

Problem Statement 1: AI-Assisted Bug Detection

Original code:

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

Test run:

```
>>> factorial(5)
24
```

AI prompt used: "Identify the logical bug in this factorial function and provide a corrected version."

AI response summary: Off-by-one error. range(1, n) goes to n-1, so misses n.

Corrected:

```
def factorial(n):
    if n < 0:
        raise ValueError("Factorial not defined for negative numbers")
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

Test corrected:

```
>>> factorial(5)
120
>>> factorial(0)
1
>>> factorial(-1)
ValueError: Factorial not defined for negative numbers
```

Comparison: AI caught edge cases like negative and zero; my manual fix missed zero initially.

Problem Statement 2: Improving Readability & Documentation

Original 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 prompt: "Critique readability and rewrite with docstring, validation, error handling."

AI improved version:

```
def calculate(a: float, b: float, operation: str) -> float:
    """
    Perform basic arithmetic operations on two numbers.

    Args:
        a (float): First number.
        b (float): Second number.
        operation (str): Operation to perform ('add', 'sub', 'mul', 'div').

    Returns:
        float: Result of the operation.

    Raises:
        ValueError: If invalid operation or division by zero.
        TypeError: If a or b not numbers.

    Examples:
        >>> calculate(10, 5, 'add')
        15.0
        """
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
```

```

        raise TypeError("Inputs must be numbers")

operations = {
    'add': a + b,
    'sub': a - b,
    'mul': a * b,
    'div': a / b if b != 0 else _raise_div_zero()
}

if operation not in operations:
    raise ValueError("Invalid operation")

return operations[operation]

def _raise_div_zero():
    raise ValueError("Division by zero")

```

Tests:

```

>>> calculate(10, 5, 'add')
15.0
>>> calculate(10, 0, 'div')
ValueError: Division by zero
>>> calculate('a', 5, 'add')
TypeError: Inputs must be numbers

```

Problem Statement 3: Enforcing Coding Standards

Original:

```

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

```

PEP8 compliant:

```
def is_prime(n):
    """Check if n is prime."""
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Tests:

```
>>> is_prime(17)
True
>>> is_prime(4)
False
>>> is_prime(1)
False
```

Problem Statement 4: AI as Code Reviewer

Original:

```
def processData(d):
    return [x * 2 for x in d if x % 2 == 0]
```

Manual review: Poor naming (processData vague), no types/validation, assumes list/ints, no doc.

AI review prompt: "Review for readability, add validation/type hints, generalize."

Improved:

```
from typing import List, Union, Callable

def double_even_numbers(numbers: List[Union[int, float]]) -> List[Union[int, float]]:
    """
    Double even numbers in a list.
    """

Args:
```

```

numbers: List of numbers.

Returns:
    List of doubled even numbers.

Raises:
    TypeError: If not list.

"""

if not isinstance(numbers, list):
    raise TypeError("Input must be a list")
return [num * 2 for num in numbers
        if isinstance(num, (int, float)) and num % 2 == 0]

```

Tests:

```

>>> double_even_numbers([1, 2, 3, 4])
[4, 8]
>>> double_even_numbers("not list")
TypeError: Input must be a list

```

Problem Statement 5: Performance Optimization

Original:

```

def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total

```

Test large input:

```

import time
nums = list(range(1000000))
start = time.time()
sum_of_squares(nums)
print(time.time() - start)
# 0.145s (approx)

```

AI analysis: $O(n)$ fine, but loop slower than `sum(genexp)`.

Optimized:

```
def sum_of_squares(numbers):
    return sum(x ** 2 for x in numbers)
```

Test:

```
start = time.time()
sum_of_squares(nums)
print(time.time() - start)
# 0.035s (approx)
```

Faster by 4x using built-in `sum`. Trade-off: Slightly less readable for beginners, but standard idiom.

Execution Times Comparison

Function	Time (s) for 1M nums
Original loop	0.145
Optimized sum	0.035