

Name: K.Likitha

Hall Ticket: 2303A52144

Batch: 41

Task 1: AI-Assisted Bug Detection

Prompt:

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

Code:

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
print(factorial(5))
#Identify the logical bug in the code, Explain why the bug occurs (e.g., off-by-one error), Provide a corrected version
# Logical Bug: The function does not handle the case when n is 0 or negative. The factorial of 0 is defined as 1, and factorials for negative numbers are not defined.
# Explanation: The function assumes that n will always be a positive integer. If n is 0, the loop will not execute, and the result will correctly return 1. However, if n is negative, the function will still return 1, which is incorrect since factorials for negative numbers are not defined.
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
print(factorial(5))
```

Output:

```
• (.venv) PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> & C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding\.venv\Scripts\python.exe c:/Users/lilac/OneDrive/Desktop/AI-Assisted-Coding/ai.py
120
120
```

Explanation:

The AI correctly identified the off-by-one error and fixed the loop by changing `range(1, n)` to `range(1, n + 1)`, which produces the correct result for positive integers. However, the initial AI fix did not explicitly handle edge cases such as negative numbers. A manual improvement can include input validation to raise an error for negative values. The case `n = 0` is handled correctly by the loop, as `factorial(0)` returns 1.

Task 2: Improving Readability & Documentation

Prompt:

Rewrite the given Python function to improve readability, add documentation, handle invalid inputs, and safely manage division by zero.

Code:

```

def calc(x, y, op):
    if op == 'add':
        return x + y
    elif op == 'sub':
        return x - y
    elif op == 'mul':
        return x * y
    elif op == 'div':
        if y != 0:
            return x / y
        else:
            return 'Error: Division by zero'
    else:
        return 'Error: Invalid operation'

def perform_calculation(operand1, operand2, operation):
    """
    Performs a basic arithmetic operation between two operands.

    Args:
        operand1 (float or int): The first number for the operation.
        operand2 (float or int): The second number for the operation.
        operation (str): The arithmetic operation to perform. Supported operations are
            'add', 'sub', 'mul', and 'div'.

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

    Raises:
        ValueError: If an invalid operation is specified.
        ZeroDivisionError: If division by zero is attempted.

    Examples:
        >>> perform_calculation(10, 5, 'add')
        15
        >>> perform_calculation(10, 5, 'sub')
        5
        >>> perform_calculation(10, 5, 'mul')
        50
        >>> perform_calculation(10, 2, 'div')
        5.0
        >>> perform_calculation(10, 0, 'div')
        ZeroDivisionError: Division by zero is not allowed.
        >>> perform_calculation(10, 5, 'mod')
        ValueError: Invalid operation specified. Supported operations are add, sub, mul,
div.
    """
    if operation == 'add':
        return operand1 + operand2
    elif operation == 'sub':
        return operand1 - operand2
    elif operation == 'mul':
        return operand1 * operand2
    elif operation == 'div':
        if operand2 == 0:

```

```

        raise ZeroDivisionError('Division by zero is not allowed.')
    return operand1 / operand2
else:
    raise ValueError('Invalid operation specified. Supported operations are add, sub,
mul, div.')

print("Defined the improved 'perform_calculation' function.")
print("Testing original 'calc' function with valid inputs:")
valid_inputs_calc = [
    (10, 5, 'add'),
    (10, 5, 'sub'),
    (10, 5, 'mul'),
    (10, 2, 'div')
]

for x, y, op in valid_inputs_calc:
    result = calc(x, y, op)
    print(f"  calc({x}, {y}, '{op}') = {result}")
print("\nTesting original 'calc' function with invalid inputs:")
invalid_inputs_calc = [
    (10, 0, 'div'),
    (10, 5, 'mod')
]

for x, y, op in invalid_inputs_calc:
    result = calc(x, y, op)
    print(f"  calc({x}, {y}, '{op}') = {result}")

```

Output:

```

• PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> & C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding\.venv\Scripts\python.exe c:/Users/lilac/OneDrive/Desktop/AI-Assisted-Coding/iy.py
Defined the improved 'perform_calculation' function.
Testing original 'calc' function with valid inputs:
  calc(10, 5, 'add') = 15
  calc(10, 5, 'sub') = 5
  calc(10, 5, 'mul') = 50
  calc(10, 2, 'div') = 5.0

Testing original 'calc' function with invalid inputs:
  calc(10, 0, 'div') = Error: Division by zero
  calc(10, 5, 'mod') = Error: Invalid operation
• PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> & C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding\.venv\Scripts\Activate.ps1
○ (.venv) PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> 

```

Explanation:

The original function returns error messages as strings and lacks proper documentation and input validation. The improved version uses descriptive names, a clear docstring, and exception handling to make the code more readable, robust, and maintainable.

Task 3: Enforcing Coding Standards

Prompt:

Identify all PEP8 violations in the given Python function.
 Refactor it to be PEP8-compliant while preserving the original functionality.

Code:

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

# Test with prime numbers
print(f"Is 7 a prime number? {Checkprime(7)}")
print(f"Is 11 a prime number? {Checkprime(11)}")

# Test with non-prime numbers
print(f"Is 4 a prime number? {Checkprime(4)}")
print(f"Is 9 a prime number? {Checkprime(9)}")

def check_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 be checked for primality.

    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if number <= 1:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
    return True

def check_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 be checked for primality.

    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if number <= 1:
        return False
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            return False
```

```

    return True

# Test with prime numbers
print(f"Is 7 a prime number? {check_prime(7)}")
print(f"Is 11 a prime number? {check_prime(11)}")

# Test with non-prime numbers
print(f"Is 4 a prime number? {check_prime(4)}")
print(f"Is 9 a prime number? {check_prime(9)}")

```

Output:

```

(.venv) PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> & C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding\.venv\Scripts\python.exe c:/Users/lilac/OneDrive/Desktop/AI-Assisted-Coding/ko.py
Is 7 a prime number? True
Is 11 a prime number? True
Is 4 a prime number? False
Is 9 a prime number? False
Is 7 a prime number? True
Is 11 a prime number? True
Is 4 a prime number? False
Is 9 a prime number? False
(.venv) PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding>

```

Explanation:

This task verifies whether the given function works correctly and identifies violations of PEP8 coding standards. The code is then refactored using AI suggestions to improve readability and maintain consistent styling without changing functionality.

Task 4: AI as a Code Reviewer in Real Projects

Prompt:

Review the given Python function for readability, edge cases, and reusability.
Refactor it with better naming, input validation, and type hints while preserving functionality.

Code:

```

def processData(numbers):
    processed_numbers = []
    for num in numbers:
        processed_numbers.append(num * 2 + 5)
    return processed_numbers

print("The `processData` function has been defined.")
from typing import List, Union

def apply_linear_transformation(
    numbers: List[Union[int, float]], multiplier: Union[int, float] = 2, addend:
    Union[int, float] = 5
) -> List[Union[int, float]]:
    """
    Applies a linear transformation (num * multiplier + addend) to each numeric element in
    a list.

    Args:
        numbers (List[Union[int, float]]): A list of numbers (integers or floats) to be
        processed.

```

```

        multiplier (Union[int, float], optional): The value to multiply each number by.
Defaults to 2.
        addend (Union[int, float], optional): The value to add to each multiplied number.
Defaults to 5.

Returns:
    List[Union[int, float]]: A new list containing the transformed numbers.

Raises:
    TypeError:
        - If the 'numbers' input is not a list or tuple.
        - If any element in the 'numbers' list is not an int or float.
"""
# Input validation for the 'numbers' parameter
if not isinstance(numbers, (list, tuple)):
    raise TypeError("Input 'numbers' must be a list or a tuple.")

processed_numbers = []
for num in numbers:
    # Input validation for individual elements within the list
    if not isinstance(num, (int, float)):
        raise TypeError(f"All elements in 'numbers' must be numeric (int or float),
but found type {type(num)}.")
    processed_numbers.append(num * multiplier + addend)
return processed_numbers

print("The `apply_linear_transformation` function has been defined with refactored
improvements.")
print("\n--- Testing apply_linear_transformation ---")

# Test Case 1: Valid Inputs (List of Integers, default parameters)
print("\nTest Case 1: Valid Integers (default params)")
input_numbers_int = [1, 2, 3, 4, 5]
try:
    result_int = apply_linear_transformation(input_numbers_int)
    print(f"Input: {input_numbers_int}, Multiplier: 2 (default), Addend: 5 (default) ->
Result: {result_int}")
except TypeError as e:
    print(f"Caught unexpected error: {e}")

# Test Case 2: Valid Inputs (List of Floats, default parameters)
print("\nTest Case 2: Valid Floats (default params)")
input_numbers_float = [1.0, 2.5, 3.7]
try:
    result_float = apply_linear_transformation(input_numbers_float)
    print(f"Input: {input_numbers_float}, Multiplier: 2 (default), Addend: 5 (default) ->
Result: {result_float}")
except TypeError as e:
    print(f"Caught unexpected error: {e}")

# Test Case 3: Custom Parameters
print("\nTest Case 3: Custom Parameters (multiplier=3, addend=10)")
input_numbers_custom = [10, 20]
custom_multiplier = 3

```

```

custom_addend = 10
try:
    result_custom = apply_linear_transformation(input_numbers_custom,
multiplier=custom_multiplier, addend=custom_addend)
    print(f"Input: {input_numbers_custom}, Multiplier: {custom_multiplier}, Addend:
{custom_addend} -> Result: {result_custom}")
except TypeError as e:
    print(f"Caught unexpected error: {e}")

# Test Case 4: Edge Case (Empty List)
print("\nTest Case 4: Empty List")
empty_list = []
try:
    result_empty = apply_linear_transformation(empty_list)
    print(f"Input: {empty_list} -> Result: {result_empty}")
except TypeError as e:
    print(f"Caught unexpected error: {e}")

# Test Case 5: Invalid Input (Non-list/tuple for 'numbers' - int)
print("\nTest Case 5: Invalid Input (numbers is int)")
invalid_input_int = 123
try:
    apply_linear_transformation(invalid_input_int)
except TypeError as e:
    print(f"Input: {invalid_input_int} (int) -> Caught expected error: {e}")

# Test Case 6: Invalid Input (Non-list/tuple for 'numbers' - string)
print("\nTest Case 6: Invalid Input (numbers is string)")
invalid_input_str = "not a list"
try:
    apply_linear_transformation(invalid_input_str)
except TypeError as e:
    print(f"Input: '{invalid_input_str}' (string) -> Caught expected error: {e}")

# Test Case 7: Invalid Input (List with non-numeric elements - string)
print("\nTest Case 7: Invalid Input (list with string element)")
list_with_str = [1, 2, 'three', 4]
try:
    apply_linear_transformation(list_with_str)
except TypeError as e:
    print(f"Input: {list_with_str} -> Caught expected error: {e}")

# Test Case 8: Invalid Input (List with non-numeric elements - None)
print("\nTest Case 8: Invalid Input (list with None element)")
list_with_none = [1, None, 3]
try:
    apply_linear_transformation(list_with_none)
except TypeError as e:
    print(f"Input: {list_with_none} -> Caught expected error: {e}")

```

Output:

```
PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> python3 main.py
The `processData` function has been defined.
The `apply_linear_transformation` function has been defined with refactored improvements.

--- Testing apply_linear_transformation ---

Test Case 1: Valid Integers (default params)
Input: [1, 2, 3, 4, 5], Multiplier: 2 (default), Addend: 5 (default) -> Result: [7, 9, 11, 13, 15]

Test Case 2: Valid Floats (default params)
Input: [1.0, 2.5, 3.7], Multiplier: 2 (default), Addend: 5 (default) -> Result: [7.0, 10.0, 12.4]

Test Case 3: Custom Parameters (multiplier=3, addend=10)
Input: [10, 20], Multiplier: 3, Addend: 10 -> Result: [40, 70]

Test Case 4: Empty List
Input: [] -> Result: []

Test Case 5: Invalid Input (numbers is int)
Input: 123 (int) -> Caught expected error: Input 'numbers' must be a list or a tuple.

Test Case 6: Invalid Input (numbers is string)
Input: 'not a list' (string) -> Caught expected error: Input 'numbers' must be a list or a tuple.

Test Case 7: Invalid Input (list with string element)
Input: [1, 2, 'three', 4] -> Caught expected error: All elements in 'numbers' must be numeric (int or float), but found type <class 'str'>.

Test Case 8: Invalid Input (list with None element)
Input: [1, None, 3] -> Caught expected error: All elements in 'numbers' must be numeric (int or float), but found type <class 'NoneType'>.
PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding>
```

Explanation:

This task evaluates the original function for clarity, robustness, and handling of edge cases. AI suggestions are used to refactor the code with improved naming, validation, and generalization for real-world use.

Task 5: AI-Assisted Performance Optimization

Prompt:

Analyze the performance of the given function and suggest optimizations. Rewrite the function to improve execution speed while preserving correctness.

Code:

```
import time
import numpy as np

def sum_of_squares(numbers):
    """
    Calculates the sum of squares of numbers in a list or iterable.
    """
    return sum(n ** 2 for n in numbers)

def sum_of_squares_optimized(numbers):
    """
    Calculates the sum of squares of numbers using NumPy for optimization.
    """
    np_array = np.array(numbers)
    return np.sum(np_array ** 2)
```

```
# Create a large list of numbers
large_numbers_dataset = list(range(1_000_000))

# Measure execution time of original function
start = time.perf_counter()
sum_of_squares(large_numbers_dataset)
end = time.perf_counter()
print("Original function time:", end - start, "seconds")

# Measure execution time of optimized function
start = time.perf_counter()
sum_of_squares_optimized(large_numbers_dataset)
end = time.perf_counter()
print("Optimized function time:", end - start, "seconds")
```

Output:

```
Original function time: 0.07380910002393648 seconds
Optimized function time: 0.038818700006231666 seconds
(.venv) PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding>
```

Explanation:

This task evaluates the performance of a function on large datasets and identifies inefficiencies. AI is used to optimize the implementation and compare performance trade-offs.