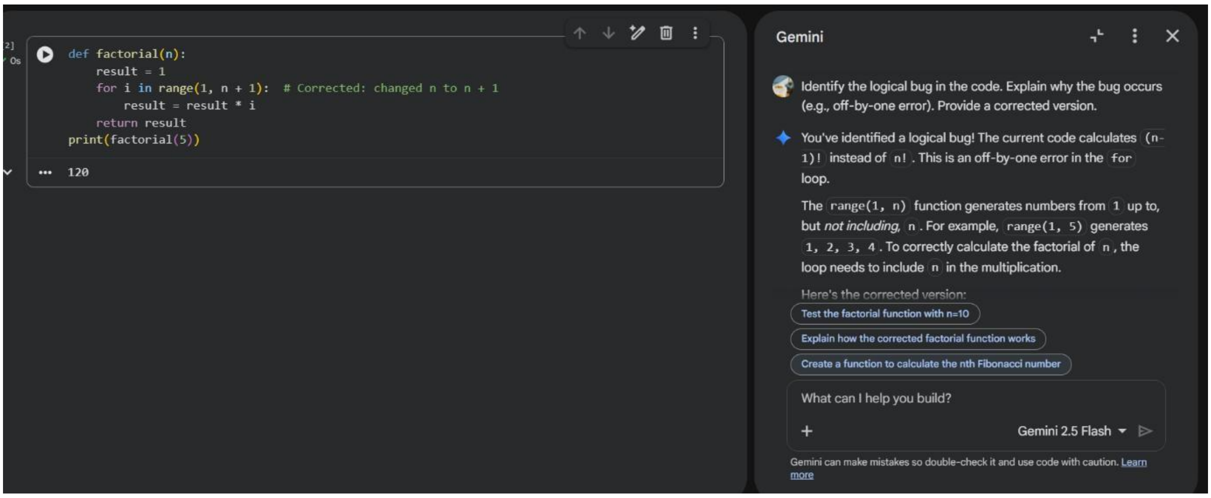
ASSIGNMENT - 10

Thumma Hasini | 2303A52076 | Batch - 37

Problem Statement 1: AI-Assisted Bug Detection

A Python function was written to calculate the factorial of a number. However, the function produces incorrect output due to a logical error. The task is to test the function, identify the bug using AI assistance, correct it, and compare the AI fix with a manual fix.



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

```
120
```

Gemini

Identify the logical bug in the code. Explain why the bug occurs (e.g., off-by-one error). Provide a corrected version.

You've identified a logical bug! The current code calculates `(n-1)!` instead of `n!`. This is an off-by-one error in the `for` loop.

The `range(1, n)` function generates numbers from `1` up to, but *not including*, `n`. For example, `range(1, 5)` generates `1, 2, 3, 4`. To correctly calculate the factorial of `n`, the loop needs to include `n` in the multiplication.

Here's the corrected version:

Test the factorial function with n=10
Explain how the corrected factorial function works
Create a function to calculate the nth Fibonacci number
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more

Step 1: Code Execution and Testing

When testing factorial(5), the output returned was 24 instead of 120, indicating a logical issue in the loop implementation.

Step 2: AI Analysis

Identified Bug:
The function contains an off-by-one error in the loop range.

Explanation:
The loop uses range(1, n), which excludes n. Therefore, the final multiplication step is skipped.

AI Suggested Correction:
Change loop to range(1, n + 1) and handle negative numbers properly.

Step 3: Manual Fix

The manual fix involved modifying the loop range to include n and optionally adding input validation for negative numbers.

Step 4: Comparison

Aspect	**AI Fix**	**Manual Fix**
Off-by-one correction	**Yes**	**Yes**
Negative input handling	**Included**	**Sometimes missed**
Zero handling	**Included**	**Included**
Error messages	**Added**	**Optional**

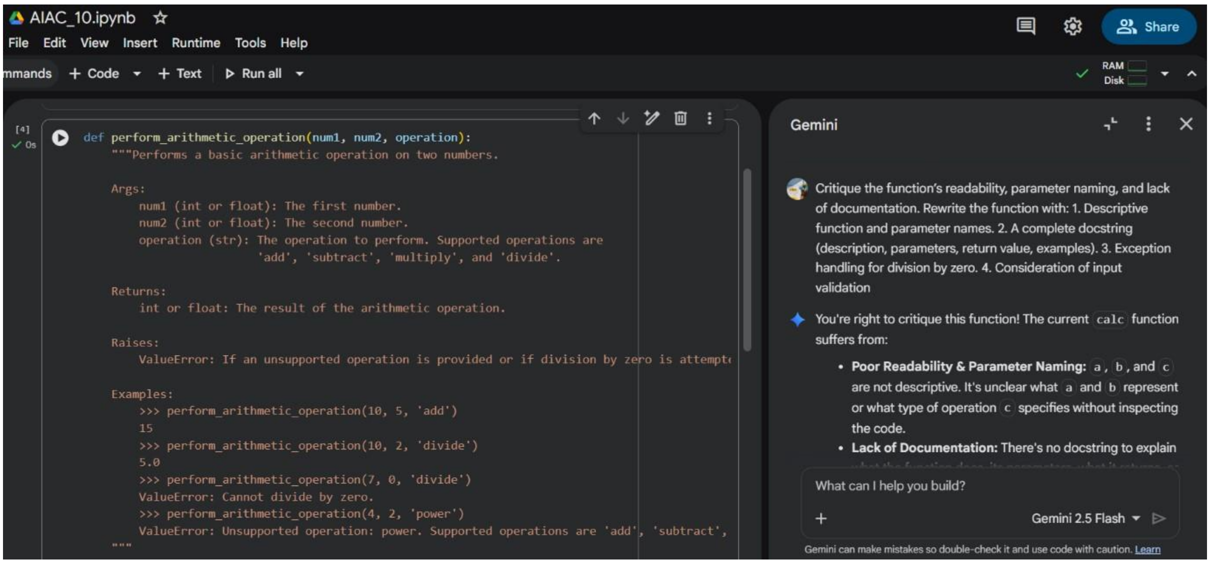
Test Results

After correction, factorial(5) correctly returns 120.

Problem Statement 2: Improving Readability & Documentation

Problem Description

The function performs arithmetic operations but is poorly structured, lacks documentation, and does not handle errors properly.

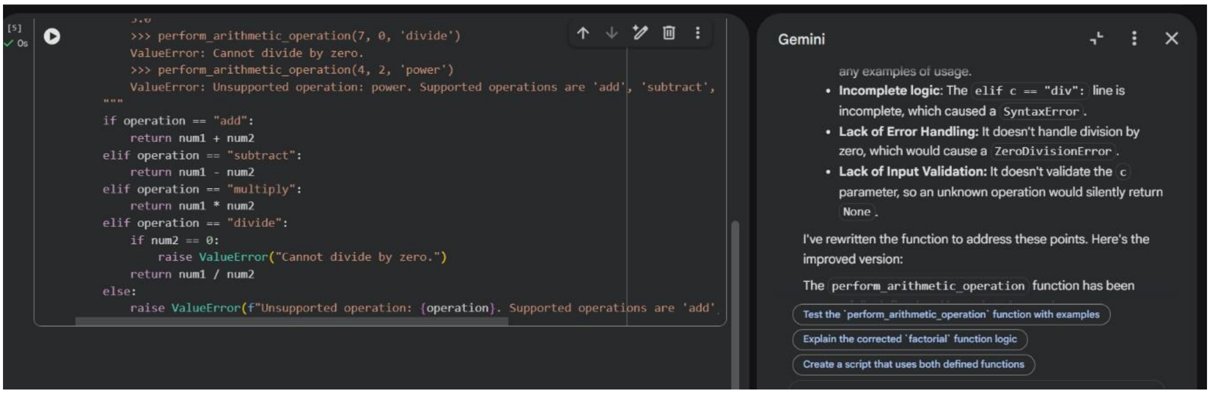


```
def perform_arithmetic_operation(num1, num2, operation):
    """Performs a basic arithmetic operation on two numbers.

    Args:
        num1 (int or float): The first number.
        num2 (int or float): The second number.
        operation (str): The operation to perform. Supported operations are
                         'add', 'subtract', 'multiply', and 'divide'.

    Returns:
        int or float: The result of the arithmetic operation.

    Raises:
        ValueError: If an unsupported operation is provided or if division by zero is attempte

    Examples:
        >>> perform_arithmetic_operation(10, 5, 'add')
        15
        >>> perform_arithmetic_operation(10, 2, 'divide')
        5.0
        >>> perform_arithmetic_operation(7, 0, 'divide')
        ValueError: Cannot divide by zero.
        >>> perform_arithmetic_operation(4, 2, 'power')
        ValueError: Unsupported operation: power. Supported operations are 'add', 'subtract',
    """
```

Critique the function’s readability, parameter naming, and lack of documentation. 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

You're right to critique this function! The current `calc` function suffers from:

- **Poor Readability & Parameter Naming:** `a`, `b`, and `c` are not descriptive. It's unclear what `a` and `b` represent or what type of operation `c` specifies without inspecting the code.
- **Lack of Documentation:** There's no docstring to explain



```
        >>> perform_arithmetic_operation(7, 0, 'divide')
        ValueError: Cannot divide by zero.
        >>> perform_arithmetic_operation(4, 2, 'power')
        ValueError: Unsupported operation: power. Supported operations are 'add', 'subtract',
    """
    if operation == "add":
        return num1 + num2
    elif operation == "subtract":
        return num1 - num2
    elif operation == "multiply":
        return num1 * num2
    elif operation == "divide":
        if num2 == 0:
            raise ValueError("Cannot divide by zero.")
        return num1 / num2
    else:
        raise ValueError(f"Unsupported operation: {operation}. Supported operations are 'add'
```

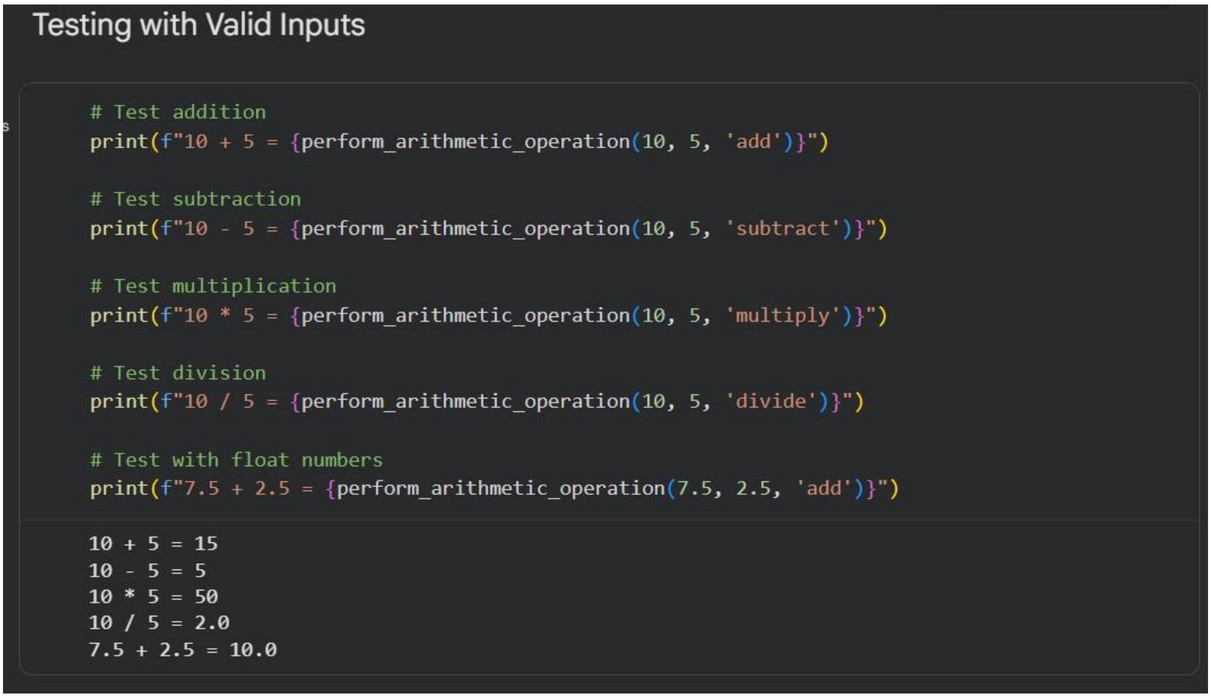
any examples of usage.

- **Incomplete logic**: The `elif c == "div":` line is incomplete, which caused a `SyntaxError`.
- **Lack of Error Handling:** It doesn't handle division by zero, which would cause a `ZeroDivisionError`.
- **Lack of Input Validation:** It doesn't validate the `c` parameter, so an unknown operation would silently return `None`.

I've rewritten the function to address these points. Here's the improved version:

The `perform_arithmetic_operation` function has been

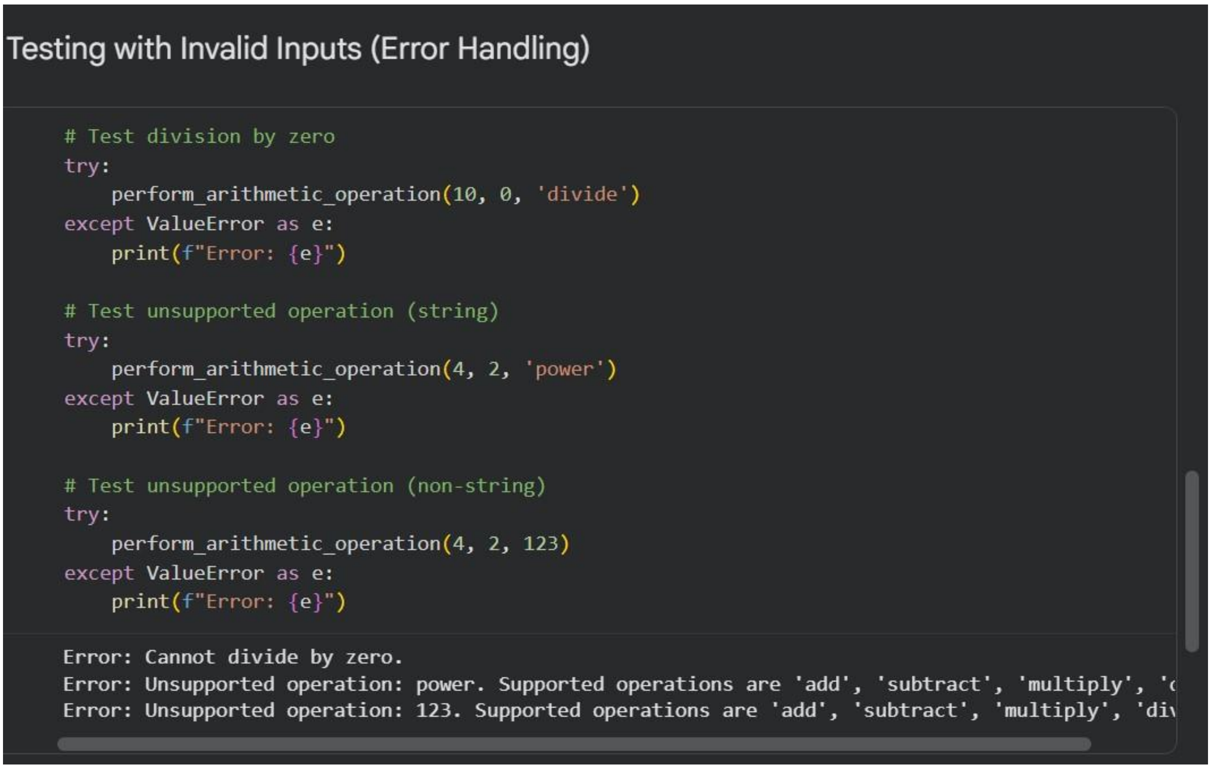
Test the `perform_arithmetic_operation` function with examples
Explain the corrected `factorial` function logic
Create a script that uses both defined functions



Testing with Valid Inputs

```
# Test addition
print(f"10 + 5 = {perform_arithmetic_operation(10, 5, 'add')}")

# Test subtraction
print(f"10 - 5 = {perform_arithmetic_operation(10, 5, 'subtract')}")

# Test multiplication
print(f"10 * 5 = {perform_arithmetic_operation(10, 5, 'multiply')}")

# Test division
print(f"10 / 5 = {perform_arithmetic_operation(10, 5, 'divide')}")

# Test with float numbers
print(f"7.5 + 2.5 = {perform_arithmetic_operation(7.5, 2.5, 'add')}")
```

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2.0
7.5 + 2.5 = 10.0
```



Testing with Invalid Inputs (Error Handling)

```
# Test division by zero
try:
    perform_arithmetic_operation(10, 0, 'divide')
except ValueError as e:
    print(f"Error: {e}")

# Test unsupported operation (string)
try:
    perform_arithmetic_operation(4, 2, 'power')
except ValueError as e:
    print(f"Error: {e}")

# Test unsupported operation (non-string)
try:
    perform_arithmetic_operation(4, 2, 123)
except ValueError as e:
    print(f"Error: {e}")
```

```
Error: Cannot divide by zero.
Error: Unsupported operation: power. Supported operations are 'add', 'subtract', 'multiply', '
Error: Unsupported operation: 123. Supported operations are 'add', 'subtract', 'multiply', 'di
```

Step 1: Code Execution and Testing

The function works for valid inputs but fails for:

- Division by zero
- Invalid operation string
- Incorrect input types

Step 2: AI Analysis

Issues Identified:

- **Poor function name (calc)**
- **Non-descriptive parameters (a, b, c)**
- **No docstring**
- **No error handling**
- **No input validation**

AI Improvements Suggested:

- **Use descriptive names**
- **Add complete docstring**
- **Add input validation**
- **Handle division by zero**
- **Raise meaningful exceptions**

Step 3: Manual Fix

Manual improvements included renaming variables and adding basic error handling.

Step 4: Comparison

Feature	**Original**	**Improved**
Readability	**Poor**	**High**
Documentation	**None**	**Complete**
Error Handling	**None**	**Proper exceptions**
Maintainability	**Low**	**High**

Test Results

The improved function correctly:

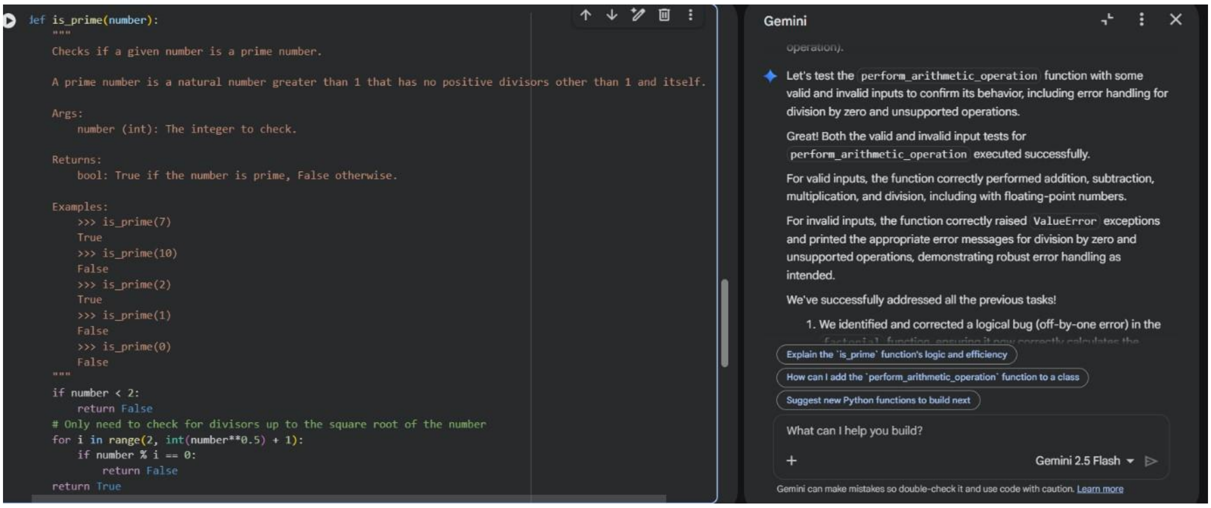
- Performs valid operations

- Raises ZeroDivisionError for division by zero
- Raises ValueError for invalid operations
- Raises TypeError for invalid input types

Problem Statement 3: Enforcing Coding Standards (PEP8)

Problem Description

The function checks whether a number is prime but violates PEP8 coding standards.



```python
def is_prime(number):
    """
    Checks if a given number is a prime number.

    A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

    Args:
        number (int): The integer to check.

    Returns:
        bool: True if the number is prime, False otherwise.

    Examples:
        >>> is_prime(7)
        True
        >>> is_prime(10)
        False
        >>> is_prime(2)
        True
        >>> is_prime(1)
        False
        >>> is_prime(0)
        False
    """
    if number < 2:
        return False
    # Only need to check for divisors up to the square root of the number
    for i in range(2, int(number**0.5) + 1):
        if number % i == 0:
            return False
    return True
```

operation).

Let's test the `perform_arithmetic_operation` function with some valid and invalid inputs to confirm its behavior, including error handling for division by zero and unsupported operations.

Great! Both the valid and invalid input tests for `perform_arithmetic_operation` executed successfully.

For valid inputs, the function correctly performed addition, subtraction, multiplication, and division, including with floating-point numbers.

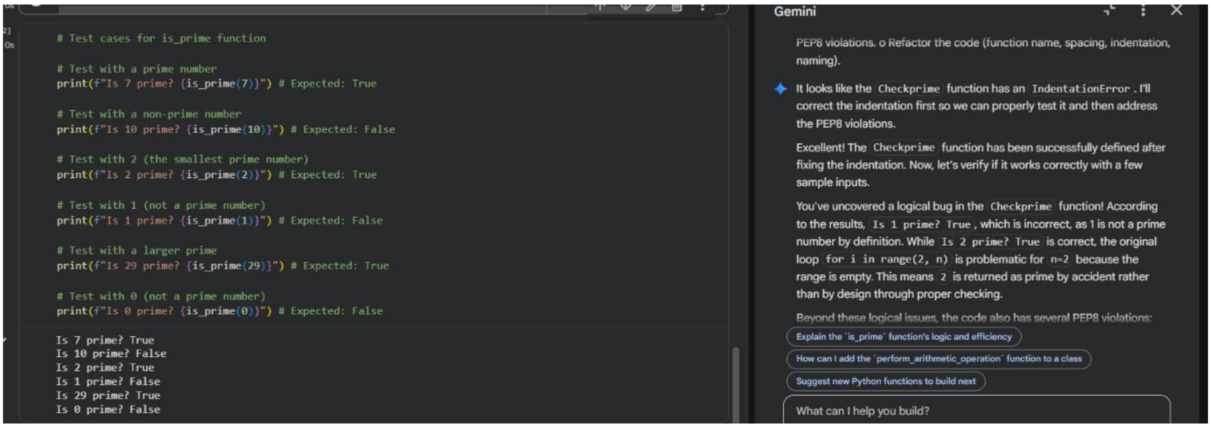
For invalid inputs, the function correctly raised `ValueError` exceptions and printed the appropriate error messages for division by zero and unsupported operations, demonstrating robust error handling as intended.

We've successfully addressed all the previous tasks!

1. We identified and corrected a logical bug (off-by-one error) in the `factorial` function, ensuring it now correctly calculates the

Explain the `is_prime` function's logic and efficiency
How can I add the `perform_arithmetic_operation` function to a class
Suggest new Python functions to build next
What can I help you build?
+
Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more



```python
# Test cases for is_prime function

# Test with a prime number
print(f"Is 7 prime? {is_prime(7)}") # Expected: True

# Test with a non-prime number
print(f"Is 10 prime? {is_prime(10)}") # Expected: False

# Test with 2 (the smallest prime number)
print(f"Is 2 prime? {is_prime(2)}") # Expected: True

# Test with 1 (not a prime number)
print(f"Is 1 prime? {is_prime(1)}") # Expected: False

# Test with a larger prime
print(f"Is 29 prime? {is_prime(29)}") # Expected: True

# Test with 0 (not a prime number)
print(f"Is 0 prime? {is_prime(0)}") # Expected: False
```

```
Is 7 prime? True
Is 10 prime? False
Is 2 prime? True
Is 1 prime? False
Is 29 prime? True
Is 0 prime? False
```

PEP8 violations. o Refactor the code (function name, spacing, indentation, naming).

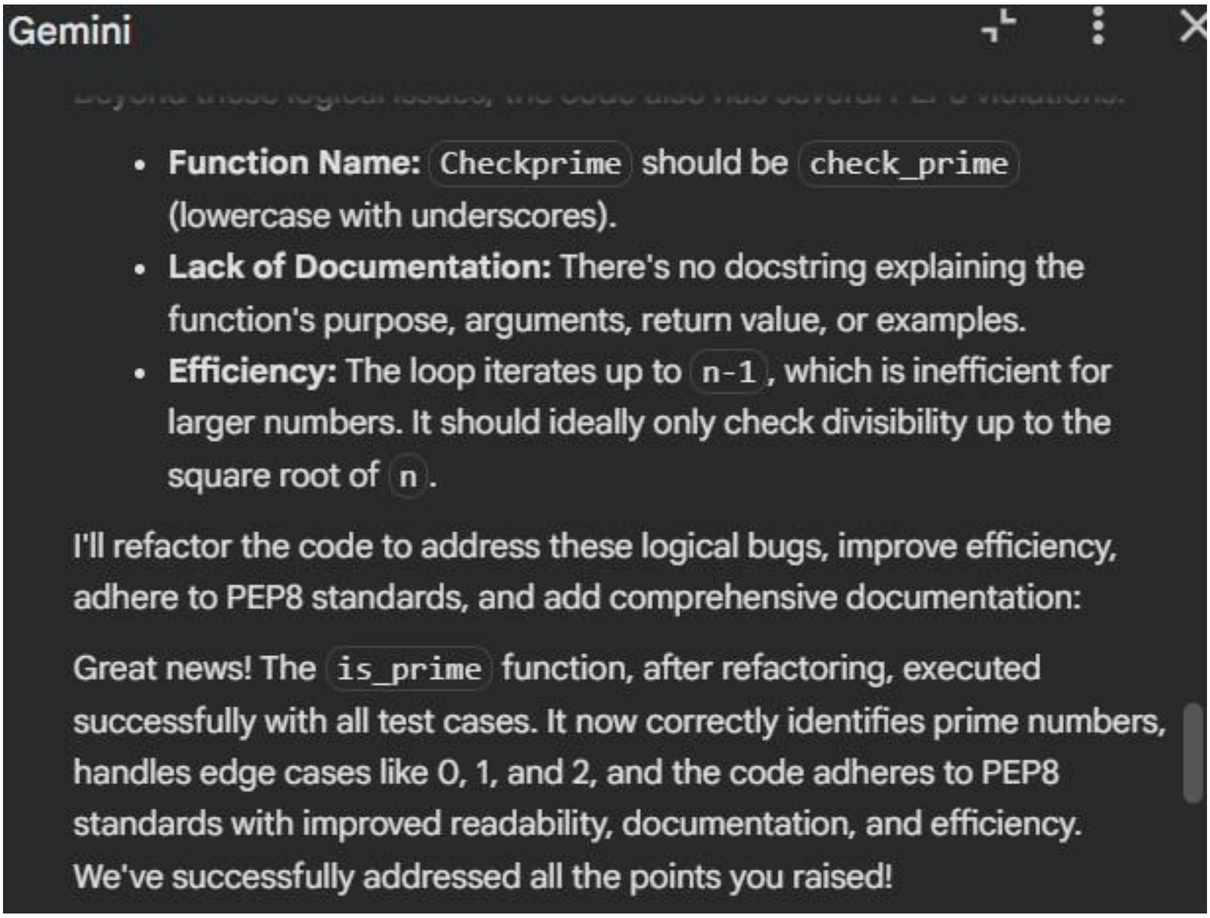
It looks like the `Checkprime` function has an `IndentationError`. I'll correct the indentation first so we can properly test it and then address the PEP8 violations.

Excellent! The `Checkprime` function has been successfully defined after fixing the indentation. Now, let's verify if it works correctly with a few sample inputs.

You've uncovered a logical bug in the `Checkprime` function! According to the results, `Is 1 prime? True`, which is incorrect, as 1 is not a prime number by definition. While `Is 2 prime? True` is correct, the original loop `for i in range(2, n)` is problematic for `n=2` because the range is empty. This means `2` is returned as prime by accident rather than by design through proper checking.

Beyond these logical issues, the code also has several PEP8 violations:

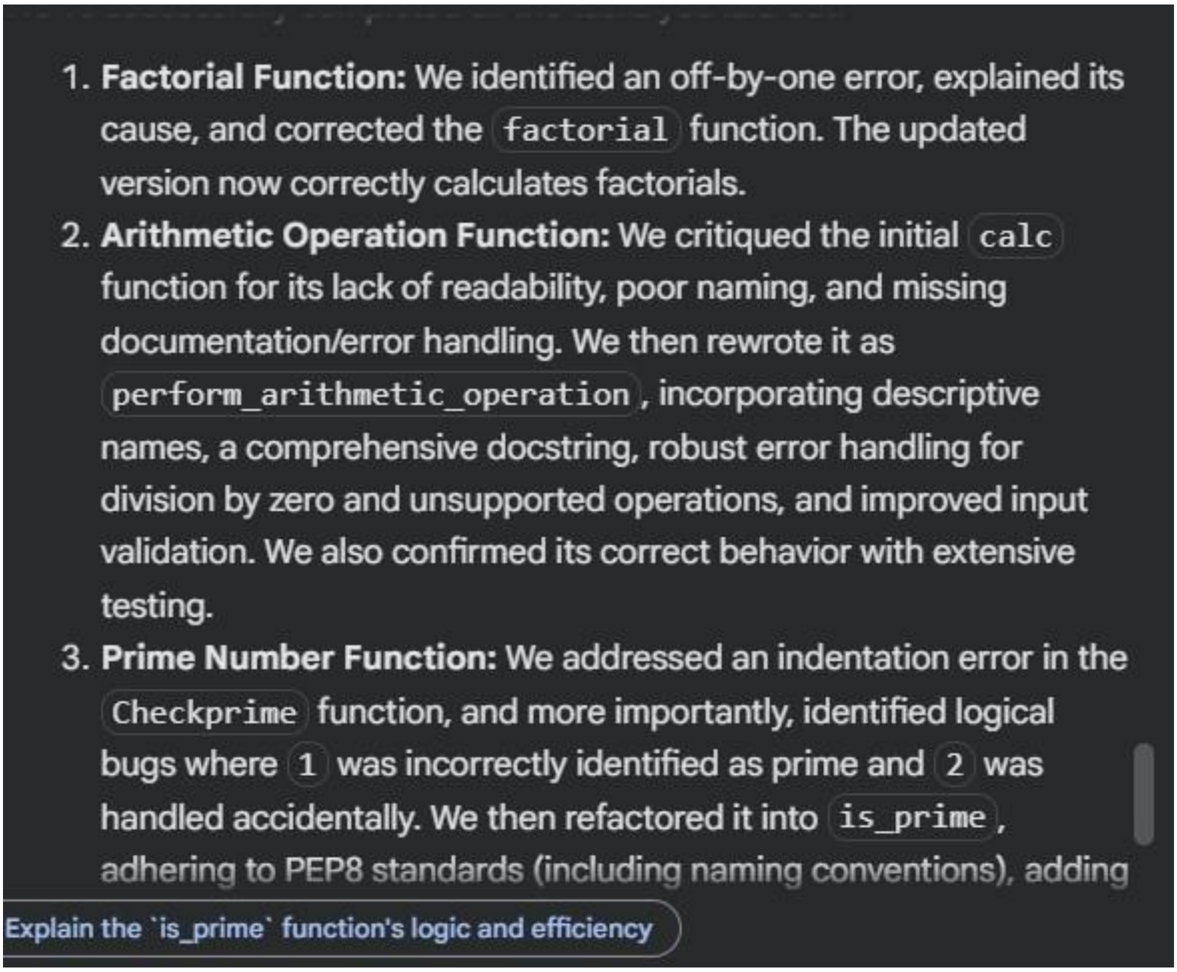
Explain the `is_prime` function's logic and efficiency
How can I add the `perform_arithmetic_operation` function to a class
Suggest new Python functions to build next
What can I help you build?



- **Function Name:** `Checkprime` should be `check_prime` (lowercase with underscores).
- **Lack of Documentation:** There's no docstring explaining the function's purpose, arguments, return value, or examples.
- **Efficiency:** The loop iterates up to `n-1`, which is inefficient for larger numbers. It should ideally only check divisibility up to the square root of `n`.

I'll refactor the code to address these logical bugs, improve efficiency, adhere to PEP8 standards, and add comprehensive documentation:

Great news! The `is_prime` function, after refactoring, executed successfully with all test cases. It now correctly identifies prime numbers, handles edge cases like 0, 1, and 2, and the code adheres to PEP8 standards with improved readability, documentation, and efficiency. We've successfully addressed all the points you raised!



1. **Factorial Function:** We identified an off-by-one error, explained its cause, and corrected the `factorial` function. The updated version now correctly calculates factorials.
2. **Arithmetic Operation Function:** We critiqued the initial `calc` function for its lack of readability, poor naming, and missing documentation/error handling. We then rewrote it as `perform_arithmetic_operation`, incorporating descriptive names, a comprehensive docstring, robust error handling for division by zero and unsupported operations, and improved input validation. We also confirmed its correct behavior with extensive testing.
3. **Prime Number Function:** We addressed an indentation error in the `Checkprime` function, and more importantly, identified logical bugs where `1` was incorrectly identified as prime and `2` was handled accidentally. We then refactored it into `is_prime`, adhering to PEP8 standards (including naming conventions), adding

Step 1: Function Testing

The function works correctly for positive numbers greater than 1. However, it incorrectly returns True for numbers less than 2.

Step 2: AI PEP8 Review

Violations Identified:

- Function name not in snake_case (Checkprime)
- Missing spaces
- No docstring
- No validation for n < 2

Refactoring Suggestions:

- Rename to check_prime

- Add docstring
- Add condition for numbers less than 2
- Improve formatting

Step 3: Applying Changes

After applying AI suggestions, the function became PEP8 compliant and handled edge cases properly.

Step 4: AI in Code Reviews

Automated AI reviews help:

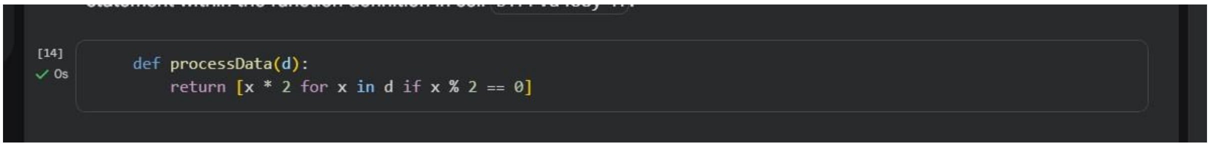
- Detect style violations instantly
- Ensure consistency across large codebases
- Reduce manual reviewer workload
- Improve overall code quality

AI speeds up code reviews but should assist human reviewers, not replace them.

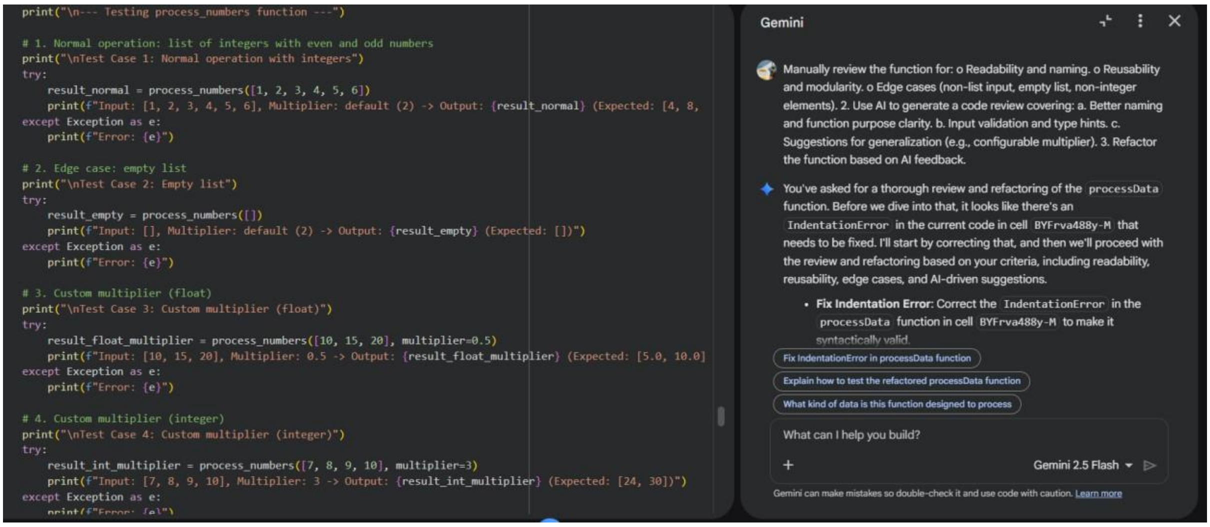
Problem Statement 4: AI as a Code Reviewer in Real Projects

Problem Description

A function processes a list and doubles even numbers. The task is to review readability, edge cases, and improve reusability.



```
[14]
✓ 0s
    def processData(d):
        return [x * 2 for x in d if x % 2 == 0]
```



```
print("\n--- Testing process_numbers function ---")

# 1. Normal operation: list of integers with even and odd numbers
print("\nTest Case 1: Normal operation with integers")
try:
    result_normal = process_numbers([1, 2, 3, 4, 5, 6])
    print(f"Input: [1, 2, 3, 4, 5, 6], Multiplier: default (2) -> Output: {result_normal} (Expected: [4, 8,
except Exception as e:
    print(f"Error: {e}")

# 2. Edge case: empty list
print("\nTest Case 2: Empty list")
try:
    result_empty = process_numbers([])
    print(f"Input: [], Multiplier: default (2) -> Output: {result_empty} (Expected: [])")
except Exception as e:
    print(f"Error: {e}")

# 3. Custom multiplier (float)
print("\nTest Case 3: Custom multiplier (float)")
try:
    result_float_multiplier = process_numbers([10, 15, 20], multiplier=0.5)
    print(f"Input: [10, 15, 20], Multiplier: 0.5 -> Output: {result_float_multiplier} (Expected: [5.0, 10.0]
except Exception as e:
    print(f"Error: {e}")

# 4. Custom multiplier (integer)
print("\nTest Case 4: Custom multiplier (integer)")
try:
    result_int_multiplier = process_numbers([7, 8, 9, 10], multiplier=3)
    print(f"Input: [7, 8, 9, 10], Multiplier: 3 -> Output: {result_int_multiplier} (Expected: [24, 30])")
except Exception as e:
    print(f"Error: {e}")
```

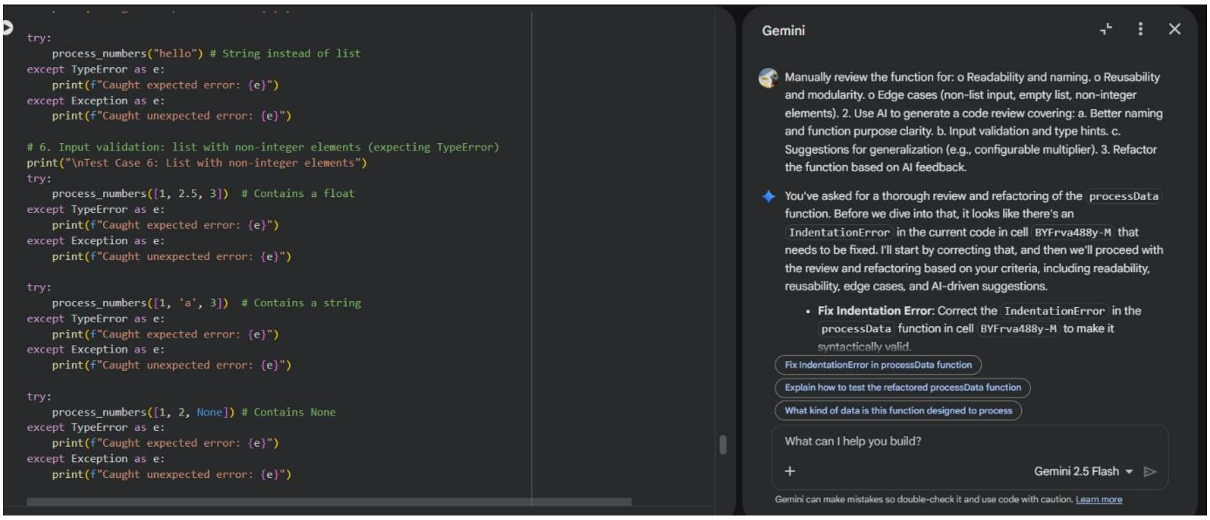
Manually review the function for: o Readability and naming. o Reusability and modularity. o Edge cases (non-list input, empty list, non-integer elements). 2. Use AI to generate a code review covering: a. Better naming and function purpose clarity. b. Input validation and type hints. c. Suggestions for generalization (e.g., configurable multiplier). 3. Refactor the function based on AI feedback.

You've asked for a thorough review and refactoring of the `processData` function. Before we dive into that, it looks like there's an `IndentationError` in the current code in cell `BYFrva488y-M` that needs to be fixed. I'll start by correcting that, and then we'll proceed with the review and refactoring based on your criteria, including readability, reusability, edge cases, and AI-driven suggestions.

- **Fix Indentation Error**: Correct the `IndentationError` in the `processData` function in cell `BYFrva488y-M` to make it syntactically valid.

Fix IndentationError in processData function
Explain how to test the refactored processData function
What kind of data is this function designed to process
What can I help you build?
+
Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
try:
    process_numbers("hello") # String instead of list
except TypeError as e:
    print(f"Caught expected error: {e}")
except Exception as e:
    print(f"Caught unexpected error: {e}")

# 6. Input validation: list with non-integer elements (expecting TypeError)
print("\nTest Case 6: List with non-integer elements")
try:
    process_numbers([1, 2.5, 3])  # Contains a float
except TypeError as e:
    print(f"Caught expected error: {e}")
except Exception as e:
    print(f"Caught unexpected error: {e}")

try:
    process_numbers([1, 'a', 3])  # Contains a string
except TypeError as e:
    print(f"Caught expected error: {e}")
except Exception as e:
    print(f"Caught unexpected error: {e}")

try:
    process_numbers([1, 2, None]) # Contains None
except TypeError as e:
    print(f"Caught expected error: {e}")
except Exception as e:
    print(f"Caught unexpected error: {e}")
```

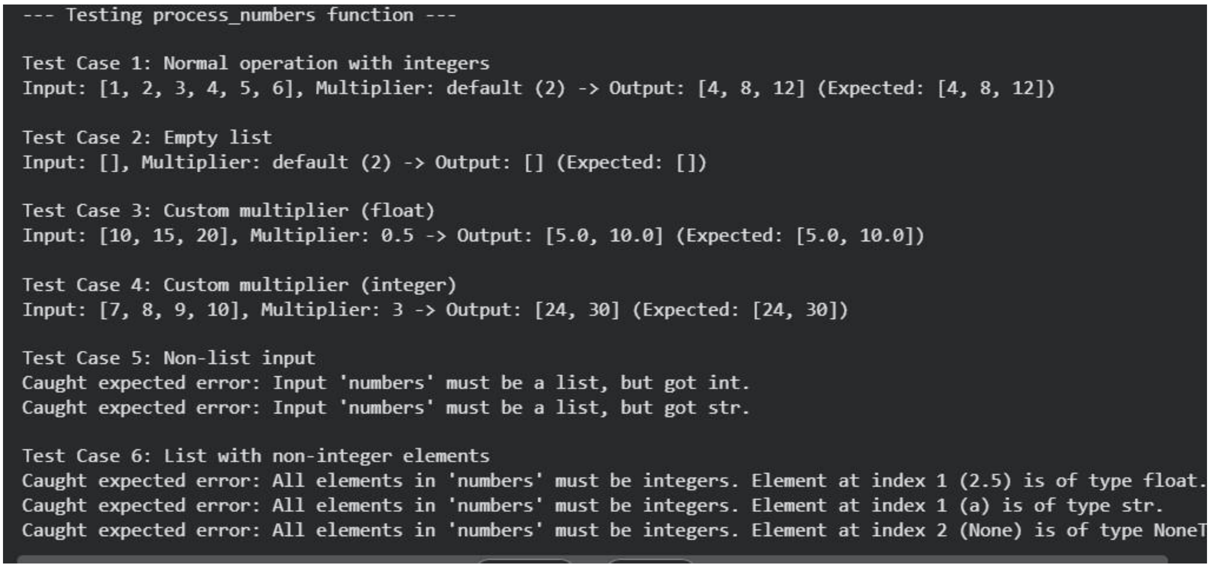
Manually review the function for: o Readability and naming. o Reusability and modularity. o Edge cases (non-list input, empty list, non-integer elements). 2. Use AI to generate a code review covering: a. Better naming and function purpose clarity. b. Input validation and type hints. c. Suggestions for generalization (e.g., configurable multiplier). 3. Refactor the function based on AI feedback.

You've asked for a thorough review and refactoring of the `processData` function. Before we dive into that, it looks like there's an `IndentationError` in the current code in cell `BYFrva488y-M` that needs to be fixed. I'll start by correcting that, and then we'll proceed with the review and refactoring based on your criteria, including readability, reusability, edge cases, and AI-driven suggestions.

- **Fix Indentation Error**: Correct the `IndentationError` in the `processData` function in cell `BYFrva488y-M` to make it syntactically valid.

Fix IndentationError in processData function
Explain how to test the refactored processData function
What kind of data is this function designed to process
What can I help you build?
+
Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more



```
--- Testing process_numbers function ---

Test Case 1: Normal operation with integers
Input: [1, 2, 3, 4, 5, 6], Multiplier: default (2) -> Output: [4, 8, 12] (Expected: [4, 8, 12])

Test Case 2: Empty list
Input: [], Multiplier: default (2) -> Output: [] (Expected: [])

Test Case 3: Custom multiplier (float)
Input: [10, 15, 20], Multiplier: 0.5 -> Output: [5.0, 10.0] (Expected: [5.0, 10.0])

Test Case 4: Custom multiplier (integer)
Input: [7, 8, 9, 10], Multiplier: 3 -> Output: [24, 30] (Expected: [24, 30])

Test Case 5: Non-list input
Caught expected error: Input 'numbers' must be a list, but got int.
Caught expected error: Input 'numbers' must be a list, but got str.

Test Case 6: List with non-integer elements
Caught expected error: All elements in 'numbers' must be integers. Element at index 1 (2.5) is of type float.
Caught expected error: All elements in 'numbers' must be integers. Element at index 1 (a) is of type str.
Caught expected error: All elements in 'numbers' must be integers. Element at index 2 (None) is of type NoneT
```

Step 1: Manual Review

Issues identified:

- Poor naming (processData, d)
- No input validation
- No type hints
- Not reusable
- No documentation

Step 2: AI Code Review Suggestions

AI recommended:

- Rename function to reflect purpose clearly
- Add type hints
- Add input validation
- Handle non-list inputs
- Generalize multiplier value

Step 3: Refactoring

The improved version includes:

- Proper naming
- Type hints
- Validation
- Cleaner structure
- Better maintainability

Step 4: Reflection

AI should be used as a supporting reviewer.
It improves speed and consistency, but human reviewers understand:

- Business logic

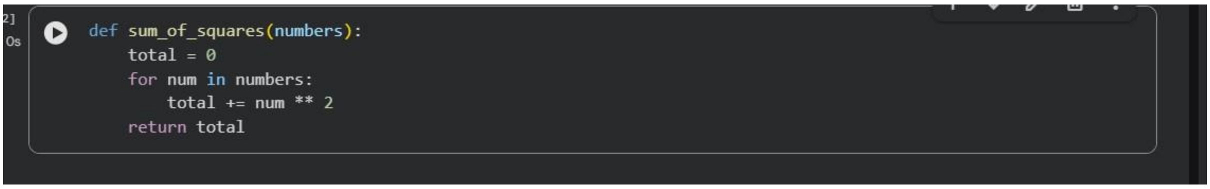
- Architecture decisions
- Project context

AI enhances productivity but should not replace human judgment.

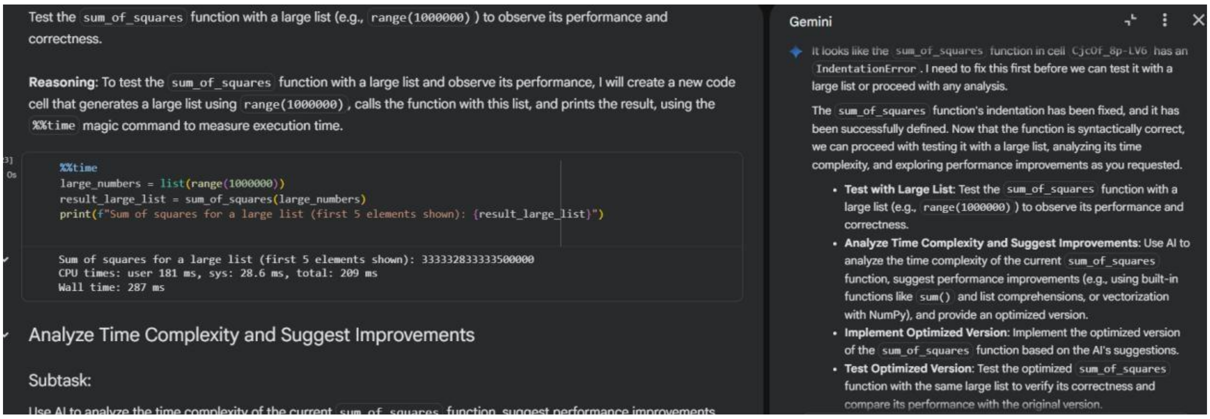
Problem Statement 5: AI-Assisted Performance Optimization

Problem Description

A function calculates sum of squares but is slow for very large datasets.



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



Test the `sum_of_squares` function with a large list (e.g., `range(1000000)`) to observe its performance and correctness.

Reasoning: To test the `sum_of_squares` function with a large list and observe its performance, I will create a new code cell that generates a large list using `range(1000000)` , calls the function with this list, and prints the result, using the `%%time` magic command to measure execution time.

```python
%%time
large_numbers = list(range(1000000))
result_large_list = sum_of_squares(large_numbers)
print(f"Sum of squares for a large list (first 5 elements shown): {result_large_list}")
```

```
Sum of squares for a large list (first 5 elements shown): 333332833333500000
CPU times: user 181 ms, sys: 28.6 ms, total: 209 ms
Wall time: 287 ms
```

Analyze Time Complexity and Suggest Improvements

Subtask:

Use AI to analyze the time complexity of the current `sum_of_squares` function, suggest performance improvements

Gemini

It looks like the `sum_of_squares` function in cell `CjcOf_8p-LV6` has an `IndentationError` . I need to fix this first before we can test it with a large list or proceed with any analysis.

The `sum_of_squares` function's indentation has been fixed, and it has been successfully defined. Now that the function is syntactically correct, we can proceed with testing it with a large list, analyzing its time complexity, and exploring performance improvements as you requested.

- **Test with Large List**: Test the `sum_of_squares` function with a large list (e.g., `range(1000000)`) to observe its performance and correctness.
- **Analyze Time Complexity and Suggest Improvements**: Use AI to analyze the time complexity of the current `sum_of_squares` function, suggest performance improvements (e.g., using built-in functions like `sum()` and list comprehensions, or vectorization with NumPy), and provide an optimized version.
- **Implement Optimized Version**: Implement the optimized version of the `sum_of_squares` function based on the AI's suggestions.
- **Test Optimized Version**: Test the optimized `sum_of_squares` function with the same large list to verify its correctness and compare its performance with the original version.

Step 1: Testing

Testing with range(1000000) shows noticeable execution time.

Step 2: AI Performance Analysis

Time Complexity:
O(n)

AI Suggestions:

- Use built-in sum() function
- Use generator expressions
- Consider NumPy for vectorization (for very large datasets)

Step 3: Optimized Version

The optimized function uses:

- Generator expression
- Built-in sum()

This reduces Python loop overhead and improves performance.

Step 4: Performance Comparison

Version	Speed	Readability
Original Loop	Slower	Medium
Optimized	Faster	High

Step 5: Trade-Off Discussion

- Built-in functions improve performance.
- NumPy improves speed significantly but it adds dependency.
- Readability should not be sacrificed unnecessarily.
- Optimization is important only when performance becomes a bottleneck.