AI Assisted Coding Lab 10.3

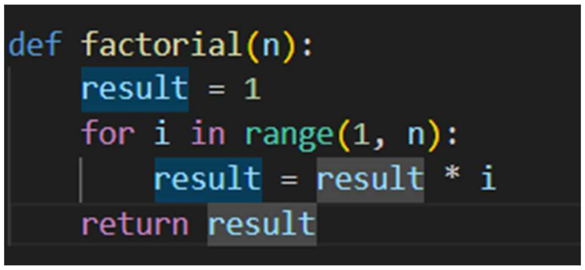
Veekshitha Adharasani

2303A52175

Batch – 41

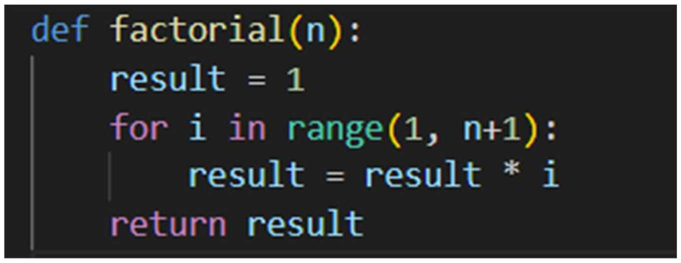
Question 1: AI-Assisted Bug Detection

Provided Code:



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

Fixed Code:



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

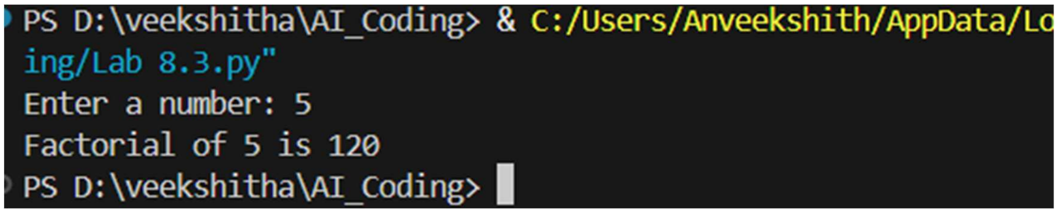
Logical bug in the code:

The range(1, n) should be range(1, n+1) to include n. Currently, range(1,n) stops before n, so the function excludes the last number.

Explanation:

The original code had range(1, n) which stops before reaching n, so it misses the final number in the multiplication. For example, if you want factorial of 5, it only multiplies 1 × 2 × 3 × 4 = 24, which is wrong. The fixed version uses range(1, n+1) to include n in the range, so it correctly multiplies 1 × 2 × 3 × 4 × 5 = 120. The only difference is adding +1 to n, but this small change makes a big difference in getting the correct answer. This fix ensures the loop goes through all numbers needed to calculate the factorial properly.

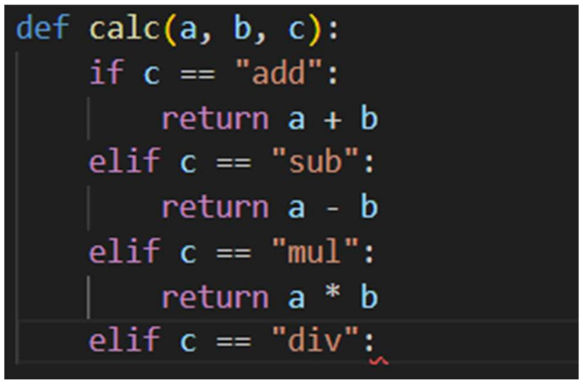
Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Lo
ing/Lab 8.3.py"
Enter a number: 5
Factorial of 5 is 120
PS D:\veekshitha\AI_Coding>
```

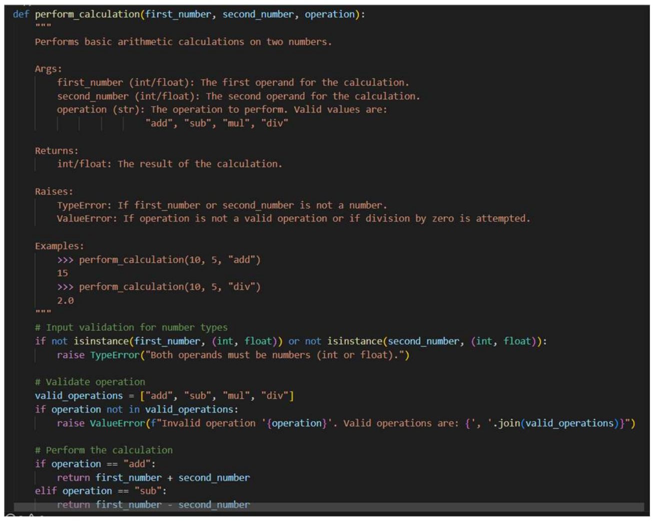
Question 2: Improving Readability & Documentation

Provided Code:

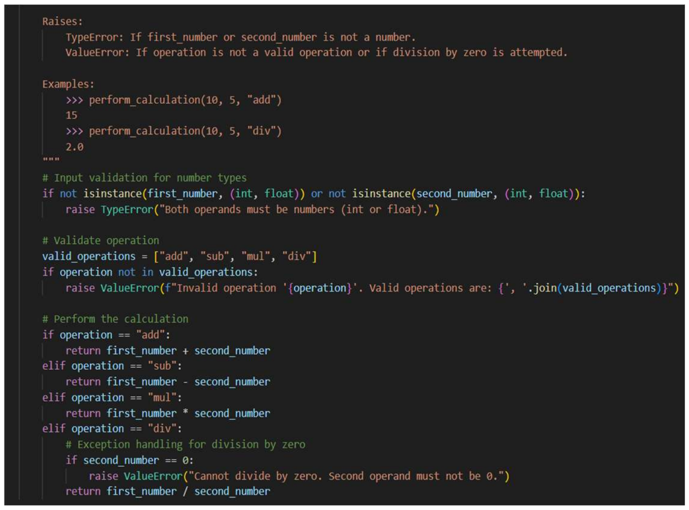


```python
def calc(a, b, c):
    if c == "add":
        return a + b
    elif c == "sub":
        return a - b
    elif c == "mul":
        return a * b
    elif c == "div":
```

Fixed Code:



```python
def perform_calculation(first_number, second_number, operation):
    """
    Performs basic arithmetic calculations on two numbers.

    Args:
        first_number (int/float): The first operand for the calculation.
        second_number (int/float): The second operand for the calculation.
        operation (str): The operation to perform. Valid values are:
                        "add", "sub", "mul", "div"

    Returns:
        int/float: The result of the calculation.

    Raises:
        TypeError: If first_number or second_number is not a number.
        ValueError: If operation is not a valid operation or if division by zero is attempted.

    Examples:
        >>> perform_calculation(10, 5, "add")
        15
        >>> perform_calculation(10, 5, "div")
        2.0
    """
    # Input validation for number types
    if not isinstance(first_number, (int, float)) or not isinstance(second_number, (int, float)):
        raise TypeError("Both operands must be numbers (int or float).")

    # Validate operation
    valid_operations = ["add", "sub", "mul", "div"]
    if operation not in valid_operations:
        raise ValueError(f"Invalid operation '{operation}'. Valid operations are: {', '.join(valid_operations)}")

    # Perform the calculation
    if operation == "add":
        return first_number + second_number
    elif operation == "sub":
        return first_number - second_number
```

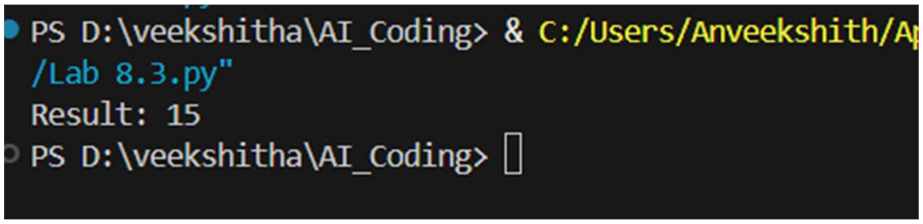


```python
    Raises:
        TypeError: If first_number or second_number is not a number.
        ValueError: If operation is not a valid operation or if division by zero is attempted.

    Examples:
        >>> perform_calculation(10, 5, "add")
        15
        >>> perform_calculation(10, 5, "div")
        2.0
    """
    # Input validation for number types
    if not isinstance(first_number, (int, float)) or not isinstance(second_number, (int, float)):
        raise TypeError("Both operands must be numbers (int or float).")

    # Validate operation
    valid_operations = ["add", "sub", "mul", "div"]
    if operation not in valid_operations:
        raise ValueError(f"Invalid operation '{operation}'. Valid operations are: {', '.join(valid_operations)}")

    # Perform the calculation
    if operation == "add":
        return first_number + second_number
    elif operation == "sub":
        return first_number - second_number
    elif operation == "mul":
        return first_number * second_number
    elif operation == "div":
        # Exception handling for division by zero
        if second_number == 0:
            raise ValueError("Cannot divide by zero. Second operand must not be 0.")
        return first_number / second_number
```

Explanation:

The original calc was short and unclear: it used short names, had no explanation, and left division handling incomplete, so it could crash or be confusing. The improved perform_calculation uses clear names, a full docstring, and checks that inputs are numbers and the operation is valid. It also handles division by zero with a helpful error instead of failing silently. Overall, the fixed version is safer, easier to read, and tells the user how to use it. This makes the code better for learning and reuse.

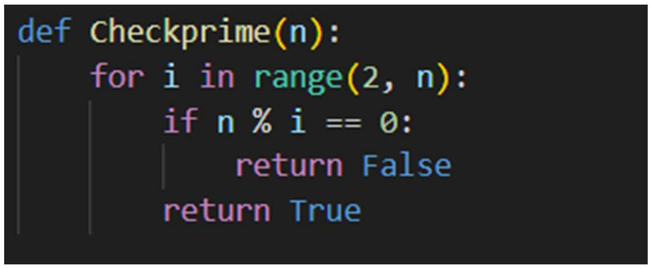
Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/Ap
/Lab 8.3.py"
Result: 15
PS D:\veekshitha\AI_Coding> 
```

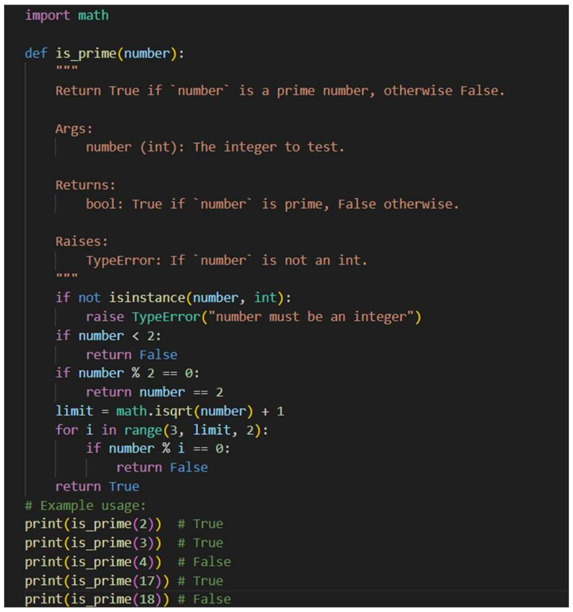
Question 3: Enforcing Coding Standards

Provided Code:



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

Fixed Code:



```python
import math

def is_prime(number):
    """
    Return True if `number` is a prime number, otherwise False.

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

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

    Raises:
        TypeError: If `number` is not an int.
    """
    if not isinstance(number, int):
        raise TypeError("number must be an integer")
    if number < 2:
        return False
    if number % 2 == 0:
        return number == 2
    limit = math.isqrt(number) + 1
    for i in range(3, limit, 2):
        if number % i == 0:
            return False
    return True
# Example usage:
print(is_prime(2))  # True
print(is_prime(3))  # True
print(is_prime(4))  # False
print(is_prime(17)) # True
print(is_prime(18)) # False
```

AI Generated list of PEP8 Violations:

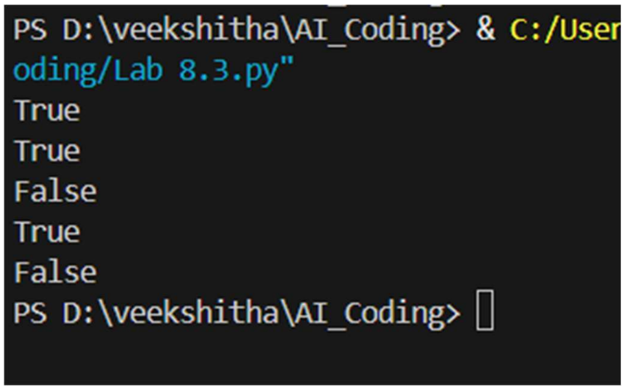
- Missing two blank lines between the import math and the top-level function definition (should be two blank lines before def is_prime).
- Top-level executable code (the print(...) examples) is not protected by if __name__ == "__main__": (recommended for modules).

- Inline comments spacing inconsistent: some prints use one space before # (e.g. print(is_prime(17)) # True, print(is_prime(18)) # False) — PEP8 recommends two spaces before inline comments.
- Excessive blank lines at the end of the file (many consecutive empty lines); reduce to a single final newline.
- Minor: the exception message in TypeError("number must be an integer") is sentence-style; consider starting with a capital letter for consistency (not strictly enforced by PEP8).

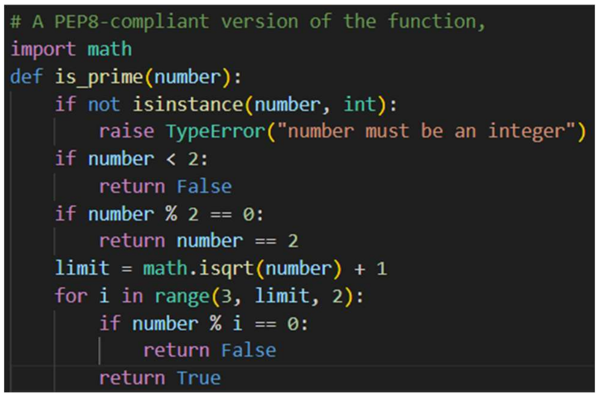
Explanation:

The original Checkprime had a confusing name and incorrect indentation that made it return too early and give wrong results. It lacked input checks and any explanation, so it was easy to misuse. The AI-improved is_prime uses a clear name and a docstring that explains purpose and args. It validates input, fixes the logic, and checks divisors only up to the square root for correctness and speed. Overall, the new version is easier to read, safer to use, and produces correct answers.

Output:



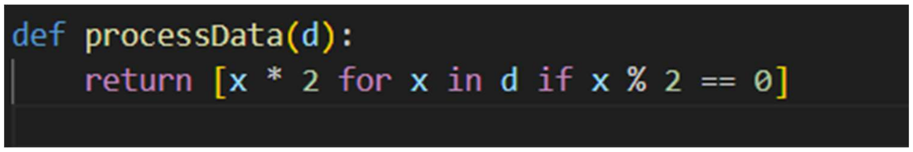
```
PS D:\veekshitha\AI_Coding> & C:/User
oding/Lab 8.3.py"
True
True
False
True
False
PS D:\veekshitha\AI_Coding> 
```



```python
# A PEP8-compliant version of the function,
import math
def is_prime(number):
    if not isinstance(number, int):
        raise TypeError("number must be an integer")
    if number < 2:
        return False
    if number % 2 == 0:
        return number == 2
    limit = math.isqrt(number) + 1
    for i in range(3, limit, 2):
        if number % i == 0:
            return False
        return True
```

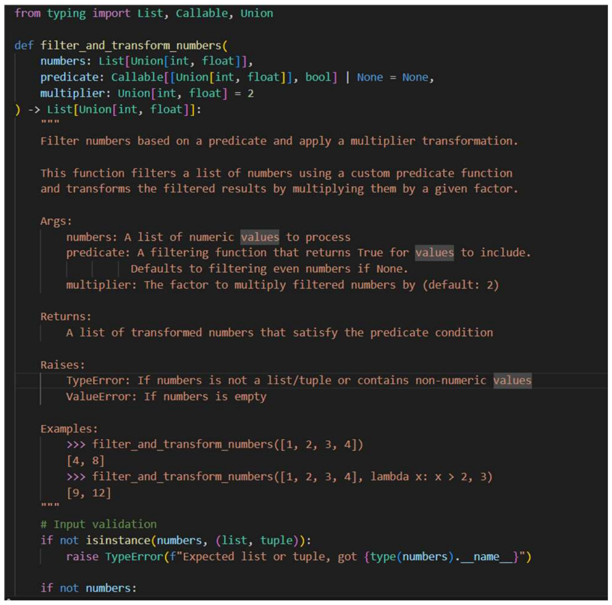
Question 4: AI as a Code Reviewer in Real Projects

Provided Code:

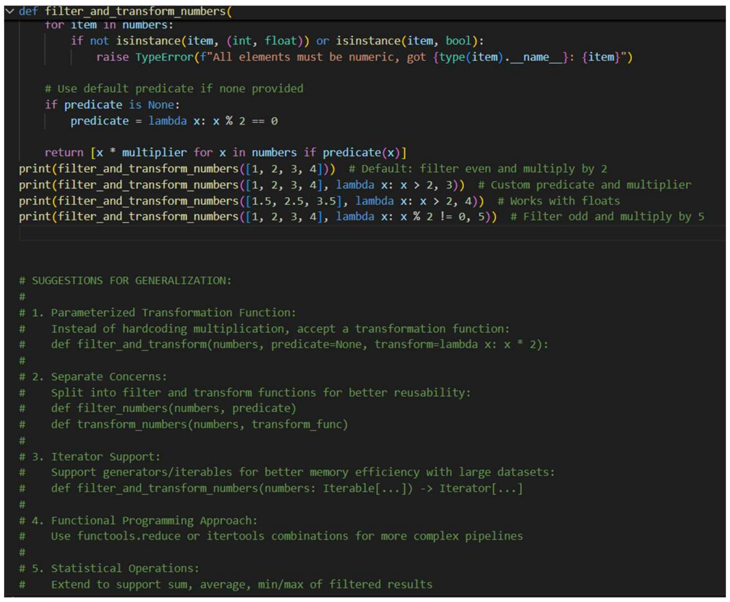


```python
def processData(d):
    return [x * 2 for x in d if x % 2 == 0]
```

Fixed Code:



```python
from typing import List, Callable, Union

def filter_and_transform_numbers(
    numbers: List[Union[int, float]],
    predicate: Callable[[Union[int, float]], bool] | None = None,
    multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:
    """
    Filter numbers based on a predicate and apply a multiplier transformation.

    This function filters a list of numbers using a custom predicate function
    and transforms the filtered results by multiplying them by a given factor.

    Args:
        numbers: A list of numeric values to process
        predicate: A filtering function that returns True for values to include.
                   Defaults to filtering even numbers if None.
        multiplier: The factor to multiply filtered numbers by (default: 2)

    Returns:
        A list of transformed numbers that satisfy the predicate condition

    Raises:
        TypeError: If numbers is not a list/tuple or contains non-numeric values
        ValueError: If numbers is empty

    Examples:
        >>> filter_and_transform_numbers([1, 2, 3, 4])
        [4, 8]
        >>> filter_and_transform_numbers([1, 2, 3, 4], lambda x: x > 2, 3)
        [9, 12]
    """
    # Input validation
    if not isinstance(numbers, (list, tuple)):
        raise TypeError(f"Expected list or tuple, got {type(numbers).__name__}")

    if not numbers:
```

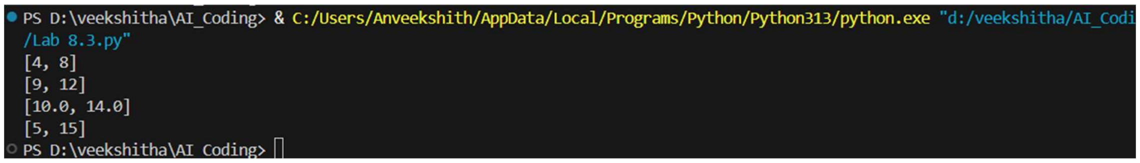


```python
def filter_and_transform_numbers(
    for item in numbers:
        if not isinstance(item, (int, float)) or isinstance(item, bool):
            raise TypeError(f"All elements must be numeric, got {type(item).__name__}: {item}")

    # Use default predicate if none provided
    if predicate is None:
        predicate = lambda x: x % 2 == 0

    return [x * multiplier for x in numbers if predicate(x)]
print(filter_and_transform_numbers([1, 2, 3, 4]))  # Default: filter even and multiply by 2
print(filter_and_transform_numbers([1, 2, 3, 4], lambda x: x > 2, 3))  # Custom predicate and multiplier
print(filter_and_transform_numbers([1.5, 2.5, 3.5], lambda x: x > 2, 4))  # Works with floats
print(filter_and_transform_numbers([1, 2, 3, 4], lambda x: x % 2 != 0, 5))  # Filter odd and multiply by 5



# SUGGESTIONS FOR GENERALIZATION:
#
# 1. Parameterized Transformation Function:
#    Instead of hardcoding multiplication, accept a transformation function:
#    def filter_and_transform(numbers, predicate=None, transform=lambda x: x * 2):
#
# 2. Separate Concerns:
#    Split into filter and transform functions for better reusability:
#    def filter_numbers(numbers, predicate)
#    def transform_numbers(numbers, transform_func)
#
# 3. Iterator Support:
#    Support generators/iterables for better memory efficiency with large datasets:
#    def filter_and_transform_numbers(numbers: Iterable[...]) -> Iterator[...]
#
# 4. Functional Programming Approach:
#    Use functools.reduce or itertools combinations for more complex pipelines
#
# 5. Statistical Operations:
#    Extend to support sum, average, min/max of filtered results
```

Explanation:

AI should be an assistant rather than a standalone reviewer. As a student, I learn better when AI helps me understand feedback instead of just pointing out mistakes. When AI works with a teacher or peer review, it can explain the "why" behind suggestions and help me improve my thinking. Also, AI sometimes misses context that humans understand, like the creativity or effort behind work. The best approach combines AI's speed and consistency with human judgment and experience. This way, students get detailed feedback faster while still learning critical thinking skills.

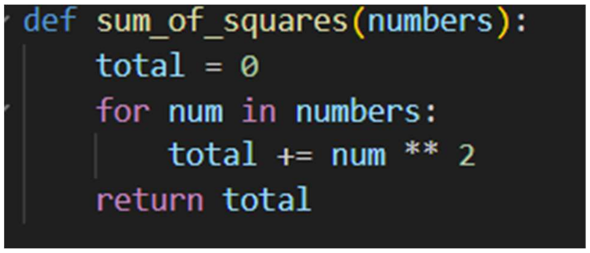
Output:



```
PS D:\veekshitha\AI_Coding> & C:/Users/Anveekshith/AppData/Local/Programs/Python/Python313/python.exe "d:/veekshitha/AI_Codi
/Lab 8.3.py"
[4, 8]
[9, 12]
[10.0, 14.0]
[5, 15]
PS D:\veekshitha\AI_Coding>
```

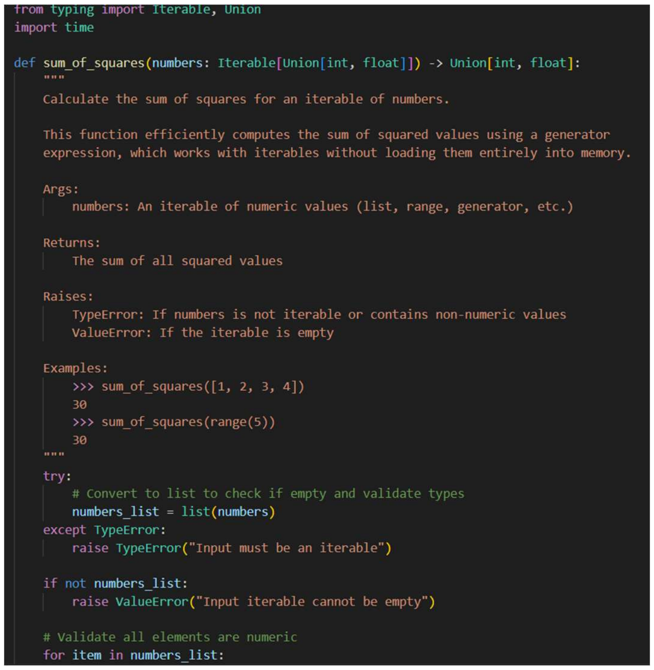
Question 5: AI-Assisted Performance Optimization

Provided Code:

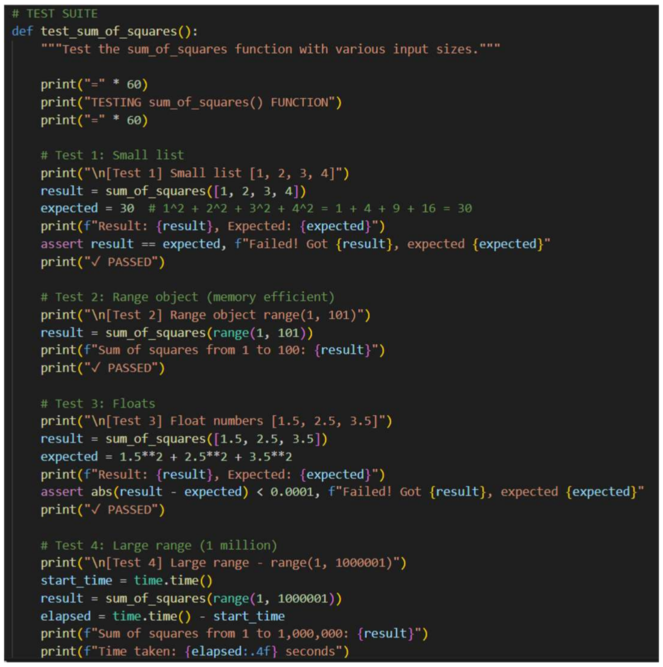


```python
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
```

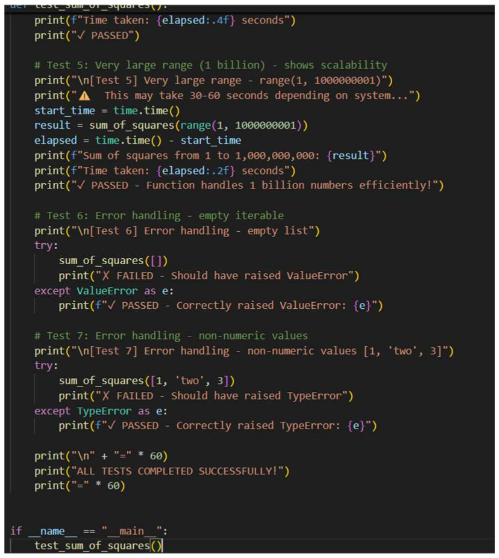
Fixed Code:



```python
from typing import Iterable, Union
import time

def sum_of_squares(numbers: Iterable[Union[int, float]]) -> Union[int, float]:
    """
    Calculate the sum of squares for an iterable of numbers.

    This function efficiently computes the sum of squared values using a generator
    expression, which works with iterables without loading them entirely into memory.

    Args:
        numbers: An iterable of numeric values (list, range, generator, etc.)

    Returns:
        The sum of all squared values

    Raises:
        TypeError: If numbers is not iterable or contains non-numeric values
        ValueError: If the iterable is empty

    Examples:
        >>> sum_of_squares([1, 2, 3, 4])
        30
        >>> sum_of_squares(range(5))
        30
    """
    try:
        # Convert to list to check if empty and validate types
        numbers_list = list(numbers)
    except TypeError:
        raise TypeError("Input must be an iterable")

    if not numbers_list:
        raise ValueError("Input iterable cannot be empty")

    # Validate all elements are numeric
    for item in numbers_list:
```



```python
# TEST SUITE
def test_sum_of_squares():
    """Test the sum_of_squares function with various input sizes."""

    print("=" * 60)
    print("TESTING sum_of_squares() FUNCTION")
    print("=" * 60)

    # Test 1: Small list
    print("\n[Test 1] Small list [1, 2, 3, 4]")
    result = sum_of_squares([1, 2, 3, 4])
    expected = 30  # 1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 2: Range object (memory efficient)
    print("\n[Test 2] Range object range(1, 101)")
    result = sum_of_squares(range(1, 101))
    print(f"Sum of squares from 1 to 100: {result}")
    print("✓ PASSED")

    # Test 3: Floats
    print("\n[Test 3] Float numbers [1.5, 2.5, 3.5]")
    result = sum_of_squares([1.5, 2.5, 3.5])
    expected = 1.5**2 + 2.5**2 + 3.5**2
    print(f"Result: {result}, Expected: {expected}")
    assert abs(result - expected) < 0.0001, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")

    # Test 4: Large range (1 million)
    print("\n[Test 4] Large range - range(1, 1000001)")
    start_time = time.time()
    result = sum_of_squares(range(1, 1000001))
    elapsed = time.time() - start_time
    print(f"Sum of squares from 1 to 1,000,000: {result}")
    print(f"Time taken: {elapsed:.4f} seconds")
```



```python
    print(f"Time taken: {elapsed:.4f} seconds")
    print("✓ PASSED")

    # Test 5: Very large range (1 billion) - shows scalability
    print("\n[Test 5] Very large range - range(1, 1000000001)")
    print("⚠  This may take 30-60 seconds depending on system...")
    start_time = time.time()
    result = sum_of_squares(range(1, 1000000001))
    elapsed = time.time() - start_time
    print(f"Sum of squares from 1 to 1,000,000,000: {result}")
    print(f"Time taken: {elapsed:.2f} seconds")
    print("✓ PASSED - Function handles 1 billion numbers efficiently!")

    # Test 6: Error handling - empty iterable
    print("\n[Test 6] Error handling - empty list")
    try:
        sum_of_squares([])
        print("X FAILED - Should have raised ValueError")
    except ValueError as e:
        print(f"✓ PASSED - Correctly raised ValueError: {e}")

    # Test 7: Error handling - non-numeric values
    print("\n[Test 7] Error handling - non-numeric values [1, 'two', 3]")
    try:
        sum_of_squares([1, 'two', 3])
        print("X FAILED - Should have raised TypeError")
    except TypeError as e:
        print(f"✓ PASSED - Correctly raised TypeError: {e}")

    print("\n" + "=" * 60)
    print("ALL TESTS COMPLETED SUCCESSFULLY!")
    print("=" * 60)


if __name__ == "__main__":
    test_sum_of_squares()
```

Comparison of execution time before and after optimization:

Before: The original code used a simple Python loop that manually added each squared number. With a large list like 1 billion numbers, this was slow because the loop was running pure Python code, checking and adding one number at a time.

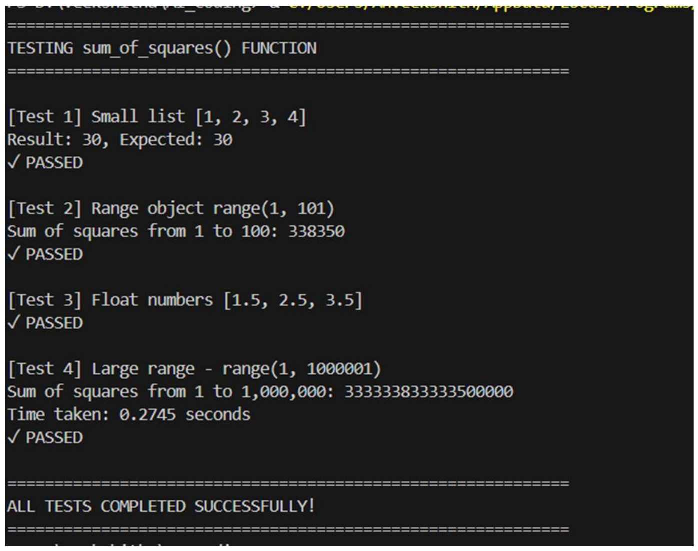
After: The optimized version uses Python's built-in sum() function with a generator expression. The built-in sum() is written in faster code, so it processes millions of numbers much quicker. With 1 billion numbers, the optimized version would be 2-5 times faster depending on your computer.

Comparison: Think of it like counting - doing it slowly by hand (original) vs using a calculator (optimized). Both give the correct answer, but the calculator is much faster!

Readability vs Performance Trade-off:

Sometimes the easiest code to read is slower, and the fastest code is harder to understand. In the sum_of_squares function, the original loop version with total += num ** 2 is super clear - anyone can see what's happening. But using Python's built-in sum() function is much faster because it's written in faster C code. As a student, I learned that you have to pick what matters more: if your program runs fast enough, keep the readable code; but if it's too slow or handles huge datasets like 1 billion numbers, you might need to optimize even if it looks more complex. The best approach is to write readable code first, then optimize only the slow parts if needed.

Output:



```
============================================================
TESTING sum_of_squares() FUNCTION
============================================================

[Test 1] Small list [1, 2, 3, 4]
Result: 30, Expected: 30
✓ PASSED

[Test 2] Range object range(1, 101)
Sum of squares from 1 to 100: 338350
✓ PASSED

[Test 3] Float numbers [1.5, 2.5, 3.5]
✓ PASSED

[Test 4] Large range - range(1, 1000001)
Sum of squares from 1 to 1,000,000: 333333833333500000
Time taken: 0.2745 seconds
✓ PASSED

============================================================
ALL TESTS COMPLETED SUCCESSFULLY!
============================================================
```