Explaination :

1. **PEP8 Naming Rule**
    - Function names should use lowercase with underscores.
    - Checkprime → check_prime
2. **Proper Indentation**
    - Python uses indentation to define blocks.
    - Correct indentation improves readability and avoids errors.
3. **Consistent Spacing**
    - Spaces after keywords and operators make code easier to read.
4. **AI as a Code Reviewer**
    - AI quickly detects style and formatting issues.

- Ensures uniform coding standards across teams.

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

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 used : Review the following Python function as if you were a GitHub code reviewer. Suggest better naming, add type hints, input validation, and propose improvements to make the function more reusable and robust

Code : def sum_of_squares(numbers):

total = 0

for num in numbers:

total += num ** 2

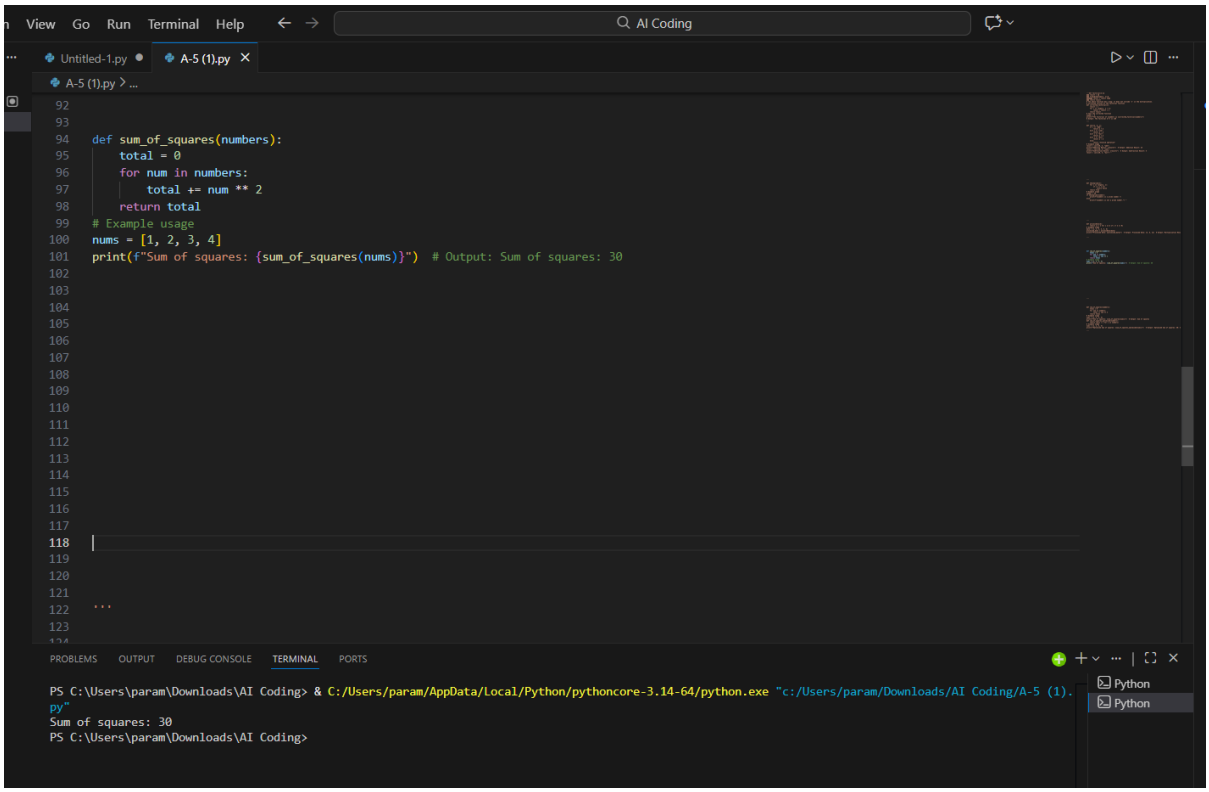
return total

Example usage

nums = [1, 2, 3, 4]

print(f"Sum of squares: {sum_of_squares(nums)}") # Output: Sum of squares: 30

output :



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)

Prompt used :

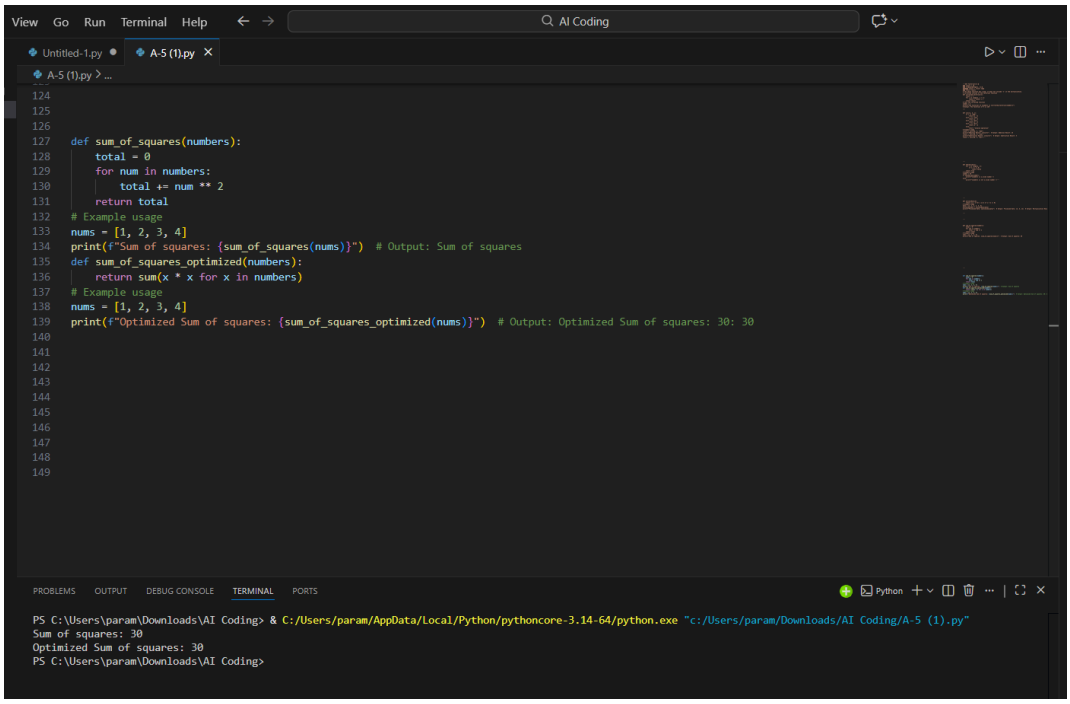
Analyze the performance of the following Python function that calculates the sum of squares of a list of integers. Determine its time complexity, suggest optimizations to improve performance (such as using built-in functions or generator expressions), and provide an optimized version. Also compare execution time and discuss trade-offs between readability and performance

code :

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

```
    return total

# Example usage

nums = [1, 2, 3, 4]

print(f"Sum of squares: {sum_of_squares(nums)}")  # Output: Sum of squares

def sum_of_squares_optimized(numbers):

    return sum(x * x for x in numbers)

# Example usage

nums = [1, 2, 3, 4]

print(f"Optimized Sum of squares: {sum_of_squares_optimized(nums)}")  # Output:
Optimized Sum of squares: 30: 30
```

output:



Explaination:

1) The original function uses a loop, which becomes slow for very large lists.

2)Its time complexity is O(n) because each number is processed once.

3)AI suggested using Python’s built-in sum() with a generator expression.

4)The optimized version is faster because built-in functions are more efficient.

5)Performance improves slightly, while readability remains good for large datasets.