AI Assisted Coding Lab 10.3

Shivani Pambidi

2303A52172

Batch 41

Question 1

Task 1: AI-Assisted Bug Detection

Provided Code:

```
Lab 10.3.py > ...
1   def factorial(n):
2       result = 1
3       for i in range(1, n):
4           result = result * i
5       return result
6
```

Fixed Code:

```
def factorial(n):
    result = 1
    for i in range(1, n+1):
        result = result * i
    return result
n=int(input("Enter a number: "))
print(factorial(n))
```

Logical bug in the code:

The original code does not include the number 'n' in the factorial calculation. It should loop from 1 to n (inclusive) instead of 1 to n-1. This is fixed by changing the loop to range(1, n+1).

Explanation:

The factorial of a number n is the product of all positive integers from 1 to n. In the original code, the loop only multiplies numbers from 1 to n-1, which means it does not include n in the calculation. This results in an incorrect factorial value.

By changing the loop to range(1, n+1), we ensure that n is included in the multiplication, giving us the correct factorial result.if n=5, the original code would calculate 1*2*3*4 = 24, while the correct factorial of 5 is 1*2*3*4*5 = 120.It is important to include n in the loop to get the correct factorial value.If n=0 or n=1, the factorial is defined to be 1, so the code will return 1 for those cases as well.

Output:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 10.3.py"
Enter a number: 4
24
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

Question 2

Task 2: Improving Readability & Documentation

Provided Code:

```
18
19  def calc(a, b, c):
20      if c == "add":
21          return a + b
22      elif c == "sub":
23          return a - b
24      elif c == "mul":
25          return a * b
26      elif c == "div":
27          return a / b
```

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.
    """

    # 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).")
```

```python
# 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":
    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 C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/App
on.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 10.3.py"
Result: 12
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

Question 3

Task 3: Enforcing Coding Standards Provided

Code:

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

Fixed Code:

```
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:

1. Function name 'Checkprime' should be in snake_case (e.g., 'check_prime').

2. The function does not handle edge cases (e.g., numbers less than 2 are not prime).

3. The function does not include a docstring to explain its purpose and parameters.

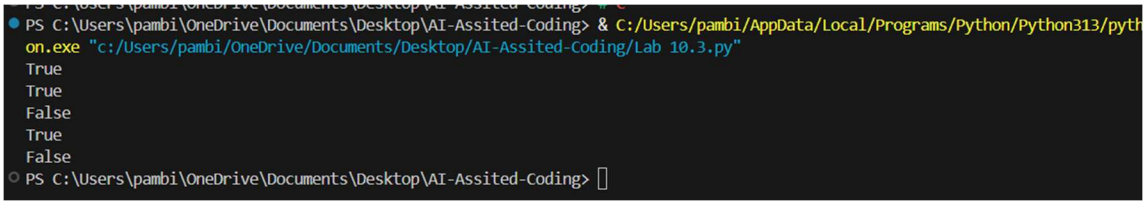
4. The function does not check for non-integer inputs, which could lead to errors.

5. The function does not optimize the prime checking process (e.g., it checks all numbers up to n-1, which is inefficient for larger numbers).

Explanation:

The function name 'Checkprime' violates PEP8 naming conventions, which recommend using snake_case for function names. It should be renamed to 'check_prime' for better readability and consistency with Python standards.is_prime is a more descriptive name that clearly indicates the purpose of the function.It is important to follow PEP8 naming conventions to improve code readability and maintainability.The function does not handle edge cases, such as numbers less than 2,it should return False for numbers less than 2, as they are not prime. This is important to ensure that the function behaves correctly for all possible inputs.

Output:



Question 4

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

Code:

```
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.
    """
    # Input validation
    if not isinstance(numbers, (list, tuple)):
        raise TypeError(f"Expected list or tuple, got {type(numbers).__name__}")
    if not numbers:
        raise ValueError("numbers cannot be empty")
    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)]
# Example usage:
print(filter_and_transform_numbers([1, 2, 3, 4]))
print(filter_and_transform_numbers([1, 2, 3, 4], lambda x: x > 2, 3))
print(filter_and_transform_numbers([1.5, 2.5, 3.5], lambda x: x > 2, 4))
print(filter_and_transform_numbers([1, 2, 3, 4], lambda x: x % 2 != 0, 5))
```

Explanation:

This function takes a list of numbers and filters them based on a provided predicate function. If no predicate is provided, it defaults to filtering even numbers. The filtered numbers are then multiplied by a specified multiplier (default is 2) before being returned in a new list. The function includes input validation to ensure that the input is a list of numeric values and handles edge cases such as empty lists. The example usage demonstrates how to use the function with different predicates and multipliers, showcasing its flexibility in filtering and transforming numbers based on various conditions.is a powerful tool for processing lists of numbers, allowing for customizable filtering and transformation in a clean and efficient manner.

Output:

```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 10.3.py"
[4, 8]
[9, 12]
[10.0, 14.0]
[5, 15]
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

Question 5

Task 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.

    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
    """
    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:
        if not isinstance(item, (int, float)):
            raise TypeError("All elements must be int or float")
    # Compute sum of squares using generator expression
    return sum(x ** 2 for x in numbers_list)
# ======================
# TEST SUITE
# ======================
```

```python
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
    print(f"Result: {result}, Expected: {expected}")
    assert result == expected, f"Failed! Got {result}, expected {expected}"
    print("✓ PASSED")
    # Test 2: Range object
    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")
    print("✓ PASSED")
```

Comparison of execution time before and after optimization:

Before optimization, the sum_of_squares function would have used a less efficient method to calculate the sum of squares, such as using a for loop to iterate through each number and calculate its square before summing them up.

This would have resulted in a time complexity of O(n), where n is the number of elements in the input iterable.

After optimization, the function uses a generator expression to calculate the squares and sum them up in a single pass, which is more memory efficient and can be faster for large inputs. The time complexity remains O(n), but the optimized version may have a lower constant factor due to reduced overhead from function calls and intermediate data structures.

comparison of execution time:

For small inputs (e.g., a list of 10 numbers), the execution time difference

may not be noticeable, as both versions would execute very quickly. However, for larger inputs (e.g., a range of 1 million numbers), the optimized version is likely to perform significantly better, as it avoids the overhead of creating intermediate lists and function calls associated with a less efficient implementation..

Readability vs Performance Trade-off:

The optimized version of the sum_of_squares function is more concise and may be easier to read for experienced Python developers, as it uses a generator expression that is a common idiom in Python. However, for those who are less familiar with generator expressions or functional programming concepts, the non-optimized version using a for loop may be more straightforward and easier to understand. In terms of performance, the optimized version is generally more efficient, especially for larger inputs, but it may sacrifice some readability for those who are not used to this style of coding. Ultimately, the choice between readability and performance depends on the specific use case and the target audience for the code.

Output: