

AI Assisted Coding — Lab Assignment 10.3

Code Review and Quality: Using AI to Improve Code Quality and Readability

Problem Statement 1: AI-Assisted Bug Detection

Task Description

A junior developer wrote a Python function to calculate factorials. The function contains a logical bug. We use AI to identify, explain, and correct the bug.

Original Buggy Code

```
def factorial(n):
    result = 1
    for i in range(1, n):      # Bug: should be range(1, n+1)
        result = result * i
    return result
```

Prompt Used

Prompt Sent to AI Tool

"The following Python function is meant to calculate the factorial of a number n. When I run factorial(5), it returns 24 instead of 120. Please identify the logical bug, explain why it occurs, and provide a corrected version. Also check for edge cases like negative numbers and zero."

AI-Generated Output / Explanation

The AI correctly identified the off-by-one error in the range. Python's range(1, n) goes from 1 to n-1 (exclusive), so for n=5, it multiplies $1 \times 2 \times 3 \times 4 = 24$ instead of $1 \times 2 \times 3 \times 4 \times 5 = 120$. The fix is to use range(1, n+1) so the loop includes n itself.

Corrected Code

```
def factorial(n):
    """Calculate the factorial of a non-negative integer n."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0:
        return 1
    result = 1
    for i in range(1, n + 1):    # Fixed: n+1 ensures n is included
        result = result * i
    return result

# Test
print(factorial(5))    # Output: 120
print(factorial(0))    # Output: 1
```

Output

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

Explanation

The original code had a classic off-by-one error — the loop's range excluded the number n itself. Imagine you're counting steps from 1 to 5 but you stop at 4. You'd miss the last step! The AI caught this immediately, explained the logic clearly, and also added edge case handling for 0 (which is a valid factorial = 1) and negative numbers (which are undefined). The fix is small — just adding +1 to the range end — but it's critical for correctness.

Problem Statement 2: Improving Readability & Documentation

Task Description

A poorly written but functional calculator function needs to be improved for readability, documentation, and robustness. AI is used to critique and rewrite it.

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":
        return a / b
```

Prompt Used

Prompt Sent to AI Tool

"Please critique the following Python function for readability, parameter naming, and lack of documentation. Then rewrite it with: descriptive function and parameter names, a complete docstring with description/parameters/returns/examples, exception handling for division by zero, and input validation."

AI-Generated Critique

The AI pointed out that: the function name `calc()` is too vague; parameters `a`, `b`, `c` give no hint of their purpose; there is no docstring; there is no handling for division by zero; non-string operation inputs are silently ignored; and the function has no return for unrecognised operations.

Improved Code (AI-Generated)

```
def arithmetic_operation(num1: float, num2: float, operation: str) -> float:
    """
    Perform a basic arithmetic operation on two numbers.

    Parameters:
        num1 (float): The first operand.
        num2 (float): The second operand.
        operation (str): One of 'add', 'sub', 'mul', 'div'.

    Returns:
        float: Result of the arithmetic operation.

    Raises:
        ValueError: If operation is not one of the supported strings.
        ZeroDivisionError: If operation is 'div' and num2 is zero.

    Examples:
        >>> arithmetic_operation(10, 5, 'add')
```

```

15
>>> arithmetic_operation(10, 0, 'div')
ZeroDivisionError
"""
if not isinstance(operation, str):
    raise TypeError("Operation must be a string: 'add', 'sub', 'mul',
'div'.")

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}'. Use 'add',
'sub', 'mul', or 'div'.")

```

Output

```

>>> arithmetic_operation(10, 5, 'add') → 15 >>> arithmetic_operation(10, 2, 'div') →
5.0 >>> arithmetic_operation(10, 0, 'div') → ZeroDivisionError: Cannot divide by zero.
>>> arithmetic_operation(10, 5, 'mod') → ValueError: Unsupported operation

```

Explanation

The original function was like a vending machine with buttons labelled A, B, C — it works, but you have no idea what each button does. The AI rewrote it with clear names (`arithmetic_operation`, `num1`, `num2`, `operation`), a full docstring so any developer can understand it instantly, and proper error handling so the function fails gracefully instead of crashing silently. Think of it as upgrading from a sticky-note manual to a professional user guide.

Problem Statement 3: Enforcing Coding Standards (PEP8)

Task Description

A developer's function violates PEP8 standards. We use AI to list all violations, refactor the code, and verify that functionality is preserved.

Original Code

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

Prompt Used

Prompt Sent to AI Tool

"The following Python code needs to be PEP8 compliant. Please list all PEP8 violations, refactor the code (fix function name, spacing, indentation, naming conventions), and ensure the logic is preserved and correct."

PEP8 Violations Identified by AI

1. Function name `Checkprime` uses CapWords (PascalCase) — PEP8 requires `snake_case` for functions.
2. No docstring — PEP8 recommends all public functions have a docstring.
3. No blank line after the function definition for readability.
4. No type hints — modern PEP8-compliant code uses type annotations.
5. Logic issue: `range(2, n)` means $n=2$ (smallest prime) returns `True` but $n=1$ also returns `True` (1 is not prime).

Refactored PEP8-Compliant Code

```
def check_prime(n: int) -> bool:
    """
    Check if a given integer is a prime number.

    Args:
        n (int): The number to check.

    Returns:
        bool: True if n is prime, False otherwise.
    """
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1): # Optimised: only check up to
        sqrt(n)
        if n % i == 0:
            return False
    return True
```

```
# Test
print(check_prime(2))      # True
print(check_prime(7))      # True
print(check_prime(10))     # False
print(check_prime(1))      # False
```

Output

```
>>> check_prime(2) → True >>> check_prime(7) → True >>> check_prime(10) →
False >>> check_prime(1) → False
```

Note on AI Reviews in Large Teams

AI-powered code review tools can dramatically speed up the review process in large teams. Instead of a senior developer spending time catching style violations, an AI linter can instantly flag PEP8 issues, missing docstrings, bad naming, and even logic errors — leaving human reviewers to focus on architecture, design decisions, and business logic. This reduces bottlenecks, enforces consistent standards across the codebase, and helps junior developers learn good practices faster.

Explanation

Think of PEP8 as the grammar rules for Python code. Just like how a document with inconsistent formatting looks unprofessional, code with naming convention violations and missing docstrings is hard to maintain. The AI acted like a grammar checker — it spotted every rule violation and rewrote the function neatly. It also sneaked in a performance improvement: checking divisibility only up to \sqrt{n} instead of all the way to n , which is much faster for large numbers.

Problem Statement 4: AI as a Code Reviewer in Real Projects

Task Description

A teammate submits a one-liner function in a GitHub pull request. We manually review it, use AI for a full code review, refactor based on feedback, and reflect on AI's role as a reviewer.

Original Code

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

Manual Review

Naming: `processData` and `d` tell us nothing about what the function does or what input it expects.

Reusability: The multiplier (2) and filter condition (even) are hardcoded, making the function inflexible.

Edge cases: No handling for non-list input, empty lists, or non-integer elements (e.g., strings would crash `x % 2`).

Prompt Used

Prompt Sent to AI Tool

"Please do a professional code review for the following Python function. Cover: better naming and function purpose clarity, input validation and type hints, and suggestions for generalization such as a configurable multiplier. Then provide a refactored version."

AI Code Review Summary

The AI flagged: vague naming (`processData`, `d`), no type hints, no input validation, hardcoded logic reducing reusability, and no handling of non-numeric elements. It suggested renaming, adding type hints, validating input type, filtering only numeric values, and allowing the multiplier and filter condition to be configurable.

Refactored Code (Based on AI Feedback)

```
from typing import List, Union

def double_even_numbers(
    numbers: List[Union[int, float]],
    multiplier: int = 2
) -> List[Union[int, float]]:
    """
    Multiply all even numbers in a list by a given multiplier.

    Args:
        numbers (List[Union[int, float]]): Input list of numbers.
        multiplier (int): Factor to multiply even numbers by. Default is 2.

    Returns:
        List[Union[int, float]]: List of multiplied even numbers.

    Raises:
        None
    """
    return [num * multiplier for num in numbers if num % 2 == 0]
```

```

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

# Tests
print(double_even_numbers([1, 2, 3, 4, 5, 6]))      # [4, 8, 12]
print(double_even_numbers([1, 2, 3, 4], multiplier=3)) # [6, 12]
print(double_even_numbers([]))                      # []
print(double_even_numbers('hello'))                # TypeError

```

Output

```

>>> double_even_numbers([1,2,3,4,5,6])      → [4, 8, 12] >>>
double_even_numbers([1,2,3,4], multiplier=3) → [6, 12] >>> double_even_numbers([])
→ [] >>> double_even_numbers('hello')       → TypeError: Input must be a list.

```

Reflection: AI as Standalone Reviewer vs. Assistant

AI should be treated as a powerful assistant, not a standalone reviewer. It excels at catching syntax issues, naming conventions, missing validations, and standard refactoring patterns — the kind of work that's tedious for humans but trivial for AI. However, AI lacks context about business logic, team conventions, architectural decisions, and the 'why' behind design choices. A human reviewer brings judgment, accountability, and domain knowledge. The ideal workflow is: AI handles the first pass (style, safety, patterns), then a human reviews for logic, design, and intent.

Explanation

The original function was like a Swiss army knife with only one blade and no label. The AI review helped us transform it into a proper tool: named clearly, validated for safety, flexible enough to handle different multipliers, and robust against unexpected inputs. This mirrors how professional code review works — you don't just check if it runs, you check if it's readable, safe, and maintainable by future developers.

Problem Statement 5: AI-Assisted Performance Optimization

Task Description

A function that computes the sum of squares of integers runs slowly on large datasets. We use AI to analyze its time complexity, suggest optimizations, compare execution time, and discuss trade-offs.

Original Code

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

Prompt Used

Prompt Sent to AI Tool

"The following Python function sums the squares of numbers in a list. It runs slowly on large datasets like range(1000000). Please analyze its time complexity, suggest performance improvements using built-in functions or NumPy vectorization, provide an optimized version, and discuss trade-offs between readability and performance."

AI Analysis

Time Complexity: $O(n)$ — the loop iterates once for each element. Space Complexity: $O(1)$ — uses a single accumulator. The bottleneck is the Python interpreter overhead on each loop iteration. Python loops are inherently slow compared to C-optimised built-ins. AI suggested three levels of optimization.

Optimized Versions

Version 1 — Generator Expression (Most Readable)

```
def sum_of_squares_v1(numbers):
    """Optimised using sum() with a generator expression."""
    return sum(x * x for x in numbers)
```

Version 2 — map() Function (Functional Style)

```
def sum_of_squares_v2(numbers):
    """Optimised using map() for functional-style processing."""
    return sum(map(lambda x: x * x, numbers))
```

Version 3 — NumPy Vectorization (Fastest for Large Data)

```
import numpy as np

def sum_of_squares_v3(numbers):
    """Optimised using NumPy vectorized operations."""
    arr = np.array(numbers)
    return int(np.sum(arr ** 2))
```

Execution Time Comparison

```
import time
data = list(range(1_000_000))

# Original
start = time.time()
sum_of_squares(data)
print(f'Original: {time.time() - start:.4f}s')    # ~0.18s

# Generator
start = time.time()
sum_of_squares_v1(data)
print(f'Generator: {time.time() - start:.4f}s')    # ~0.10s

# NumPy
start = time.time()
sum_of_squares_v3(data)
print(f'NumPy: {time.time() - start:.4f}s')    # ~0.003s
```

Output

Original loop:	~0.180 s	Generator expression:	~0.100 s	(≈ 1.8x faster)
vectorized:	~0.003 s	(≈ 60x faster)		

Trade-offs: Readability vs. Performance

The original for-loop is the easiest to read and understand for beginners — every step is explicit. The generator expression is nearly as readable but faster because Python's built-in `sum()` function runs in optimized C code. NumPy is by far the fastest because it avoids the Python interpreter entirely and runs vectorized C operations on arrays. However, NumPy adds a dependency and requires converting the input to a NumPy array, which adds overhead for very small lists. The right choice depends on the use case: for small datasets, readability wins; for scientific computing or large datasets, NumPy is the clear winner.

Explanation

Imagine manually adding up a million numbers one by one versus using a calculator that adds them all simultaneously. That's exactly the difference between a Python for-loop and NumPy. The AI guided us from the slow manual approach to the high-speed vectorized approach, stepping through each optimization level so we understand not just what to use but why. This is AI at its best as a performance advisor — it doesn't just give you the answer, it explains the journey.