ASSIGNMENT-10.3

SINDHU KATAKAM

2303A52481

BATCH-41

Problem Statement 1:Task1- AI-Assisted Bug Detection

Provided code:

```
def factorial(n):
    result = 1
    for i in range(1, n):
        result = result * i
    return result
print(factorial(5))
```

Corrected code:

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
print(factorial(5))
```

Logical bug in the code:

The logic bug in the code is in the for loop range. The loop should include n, but currently it goes from 1 to n-1 (because range(1, n) excludes n). This means factorial(5) returns 24 instead of 120. The correct loop should be range(1, n+1).

Brief comparision of did AI miss any edge cases

The AI did not miss any edge cases in this scenario. The code calculates the factorial for positive integers, and the identified bug (incorrect range) affects all inputs greater than 1. However, the code does not handle edge cases such as negative numbers or non-integer inputs, which could cause unexpected behavior or errors. If those cases are important, additional input validation would be needed.

Output:

```
PS D:\AI_ASSIT_CODING> & C:/
T_CODING/Assignment-10.3.py
Factorial of 5 is 120
```

Problem Statement 2: 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":
        return a / b
```

Modified code using AI

```
def calculate(num1, num2, operation):
    """
    Perform a basic arithmetic operation on two numbers.

    Parameters:
        num1 (float or int): The first operand.
        num2 (float or int): The second operand.
        operation (str): The operation to perform.
                         Supported values: 'add', 'sub', 'mul', 'div'.

    Returns:
        float: The result of the arithmetic operation.

    Raises:
        ValueError: If an unsupported operation is provided.
        ZeroDivisionError: If division by zero is attempted.

    Examples:
        >>> calculate(10, 5, 'add')
        15
        >>> calculate(10, 5, 'div')
        2.0
        >>> calculate(10, 0, 'div')
        Traceback (most recent call last):
```

```python
        ZeroDivisionError: Cannot divide by zero.
        >>> calculate(10, 5, 'mod')
        Traceback (most recent call last):
            ...
        ValueError: Unsupported operation: mod
    """
    if operation == "add":
        return num1 + num2
    elif operation == "sub":
        return num1 - num2
    elif operation == "mul":
        return num1 * num2
    elif operation == "div":
        if num2 == 0:
            raise ZeroDivisionError("Cannot divide by zero.")
        return num1 / num2
    else:
        raise ValueError(f"Unsupported operation: {operation}")

# Valid inputs
print(calculate(10, 5, "add"))  # 15
print(calculate(10, 5, "sub"))  # 5
print(calculate(10, 5, "mul"))  # 50
print(calculate(10, 5, "div"))  # 2.0
```

```python
# Invalid inputs
try:
    print(calculate(10, 0, "div"))
except Exception as e:
    print(e)  # Cannot divide by zero.

try:
    print(calculate(10, 5, "mod"))
except Exception as e:
    print(e)  # Unsupported operation: mod
```

Comparision between original and AI generated code

The original function had short and ambiguous names for its parameters and did not describe what the function did. The AI-assisted version has descriptive names such as num1, num2, and operation, so it is clear what each part of the function does. The version also has a very detailed docstring that describes how the function works, what to expect from the input and output, and even provides examples. The version checks for errors, such as dividing by zero or performing an illegal operation, and provides very helpful messages. The AI version is much easier to read, understand, and use safely.

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users,
Assignment-10.3.py
15
5
50
2.0
Cannot divide by zero.
Unsupported operation: mod
PS D:\AI_ASSIT_CODING>
```

Problem Statement 3: Enforcing Coding Standards

Provided code:

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

Modified code:

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

AI generated PEP8 violations:

- Function name should be lowercase with underscores (checkpoint is okay, but is_prime is clearer).
- Missing two blank lines before function definition.
- No docstring.
- Parameter n is not descriptive.
- Indentation is correct, but spacing around operators can be improved.
- No space after commas in function calls

Comparision between original and AI generated code

The original function had a name that was not as clear and did not follow Python style guidelines. The improved version by the AI has a better name, check_prime, and follows proper spacing and indentation guidelines, making the code look clean and professional. Both functions do the same thing, but the improved version is easier to read and use. Following proper naming and formatting guidelines helps everyone quickly understand what the function does. The AI version is more organized and friendly to use.

Output:

```
PS D:\AI_ASSIT_CODING>
Assignment-10.3.py
True
False
```

```python
#A PEP8-compliant version of the function, e.g.
def is_prime(number):
    if number < 2:
        return False
    for i in range(2, number):
        if number % i == 0:
            return False
    return True
```

Problem Statement 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]
```

Modified code using AI

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

def double_even_numbers(numbers: Iterable[Union[int, float]]) -> List[Union[int, float]]:
    result = []
    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError(f"All elements must be int or float, got {type(num)}")
        if num % 2 == 0:
            result.append(num * 2)
    return result
```

Comparision between original and AI generated code

The original code was simple and just doubled the even numbers in a list, but it didn't check if the input was correct or explain how it worked. The AI-generated code uses clearer names, checks that all items are numbers, and gives an error if something is wrong, which helps prevent mistakes. It also adds type hints and a better structure, making it easier to read and understand. The new version is more reliable and professional, so it's better for both learning and real projects. Overall, the AI code is safer and easier to use, especially when working with different kinds of data.

Output:

```
PS D:\AI ASSIT CODING> &
Assignment-10.3.py
[4, 8, 12]
[36, 44, 8, 104]
```

Problem Statement 5: — AI-Assisted Performance Optimization

Provided code **AI generated code**

```python
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
# Example usage
numbers = [1, 2, 3, 4]
result = sum_of_squares(numbers)
print("The sum of squares is:", result)
```

```python
import numpy as np

def sum_of_squares(numbers):
    arr = np.asarray(numbers)
    return np.sum(arr ** 2)

# Example usage
numbers = [1, 2, 3, 4]
result = sum_of_squares(numbers)
print("The sum of squares is:", result)
```

```
The sum of squares is: 30
```

Execution time before and after optimization

Before optimization, the function uses a Python loop, which is slow for very large lists (like 1,000,000 elements) because each operation is handled one at a time by the Python interpreter. After optimization with NumPy, the function uses vectorized operations, which are much faster because they run in compiled code and process many elements at once. For a list of 1,000,000 numbers, the NumPy version can be dozens of times faster than the pure Python version.

Trade-offs between readability and performance

The original version is very readable and easy to understand for beginners, as it uses basic Python syntax. The optimized NumPy version is slightly less readable for those unfamiliar with NumPy, but it offers huge performance benefits for large datasets. In summary, the pure Python version is better for small lists and learning, while the NumPy version is best for large data and high performance, even if it requires learning an extra library.

Comparision between original and AI generated code

The original code uses a simple for loop to add up the squares of each number in the list, which is easy to read and understand for beginners. However, it can be slow when working with very large lists because it processes each number one by one. The AI-generated code uses the NumPy library to do the same calculation much faster by handling all the numbers at once with built-in functions. This makes the code run much quicker for big lists, but it might look confusing if you haven't used NumPy before. Overall, the AI version is better for speed with large data, while the original is easier for learning and small tasks.

Output:

```
The sum of squares is: 30
```

```
def sum_of_squares_optimized(numbers):
    return sum(x * x for x in numbers)
```