

# **AI ASSISTED CODING**

## **ASSIGNMENT - 10.3**

**ROLL NO. 2303A510J0**

### **Problem Statement 1: AI-Assisted Bug Detection**

**Scenario: A junior developer wrote the following Python function to**

**calculate factorials:**

```
def factorial(n):
```

```
    result = 1
```

```
    for i in range(1, n):
```

```
        result = result * i
```

```
    return result
```

#### **Instructions:**

- 1. Run the code and test it with factorial(5).**
- 2. Use an AI assistant to:**
  - o Identify the logical bug in the code.**
  - o Explain why the bug occurs (e.g., off-by-one error).**
  - o Provide a corrected version.**
- 3. Compare the AI's corrected code with your own manual fix.**
- 4. Write a brief comparison: Did AI miss any edge cases (e.g.,**

negative numbers, zero)?

**Expected Output:**

**Corrected function should return 120 for factorial(5).**

**Original code:**

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

**Output:**

24

**Prompt:**

**Review the code**



```
    Raises `TypeError` if `n` is not an int and `ValueError` if `n` is
negative.
    """
    if not isinstance(n, int):
        raise TypeError("n must be an integer")
    if n < 0:
        raise ValueError("n must be non-negative")

    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

if __name__ == "__main__":
    print(factorial(5))
```

## Output:

120

### Brief Comparison — AI vs Manual Fix

The bug in the original factorial function was an off-by-one error because `range(1, n)` excludes `n`, so `factorial(5)` returned 24 instead of 120. My manual fix changed it to `range(1, n+1)` to include the final value.

The AI fix also corrected the loop but additionally added edge-case handling for zero, negative numbers, and non-integer inputs. My fix was minimal, while the AI version was more robust.

Conclusion: The AI solution handled more edge cases than my manual fix.

---

## **Problem Statement 2: Task 2 — Improving Readability & Documentation**

**Scenario:** The following code works but is poorly written:

```
.  
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":
```

**Instructions:**

**5. Use AI to:**

- o Critique the function's readability, parameter naming, and lack of documentation.**
- o Rewrite the function with:**
  - 1. Descriptive function and parameter names.**
  - 2. A complete docstring (description, parameters, return value, examples).**

3. Exception handling for division by zero.
4. Consideration of input validation.
6. Compare the original and AI-improved versions.
7. Test both with valid and invalid inputs (e.g., division by zero, non-string operation)

## 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  
  
print(calc(10, 0, "div"))
```

## Output:

```
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_10.3> & C:/Users/rohit/AppData/Local/Programs/Python/Python314/python.exe c:/Users/rohit/OneDrive/Documents/SRU/ai_code/lab_10.3/t2.py
Traceback (most recent call last):
  File "c:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_10.3\t2.py", line 11, in <module>
    print(calc(10, 0, "div"))
          ~~~~~
  File "c:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_10.3\t2.py", line 9, in calc
    return a / b
           ~~~~
ZeroDivisionError: division by zero
```

## Prompt:

review the function, then improve it.

Do the following in order:

1. Review the code and list readability issues, poor naming, missing documentation, lack of input validation, missing exception handling, and unhandled edge cases.
2. Rewrite the function with:
  - Descriptive function and parameter names
  - Full docstring (description, parameters, return value, raised exceptions, examples)
  - Input validation (numeric inputs, operation must be a string, supported operations only)
  - Exception handling (division by zero and invalid operation)

3. Provide a short comparison table (original vs improved).
4. Show test cases with expected outputs for: valid inputs, division by zero, invalid operation, non-numeric input, and non-string operation.

```
← FUNCTION REVIEW AND IMPROVEMENT PROCESS
```

Test cases and expected outputs

- Valid input:
  - Call: `calculate_operation(5, 3, 'add')`
  - Expected: `8`
- Division by zero:
  - Call: `calculate_operation(10, 0, 'div')`
  - Expected: raises `ZeroDivisionError` with message "Division by zero is not allowed for 'div' operation"
- Invalid operation:
  - Call: `calculate_operation(1, 2, 'pow')`
  - Expected: raises `ValueError` with message like "Unsupported operation 'pow'. Supported: add, div, mul, sub"
- Non-numeric input:
  - Call: `calculate_operation('a', 2, 'add')`
  - Expected: raises `TypeError` with message "operand1 and operand2 must be numeric (int or float)"
- Non-string operation:
  - Call: `calculate_operation(1, 2, 3)`
  - Expected: raises `TypeError` with message "operation must be a string"

What I changed in the workspace

- Edited `t2.py`: replaced the original `calc` with `calculate_operation` and added a small demonstration harness under `if __name__ == '__main__':`.

Would you like me to:

**Corrected code:**



```

def calculate_operation(operand1, operand2, operation):
    """
    Perform a basic arithmetic operation on two numeric operands.

    Parameters
    -----
    operand1 : int | float
        The first numeric operand.
    operand2 : int | float
        The second numeric operand.
    operation : str
        The operation to perform. Supported values: 'add', 'sub', 'mul',
'div'.

    Returns
    -----
    int | float
        The result of applying the operation to the operands.

    Raises
    -----
    TypeError
        If `operand1` or `operand2` are not numeric, or if `operation` is
not a string.
    ValueError
        If `operation` is not one of the supported operations.
    ZeroDivisionError
        If `operation` is 'div' and `operand2` is zero.

    Examples
    -----
    >>> calculate_operation(3, 2, 'add')
    5
    >>> calculate_operation(10, 2, 'div')
    5.0
    >>> calculate_operation(10, 0, 'div')
    Traceback (most recent call last):
        ...
    ZeroDivisionError: Division by zero is not allowed for 'div' operation
    """

```

```

# Validate types
if not isinstance(operation, str):
    raise TypeError("operation must be a string")

if not isinstance(operand1, (int, float)) or not isinstance(operand2,
(int, float)):
    raise TypeError("operand1 and operand2 must be numeric (int or
float)")

op = operation.lower()

_ops = {
    'add': lambda x, y: x + y,
    'sub': lambda x, y: x - y,
    'mul': lambda x, y: x * y,
    'div': lambda x, y: x / y,
}

if op not in _ops:
    raise ValueError(f"Unsupported operation {operation!r}. Supported:
{'', ' '.join(sorted(_ops.keys()))}")

try:
    result = _ops[op](operand1, operand2)
except ZeroDivisionError:
    raise ZeroDivisionError("Division by zero is not allowed for 'div'
operation")

return result

if __name__ == '__main__':
    # Small built-in demonstration/tests when run as a script
    cases = [
        ("Valid add", 5, 3, 'add'),
        ("Valid div", 10, 2, 'div'),
        ("Div by zero", 10, 0, 'div'),
        ("Invalid op", 1, 2, 'pow'),
        ("Non-numeric", 'a', 2, 'add'),
    ]

```

```

        ("Non-string op", 1, 2, 3),
    ]

    for desc, a, b, op in cases:
        try:
            res = calculate_operation(a, b, op)
        except Exception as e:
            print(f"{desc}: Exception: {type(e).__name__}: {e}")
        else:
            print(f"{desc}: {res}")

```

## Output:

```

Valid add: 8
Valid div: 5.0
Div by zero: Exception: ZeroDivisionError:
Division by zero is not allowed for 'div'
operation
Invalid op: Exception: ValueError:
Unsupported operation 'pow'. Supported:
add, div, mul, sub
Non-numeric: Exception: TypeError: operand1
and operand2 must be numeric (int or float)
Non-string op: Exception: TypeError:
operation must be a string

```

**Aspect**

**Manual Fix**

**AI Fix**

<b>Bug Identified</b>	<b>Off-by-one loop error</b>	<b>Off-by-one loop error</b>
<b>Core Correction</b>	Changed <code>range(1, n) → range(1, n+1)</code>	<b>Same loop correction</b>
<b>Output for factorial(5)</b>	<b>120</b>	<b>120</b>
<b>Handles Zero Case</b>	<b>Works (returns 1)</b>	<b>Works (explicitly handled)</b>
<b>Handles Negative Input</b>	<b>Not handled</b>	<b>Raises proper error</b>
<b>Handles Non-Integer Input</b>	<b>Not handled</b>	<b>Input validation added</b>
<b>Robustness</b>	<b>Basic fix</b>	<b>More robust and safe</b>
<b>Production Ready</b>	<b>Partially</b>	<b>Yes</b>

---

### Problem Statement 3: Enforcing Coding Standards

**Scenario: A team project requires PEP8 compliance. A developer submits:**

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

**Instructions:**

**8. Verify the function works correctly for sample inputs.**

**9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter with AI explanation) to:**

- o List all PEP8 violations.**

- o Refactor the code (function name, spacing, indentation, naming).**

**10. Apply the AI-suggested changes and verify functionality is**

**Preserved.**

**11. Write a short note on how automated AI reviews could streamline code reviews in large teams.**

**Expected Output:**

**A PEP8-compliant version of the function, e.g.:**

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

**Code:**

```
def Checkprime(n) :  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True  
  
print(Checkprime(7))
```

**Output:**

**True**

**Prompt:**



Check the attached Python file and review the prime checking function in it.

Do these steps:

- Test the function with some inputs like 2, 3, 4, 5, 9, 0, 1 and negative numbers and show what it returns.
- Tell me what PEP8 rules are being broken (like naming, spacing, indentation, missing docstring, etc.).
- Rewrite the function so it follows PEP8 properly (better function name, spacing, indentation, add a docstring, basic input checks).
- Show the corrected code.

- Test the new version also and confirm it still works.

Keep the answer structured but simple.

 **REVIEW AND REFACTOR PRIME CHECKING FUNCTION IN PYTHON** 

I'll create a short TODO plan, run the original function on test inputs, then patch and retest the corrected function.

✓ Reviewed todo list management

Now I'll run the original `Checkprime` function against the requested inputs.



✓ Ran `pylanceRunCodeSnippet` – pylance mcp server (MCP Server)

Progress update: original tests completed — next I'll list PEP8 issues, then patch `t3.py`.

### PEP8 Issues

- **Naming:** function name `Checkprime` uses CamelCase; should be `snake_case` (`is_prime`).
- **Missing docstring:** no docstring explaining behavior and params.
- **No type checks / input validation:** function accepts non-integers/negatives incorrectly.
- **Algorithm correctness:** not PEP8 but important — treats 0,1, negatives as prime (logical bug).
- **Spacing:** file has no blank line after function (minor).
- **Imports:** none, but if using math functions, import should be at top.

Now I'll patch `t3.py` to a PEP8-compliant version with docstring, input checks, and improved algorithm, and include tests. Patching `t3.py` to replace the original function with a PEP8-compliant `is_prime` and test harness. I'll apply the change now.

GPT-5 mini • 1x

## Corrected code:

```
def is_prime(n: int) -> bool:
    """Return True if ``n`` is a prime number, otherwise False.

    Parameters
    -----
    n : int
        Integer to test for primality.

    Raises
    -----
    TypeError
        If ``n`` is not an integer.
    """
    if not isinstance(n, int):
        raise TypeError("n must be an integer")

    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    limit = int(n ** 0.5) + 1
    for i in range(3, limit, 2):
        if n % i == 0:
            return False
    return True


def _run_tests() -> None:
    tests = [2, 3, 4, 5, 9, 0, 1, -5, -2, 7]
    for x in tests:
        try:
            print(f"{x}: {is_prime(x)}")
        except Exception as exc:
            print(f"{x}: Error - {exc}")
```



```
if __name__ == "__main__":  
    _run_tests()
```

## Output:

```
2: True  
3: True  
4: False  
5: True  
9: False  
0: False  
1: False  
-5: False  
-2: False  
7: True
```

## AI reviews in large teams

AI-based code reviews help large teams by quickly detecting style violations, naming issues, and missing validations. They provide instant, consistent feedback and reduce manual review time. This allows human reviewers to focus more on logic and design instead of formatting and standards, speeding up the overall development process.

Scenario:

In a GitHub project, a teammate submits:

```
def processData(d):
```

```
    return [x * 2 for x in d if x % 2 == 0]
```

Instructions:

1. Manually review the function for:
  - o Readability and naming.
  - o Reusability and modularity.
  - o Edge cases (non-list input, empty list, non-integer elements).
2. Use AI to generate a code review covering:
  - a. Better naming and function purpose clarity.
  - b. Input validation and type hints.
  - c. Suggestions for generalization (e.g., configurable multiplier).
3. Refactor the function based on AI feedback.
4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.

Expected Output:

An improved function with type hints, validation, and clearer intent, e.g.:

```
from typing import List, Union
def double_even_numbers(numbers: List[Union[int, float]]) -> List[Union[int, float]]:
    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]
```

## Code:

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

print(processData([1, 2, 3, 4, 5, 6]))
```

## Output:

```
[4, 8, 12]
```

## Prompt:

Review the attached Python file that has the function for processing data.

Do the following:

- First manually review the function and explain issues with readability, naming, reusability and modularity.
- Mention edge cases like non-list input, empty list, and non-integer elements.
- Then give an AI-style code review with suggestions for:
  - better function and variable naming
  - clearer purpose
  - adding type hints
  - input validation
  - making it more general
- Rewrite/refactor the function based on those suggestions.
- Show the improved version of the code.
- Finally write a short reflection (5–6 lines) on whether AI should be a standalone code reviewer or just an assistant.

Keep the explanation simple and structured.

## Corrected code:

```
from typing import Iterable, List

def process_numbers(data: Iterable[int], multiplier: int = 2) -> List[int]:
    """Return a list with each even integer from `data` multiplied by
    `multiplier`.


Args:



data: An iterable of integers.



multiplier: Integer factor to multiply even values by (default 2).



Raises:



TypeError: If `data` is not iterable or contains non-integer
elements,



or if `multiplier` is not an int.


```

```

Returns:
    A list of processed integers.
"""
if not isinstance(multiplier, int):
    raise TypeError("multiplier must be an int")

try:
    seq = list(data)
except TypeError:
    raise TypeError("data must be an iterable (e.g., list) of
integers")

result: List[int] = []
for i, x in enumerate(seq):
    if not isinstance(x, int):
        raise TypeError(f"element at index {i} is not an int: {x!r}")
    if x % 2 == 0:
        result.append(x * multiplier)

return result

if __name__ == "__main__":
    sample = [1, 2, 3, 4, 5, 6]
    print(process_numbers(sample))

```

## Output:

```
[4, 8, 12]
```

## Reflection:

AI is useful in code reviews because it quickly finds naming issues, missing validation, and style problems. It gives fast and consistent feedback and helps developers improve code quality. However, AI should not be a standalone reviewer because it may miss project context or design decisions. Human reviewers are still needed for logic, architecture, and business requirements. So AI works best as a code review assistant, not a replacement.

---

### Problem Statement 5: — AI-Assisted Performance Optimization

Scenario: You are given a function that processes a list of integers, but it runs slowly on large datasets:

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

Instructions:

1. Test the function with a large list (e.g., `range(1000000)`).
2. Use AI to:
  - o Analyze time complexity.
  - o Suggest performance improvements (e.g., using built-in functions, vectorization with NumPy if applicable).
  - o Provide an optimized version.
3. Compare execution time before and after optimization.
4. Discuss trade-offs between readability and performance.

Expected Output:

An optimized function, such as:

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

### Code:

```
def sum_of_squares(numbers):  
    total = 0  
    for num in numbers:  
        total += num ** 2  
    return total  
  
print(sum_of_squares([1, 2, 3, 4]))
```

### Output:

30

### Prompt:

Check the attached Python file that contains the `sum_of_squares` function.

Do these steps:

- Test it with a large list like `range(1000000)` and show what happens.
- Analyze the time complexity in simple terms.
- Suggest ways to make it faster (like built-in functions or NumPy if suitable).
- Give an optimized version of the function.
- Compare execution time before and after optimization with example timing code.
- Keep the explanation simple and structured.

## Test Results

- **File:** `t5.py:1-8`
- **What I ran:** timed sum of squares on `range(1_000_000)`.
- **Observed timings (best of 3 runs):**
  - **original\_time** (uses `num ** 2`): 0.06479 s
  - **fast\_time** (uses `n * n`): 0.03405 s
  - **builtin\_time** (`sum(n*n for n in numbers)`): 0.04671 s
  - **numpy\_time:** numpy not available in this environment

## Time Complexity

- **Big-O:**  $O(n)$  — the function does one operation per element.
- **Simple:** runtime grows linearly with the number of items.

## Ways to make it faster

- **Micro-optimizations:** replace exponent `** 2` with multiplication `n * n` (faster in CPython).
- **Reduce attribute lookups:** keep frequently used names in local variables.
- **Built-in helpers:** `sum(...)` with a generator can be concise — performance varies.
- **Vectorized libraries:** use NumPy to compute squares and sum in C (large speedups for big numeric arrays).
- **Alternative runtimes:** PyPy or a C extension to get more speed for pure-Python loops.

Optimized code:

Micro version:

```
def sum_of_squares_fast(numbers):  
    total = 0  
    for n in numbers:  
        total += n * n
```

```
        return total

print(sum_of_squares_fast([1, 2, 3, 4]))
```

**Output:**

```
30
```

**Numpy version:**

```
import numpy as np

def sum_of_squares_numpy(numbers):
    arr = np.array(numbers, dtype=np.int64)    # conversion cost included
    return int((arr * arr).sum())

print(sum_of_squares_numpy([1, 2, 3, 4]))
```

**Output:**

```
30
```

## Readability vs Performance

Performance optimizations can make code run faster, especially on large datasets, but sometimes they reduce readability. A simple loop is easier for beginners to understand, while optimized versions using generators, built-in functions, or NumPy are faster but slightly harder to read. In most real projects, a balance is preferred — use readable code first, then optimize only where performance actually matters. Clean and fast code together is the best goal.



Aspect	Original Version	Optimized Version
Implementation	Uses manual loop and accumulator	Uses built-in <code>sum()</code> with generator expression
Code Length	Longer	Shorter
Readability	Very easy to understand	Still readable but slightly more advanced
Time Complexity	$O(n)$	$O(n)$
Practical Speed	Slower due to Python loop overhead	Faster due to optimized built-in sum
Memory Usage	Low	Low (generator expression is memory efficient)
Large Dataset Performance	Moderate	Better
Pythonic Style	Basic	More Pythonic