AI ASSISTANT CODING - ASSIGNMENT-10.3

Name: Valaboju Harshavardhanu

Roll No: 2303A52354 **Batch:** 41

Task 1: AI-Assisted Bug Detection

Prompt:

identify the the logical bug in the code of 10.3.py

Code:

```
10.3.py > ...
1   def factorial(n):
2       result = 1
3       for i in range(1, n + 1):
4           result = result * i
5       return result
6   print(factorial(5))
7
```

Output:

```
PS C:\Use> & C:\Users\harsh\Ap
ktop/AI Assist lab/10.3.py"
120
```

Justification:

In the code given in the scenario has the bug that is in the for loop the range is upto n so the it will run until n-1 this results the incorrect output.

The the AI fixed the bug bug by adding the n+1 in place of n in the for loop.

The AI generated code and the manual code i have fixed is the similar to each other.

The AI missed the edge case that is the if we perform the function with the negative numbers it has to get the output not defined but it is showing the -1.

Task 2: Improving Readability &Documentation

Prompt:

rewrite the function with descriptive function and parameter names and a complete docstring which includes description, parameters, return value, examples and exception handling for division by zero and consideration of input validation.

Scenario 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":
```

AI Corrected Code:

```python
def perform_arithmetic_operation(first_operand, second_operand, operation_type):
    """
    Performs basic arithmetic operations on two numbers.

    This function takes two numeric operands and an operation type, then returns
    the result of the specified arithmetic operation. It validates inputs and
    handles division by zero errors gracefully.

    Parameters:
        first_operand (int or float): The first number for the operation.
        second_operand (int or float): The second number for the operation.
        operation_type (str): The type of operation to perform. Accepted values:
            - "add": Addition
            - "sub": Subtraction
            - "mul": Multiplication
            - "div": Division

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

    Raises:
        TypeError: If operands are not numeric types (int or float) or operation_type is not a string.
        ValueError: If operation_type is not one of the supported operations.
        ZeroDivisionError: If division by zero is attempted.

    Examples:
        >>> perform_arithmetic_operation(10, 5, "add")
        15
        >>> perform_arithmetic_operation(10, 5, "sub")
        5
        >>> perform_arithmetic_operation(10, 5, "mul")
        50
        >>> perform_arithmetic_operation(10, 5, "div")
        2.0
        >>> perform_arithmetic_operation(10, 0, "div")
        Raises ZeroDivisionError
    """

    # Input validation for first operand
    if not isinstance(first_operand, (int, float)):
        raise TypeError(f"First operand must be numeric (int or float), got {type(first_operand).__name__}.")
    if isinstance(first_operand, bool):
        raise TypeError("First operand cannot be a boolean value.")

    # Input validation for second operand
    if not isinstance(second_operand, (int, float)):
        raise TypeError(f"Second operand must be numeric (int or float), got {type(second_operand).__name__}.")
    if isinstance(second_operand, bool):
        raise TypeError("Second operand cannot be a boolean value.")
```

```python
    # Input validation for operation type
    if not isinstance(operation_type, str):
        raise TypeError(f"Operation type must be a string, got {type(operation_type).__name__}.")

    operation_type = operation_type.strip().lower()

    if not operation_type:
        raise ValueError("Operation type cannot be empty or whitespace-only.")

    valid_operations = {"add", "sub", "mul", "div"}
    if operation_type not in valid_operations:
        raise ValueError(f"Operation '{operation_type}' is not supported. Use one of: {', '.join(sorted(valid_operati

    # Perform the requested operation
    if operation_type == "add":
        return first_operand + second_operand
    elif operation_type == "sub":
        return first_operand - second_operand
    elif operation_type == "mul":
        return first_operand * second_operand
    elif operation_type == "div":
        if second_operand == 0:
            raise ZeroDivisionError("Cannot divide by zero. The second operand must be non-zero.")
        return first_operand / second_operand
```

```python
# Example usage with error handling
if __name__ == "__main__":
    test_cases = [
        (10, 5, "add"),           # Valid: 15
        (10, 5, "div"),           # Valid: 2.0
        (10, 0, "div"),           # Error: Division by zero
        (10, 5, "invalid"),       # Error: Invalid operation
        ("10", 5, "add"),         # Error: Invalid operand type
        (10, 5, 123),             # Error: Invalid operation type
        (True, 5, "add"),         # Error: Boolean not allowed
    ]

    for first, second, operation in test_cases:
        try:
            result = perform_arithmetic_operation(first, second, operation)
            print(f"Result: {first} {operation} {second} = {result}")
        except (TypeError, ValueError, ZeroDivisionError) as error:
            print(f"Error: {error}")
```

Output:

```
sers/harsh/OneDrive/Desktop/AI Assist lab/10.3.py"
Result: 10 add 5 = 15
Result: 10 div 5 = 2.0
Error: Cannot divide by zero. The second operand must be non-zero.
Error: Operation 'invalid' is not supported. Use one of: add, div, mul, sub
Error: First operand must be numeric (int or float), got str.
Error: Operation type must be a string, got int.
Error: First operand cannot be a boolean value.
PS C:\Users\harsh\OneDrive\Desktop\AI Assist lab>
```

Justification:

In the above task, using the AI we have improved the code for better readability using the docstrings and the documentation and we have successfully handled the Exceptions and solved it and the AI Improved code will also handled the input validation.

I have compared the Original code and the AI-Improved code it clearly shows the difference between both the codes which the clear understanding of the AI given code using the Documentation.

The both valid and invalid inputs are working correctly with the AI given code it runs correctly.

Task 3: Enforcing Coding Standards

Prompt:

List all the PEP8 violations present in the given code and refactor the code

Code:

10.3.py > ...

```python
def check_prime(n: int) -> bool:
    """Return True if n is a prime number, otherwise False."""
    if n <= 1:
        return False

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

    return True


# Function calls
print(check_prime(7))   # True
print(check_prime(10))  # False
print(check_prime(1))   # False
```

Output:

```
PS C:\Users
sers/harsh/
True
False
False
```

Justification:

PEP8 Violations :

1. Function name not in snake_case → Checkprime should be check_prime
2. Missing blank line before function definition
3. No docstring provided
4. Edge case not handled (n <= 1)

Automated AI reviews can significantly streamline code reviews in large teams by instantly detecting style violations, logical bugs, and performance issues. AI tools enforce consistent coding standards like PEP8 without human bias and reduce reviewer workload. This allows developers to focus on architecture and design decisions instead of minor formatting corrections. As a result, teams achieve faster review cycles, cleaner codebases, and improved collaboration.

Task 4: AI as a Code Reviewer in Real Projects

Prompt:

Refactor and improve the code in file by renaming for clarity purposes, and suggest generalization.

Code:

```python
from typing import Iterable, Callable, List, TypeVar
T = TypeVar("T")
U = TypeVar("U")
def transform_and_filter(
    items: Iterable[T], predicate: Callable[[T], bool], transformer: Callable[[T], U]
) -> List[U]:
    """
    Return a list of transformed items for which `predicate` is True.

    Args:
        items: An iterable of input values.
        predicate: A function that returns True for items to keep.
        transformer: A function that maps an item to the desired output.

    Returns:
        A list containing `transformer(item)` for every `item` in `items`
        where `predicate(item)` is True.

    Example:
        >>> transform_and_filter([1,2,3,4], lambda x: x%2==0, lambda x: x*2)
        [4, 8]
    """
    if not callable(predicate) or not callable(transformer):
        raise TypeError("predicate and transformer must be callable")

    return [transformer(x) for x in items if predicate(x)]


def double_even_numbers(numbers: Iterable[int]) -> List[int]:
    """
    Return a list with even integers doubled from the provided iterable.

    This is a clarity-focused wrapper around `transform_and_filter` for the
    common operation of filtering evens and doubling them.

    Args:
        numbers: An iterable of integers.

    Returns:
        A list of integers where each even input value has been multiplied by 2.

    Raises:
        TypeError: If `numbers` is not iterable of ints.

    Example:
        >>> double_even_numbers([1,2,3,4])
        [4, 8]
    """
    return transform_and_filter(numbers, lambda x: isinstance(x, int) and x % 2 == 0, lambda x: x * 2)


if __name__ == "__main__":
    sample = [1, 2, 3, 4, 5, 6]
    print("Original:", sample)
    print("Doubled evens:", double_even_numbers(sample))
```

Output:

```
Original: [1, 2, 3, 4, 5, 6]
Doubled evens: [4, 8, 12]
PS C:\Users\VIGNANAND RAO\OneDrive\Desktop\AI_coding> [
```

Justification:

Suggestions for generalization:

- Accept a max_items parameter for streaming/large iterables.
- Return an iterator (use yield) instead of a list for memory efficiency.
- Provide common predicates/transformers (e.g., is_even, scale_by(n)) in a small utilities module.
- Allow async predicates/transformers for IO-bound transforms.

AI should act as an assistant rather than a standalone reviewer. While it is efficient at detecting errors and enforcing standards, human reviewers provide critical thinking, context understanding, and design judgment. Combining AI speed with human insight leads to better and more reliable code reviews.

Task 5: AI-Assisted Performance Optimization

Prompt:

Analyze the given code, give time complexity, and then optimize the code and check for input range 1000000.

Code:

```python
def sum_of_squares(numbers):
    return sum(num ** 2 for num in numbers)
# Large test input
numbers = range(1000000)
print("Sum of squares:", sum_of_squares(numbers))
```

Output:

```
Sum of squares: 333332833333500000
PS C:\Users\VIGNANAND RAO\OneDrive\De
```

Justification:

Both the codes before and after optimization, it gives the same time complexity of O(n) as we must iterate till n number of times for getting the answer. We are optimizing the given code by using the in-built functions which helps to solve the problem easily.

Readable code helps humans understand, maintain, and debug programs easily. Highly optimized code may run faster but can become complex and harder to understand. So, it's better to keep code simple and only optimize when performance is really needed.