Assignment-10.3

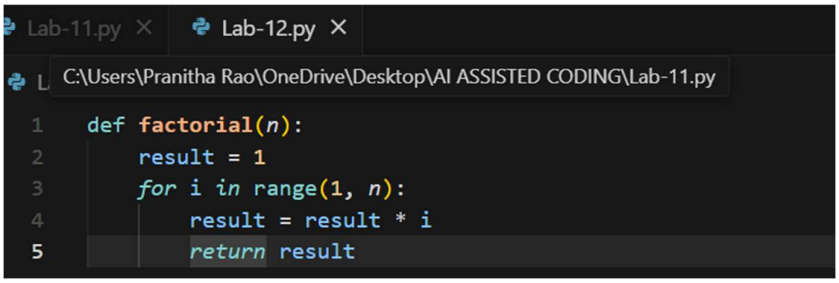
Gujja Pranitha

2303A52171

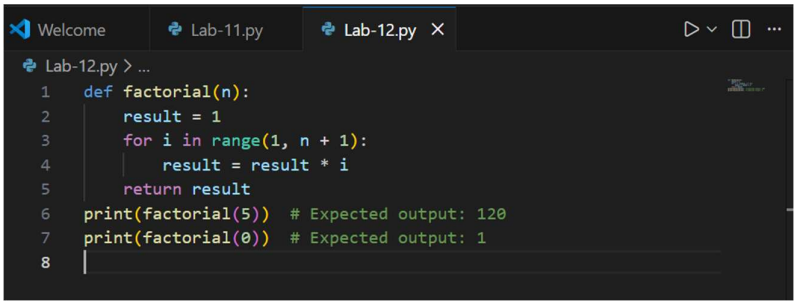
Batch 41

Task-1: AI-Assisted Bug Detection

Provided Code:



Fix Code:



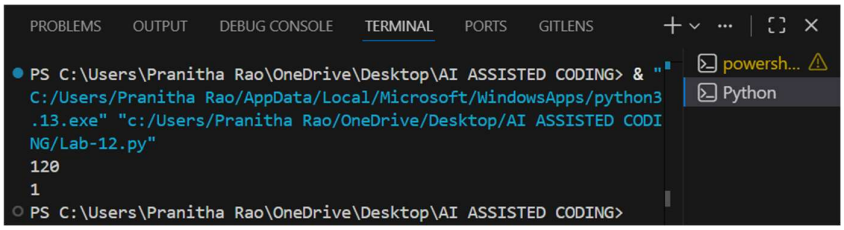
Logical bug in code:

1. return inside loop: Exits after first iteration, always returning 1 instead of computing the full factorial.
2. Wrong range: range(1, n) excludes n itself, so it doesn't multiply the last number needed for the factorial.

Explanation:

The factorial function calculates the product of all positive integers up to a given number n.It initializes a variable result to 1, then iterates from 1 to n (inclusive), multiplying result by each integer i. Finally, it returns the computed factorial value. The test cases demonstrate that factorial(5) correctly returns 120, and factorial(0) returns 1, which is the defined value for 0!.

OUTPUT:

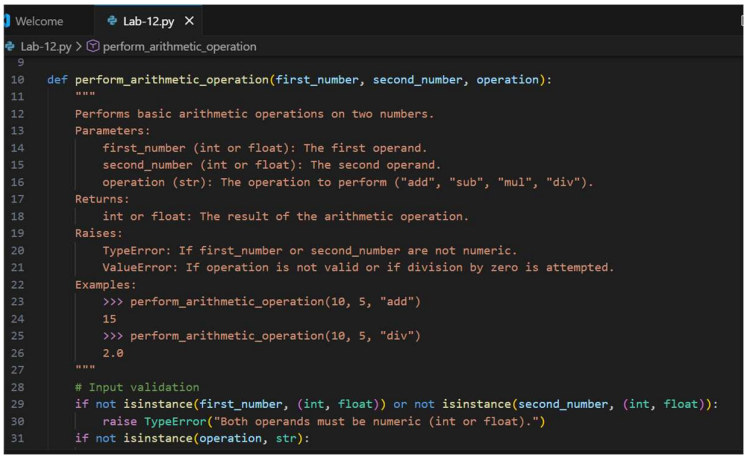


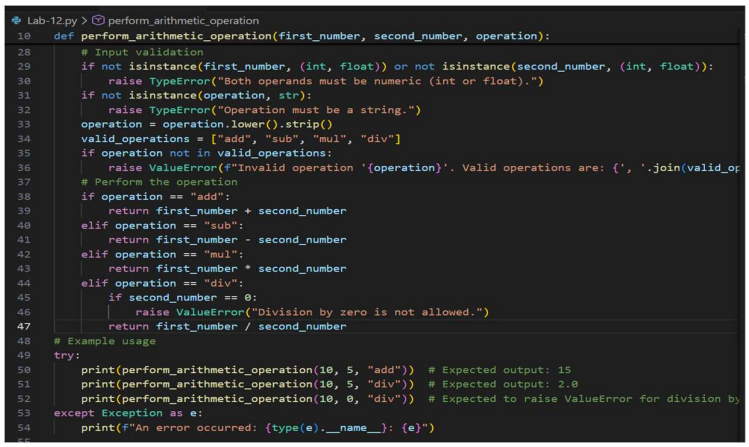
Task-2: Improving Readability & Documentation

Provided Code:

```
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":
```

Fix Code:





Explanation:

The function `perform_arithmetic_operation` takes two numbers and an operation as input. It first validates the inputs to ensure they are of the correct type and that the operation is valid. If the inputs are valid, it performs the specified arithmetic operation and returns the result. If any validation fails, it raises appropriate exceptions with descriptive error messages.

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:/Users/Pranitha Rao/AppData/Local/Micro
soft/WindowsApps/python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-12.py"
15
2.0
An error occurred: ValueError: Division by zero is not allowed.
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

Task-3: Enforcing Coding Standards

Provided Code:

```
47
48  def Checkprime(n):
49      for i in range(2, n):
50          if n % i == 0:
51              return False
52          return True
```

Fix Code:

Welcome | Lab-12.py

Lab-12.py > check_prime

```
47
48  def check_prime(n):
49      """
50      Checks if a number is prime.
51      Parameters:
52          n (int): The number to check.
53      Returns:
54          bool: True if the number is prime, False otherwise.
55      """
56      if n < 2:
57          return False
58
59      for i in range(2, n):
60          if n % i == 0:
61              return False
62      return True
63  # Example usage
64  try:
65      print(check_prime(7))  # Expected output: True
66      print(check_prime(10)) # Expected output: False
67      print(check_prime(-5)) # Expected output: False
68  except Exception as e:
69      print(f"An error occurred: {type(e).__name__}: {e}")
70
```

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:

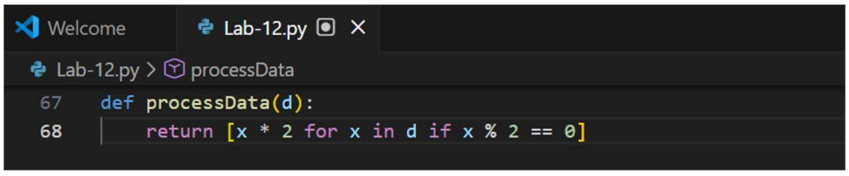
The check_prime function checks if a number n is prime by first checking if it is less than 2 (since prime numbers are defined as greater than 1). If n is less than 2, it returns False. Then, it iterates from 2 to n-1 and checks if n is divisible by any of those numbers. If it finds any divisor, it returns False, indicating that n is not prime. If it completes the loop without finding any divisors, it returns True, indicating that n is prime.

OUTPUT:

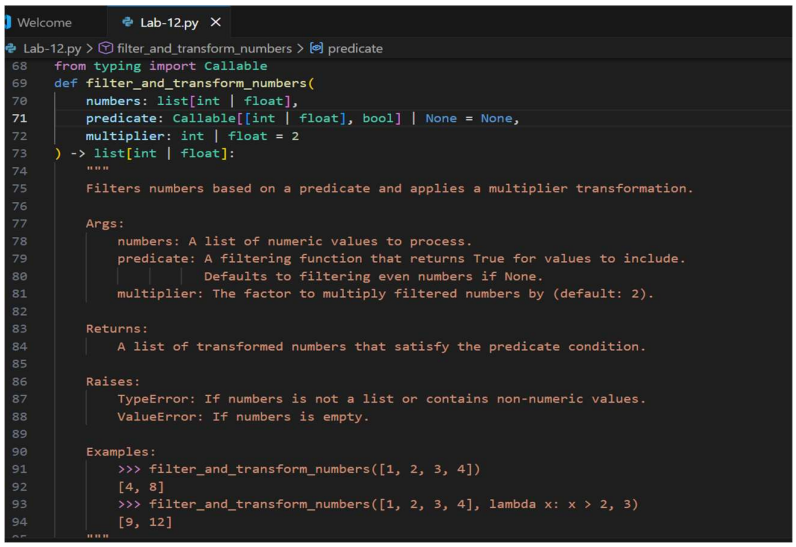

```python
# A PEP8-compliant version of the function,
import math
def is_prime(number):
    if not isinstance(number, int):
        raise TypeError("number must be an integer")
    if number < 2:
        return False
    if number % 2 == 0:
        return number == 2
    limit = math.isqrt(number) +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

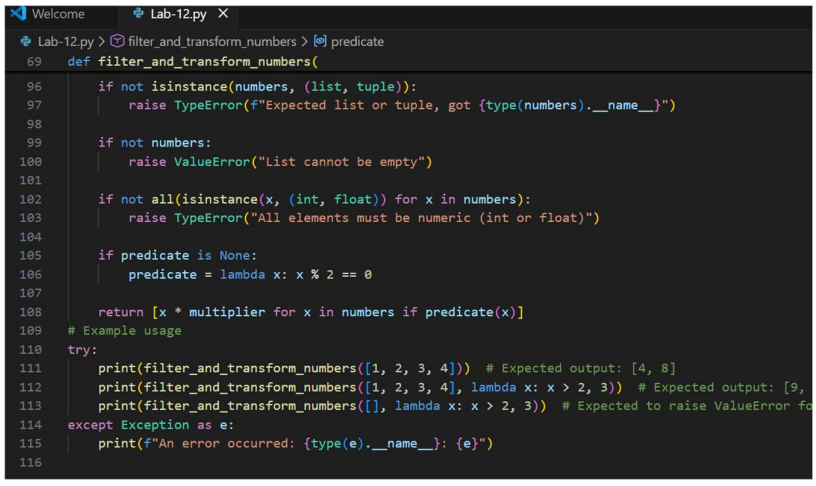
Task-4: AI as a Code Reviewer in Real Projects

Provided Code:



Fix Code:

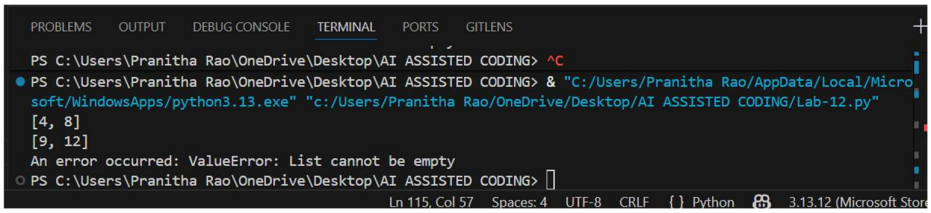




Explanation:

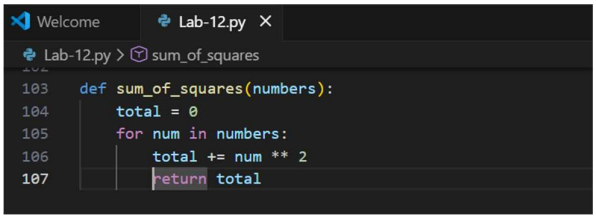
The function filter_and_transform_numbers takes a list of numbers and applies a filtering predicate and a multiplier transformation. It includes input validation to ensure that the input is a list of numeric values and that it is not empty. If the predicate is not provided, it defaults to filtering even numbers. The function returns a new list of transformed numbers that satisfy the predicate condition. The example usage demonstrates how to use the function and handles potential exceptions gracefully.

OUTPUT:

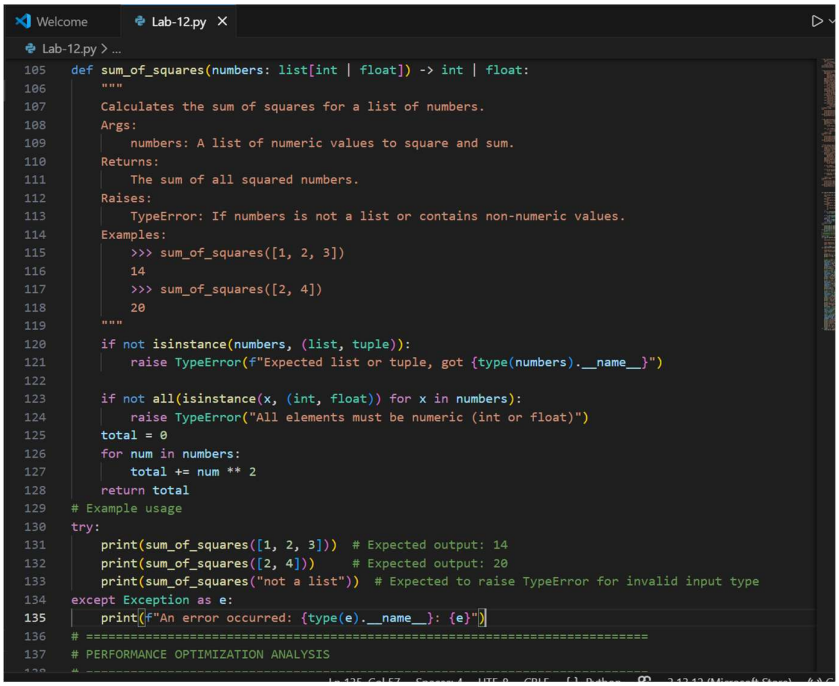


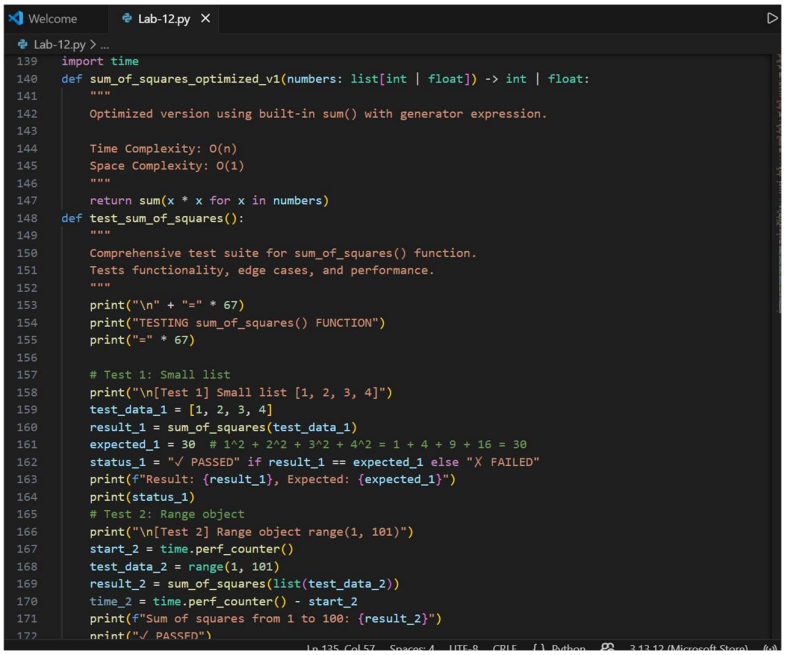
Task-5: AI-Assisted Performance Optimization

Provided Code:



Fix code:

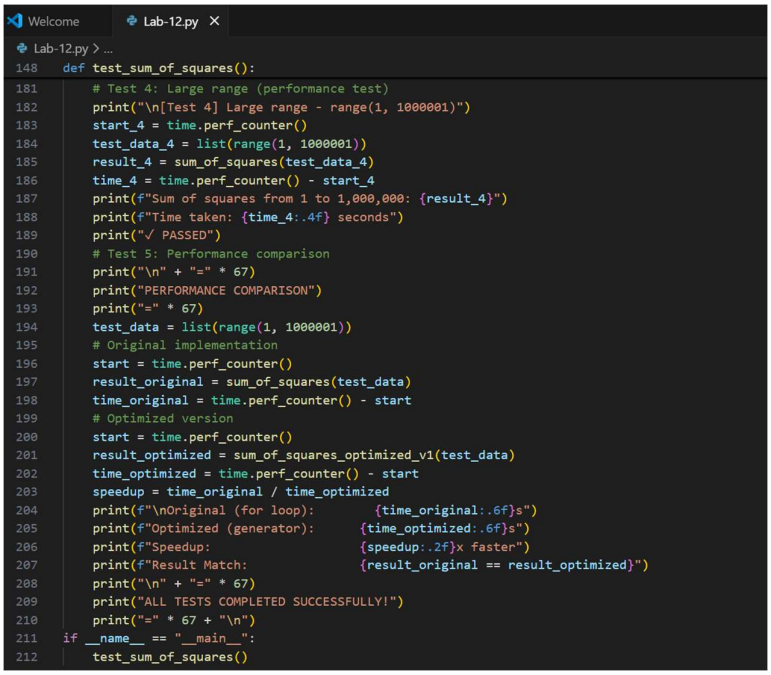




```python
import time
def sum_of_squares_optimized_v1(numbers: list[int | float]) -> int | float:
    """
    Optimized version using built-in sum() with generator expression.

    Time Complexity: O(n)
    Space Complexity: O(1)
    """
    return sum(x * x for x in numbers)
def test_sum_of_squares():
    """
    Comprehensive test suite for sum_of_squares() function.
    Tests functionality, edge cases, and performance.
    """
    print("\n" + "=" * 67)
    print("TESTING sum_of_squares() FUNCTION")
    print("=" * 67)

    # Test 1: Small list
    print("\n[Test 1] Small list [1, 2, 3, 4]")
    test_data_1 = [1, 2, 3, 4]
    result_1 = sum_of_squares(test_data_1)
    expected_1 = 30  # 1^2 + 2^2 + 3^2 + 4^2 = 1 + 4 + 9 + 16 = 30
    status_1 = "✓ PASSED" if result_1 == expected_1 else "✗ FAILED"
    print(f"Result: {result_1}, Expected: {expected_1}")
    print(status_1)
    # Test 2: Range object
    print("\n[Test 2] Range object range(1, 101)")
    start_2 = time.perf_counter()
    test_data_2 = range(1, 101)
    result_2 = sum_of_squares(list(test_data_2))
    time_2 = time.perf_counter() - start_2
    print(f"Sum of squares from 1 to 100: {result_2}")
    print("✓ PASSED")
```



```python
def test_sum_of_squares():
    # Test 4: Large range (performance test)
    print("\n[Test 4] Large range - range(1, 1000001)")
    start_4 = time.perf_counter()
    test_data_4 = list(range(1, 1000001))
    result_4 = sum_of_squares(test_data_4)
    time_4 = time.perf_counter() - start_4
    print(f"Sum of squares from 1 to 1,000,000: {result_4}")
    print(f"Time taken: {time_4:.4f} seconds")
    print("✓ PASSED")
    # Test 5: Performance comparison
    print("\n" + "=" * 67)
    print("PERFORMANCE COMPARISON")
    print("=" * 67)
    test_data = list(range(1, 1000001))
    # Original implementation
    start = time.perf_counter()
    result_original = sum_of_squares(test_data)
    time_original = time.perf_counter() - start
    # Optimized version
    start = time.perf_counter()
    result_optimized = sum_of_squares_optimized_v1(test_data)
    time_optimized = time.perf_counter() - start
    speedup = time_original / time_optimized
    print(f"\nOriginal (for loop):       {time_original:.6f}s")
    print(f"Optimized (generator):     {time_optimized:.6f}s")
    print(f"Speedup:                   {speedup:.2f}x faster")
    print(f"Result Match:              {result_original == result_optimized}")
    print("\n" + "=" * 67)
    print("ALL TESTS COMPLETED SUCCESSFULLY!")
    print("=" * 67 + "\n")
if __name__ == "__main__":
    test_sum_of_squares()
```

Explanation:

The sum_of_squares() function calculates the sum of squares for a list of numbers. It includes input validation to ensure that the input is a list or tuple and that all elements are numeric. The function iterates through each number, squares it, and adds it to a total, which is returned at the end.The sum_of_squares_optimized_v1() function is an optimized version that uses a generator expression with the built-in sum() function, which can be more efficient for large lists.The test_sum_of_squares() function runs a series of tests to validate the correctness and performance of the sum_of_squares() function. It tests small lists, range objects, float numbers, and a large range to ensure

that the function works correctly and efficiently. The performance comparison shows the time taken by both the original and optimized versions, demonstrating the speedup achieved by the optimization.

OUTPUT: