

ASSIGNMENT-10.3

2303A51644

BATCH-28

QUESTION-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)?

PYTHON ORIGINAL CODE :

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS AZURE
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding>
URSES/AI_Assistant_Coding/.venv/Scripts/Activate.ps1"
● (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_
RNING COURSES/AI_Assistant_Coding/.venv/Scripts/python.exe" "c:/Users/DE
ant_Coding/fact.py"
24
```

AI REVIEWED CODE :

The screenshot shows a code editor with a Python file named fact.py. The code defines a factorial function that includes type and value error handling, and a quick smoke-check at the end. To the right is a 'CODE REVIEW REQUEST' window. It shows a summary of changes made: fixing a loop, adding type/value checks, and updating a docstring. It also notes that the verification of factorial(5) returned 120 (verified). A 'Changes' section lists the update to the factorial function. Below the review window is a terminal window showing the execution of fact.py and the output 120.

```
fact.py > ...
1 def factorial(n: int) -> int:
2     """Return n! for a non-negative integer n.
3
4     Raises TypeError if 'n' is not an int and ValueError if 'n' is negative.
5     """
6     if not isinstance(n, int):
7         raise TypeError("n must be an integer")
8     if n < 0:
9         raise ValueError("n must be non-negative")
10    result = 1
11    for i in range(1, n + 1):
12        result *= i
13    return result
14
15
16 if __name__ == "__main__":
17     # quick smoke-check
18     print("5! =", factorial(5))
19 print(factorial(5))

← CODE REVIEW REQUEST
COURSES/AI_Assistant_Coding/.venv/Scripts/python.exe" -c "from fact import factorial; print(factorial(5))"
"
✓ Completed: Update fact.py (2/3)

Summary
• File: fact.py:1-40 — updated.
• Fix: Corrected loop to include n, added type/value checks, docstring, and a smoke-check.
• Verification: factorial(5) returns 120 (verified).

Changes
• [fact] factorial | now:
Todos (3/3)
Describe what to build next
```

OUTPUT :

```
...
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding> & "C:/Users/DELL/OneDrive/Desktop\LEARNING COURSES\AI_Assistant_Coding/.venv/Scripts/python.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assistant_Coding/fact.py"
● 5! = 120
120
○ (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding>
```

JUSTIFICATION :

1. The original function contains an off-by-one error because range(1, n) excludes n, leading to incorrect factorial results.
2. AI helps quickly identify logical flaws without manual debugging cycles.
3. The corrected version ensures accurate computation for standard inputs like factorial(5) = 120.
4. AI may not always handle edge cases (negative numbers, non-integers) unless explicitly prompted.
5. Combining manual review with AI improves correctness and robustness.

QUESTION-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).
```

PYTHON CODE 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
```

AI REVIEWED CODE :

Program.py > calc

```
1 from typing import Union
2 def calc(a: Union[int, float], b: Union[int, float], op: str) -> Union[int, float]:
3     """Perform a simple arithmetic operation on `a` and `b`.
4
5     `op` may be one of: 'add', 'sub', 'mul', 'div' (case-insensitive).
6     Raises ValueError for unknown operations and ZeroDivisionError for division by zero.
7
8     if not isinstance(op, str):
9         raise TypeError("op must be a string")
10    op_key = op.strip().lower()
11    if op_key == "add":
12        return a + b
13    if op_key == "sub":
14        return a - b
15    if op_key == "mul":
16        return a * b
17    if op_key == "div":
18        if b == 0:
19            raise ZeroDivisionError("division by zero")
20        return a / b
21    raise ValueError(f"unknown operation: {op}")
22
23
24 if __name__ == "__main__":
25     # quick examples
26     print("2 + 3 =", calc(2, 3, "add"))
27     print("5 - 2 =", calc(5, 2, "sub"))
28     print("3 * 4 =", calc(3, 4, "mul"))
29     print("10 / 2 =", calc(10, 2, "div"))
```

CODE REVIEW REQUEST

correct this code and review this

I'll create a short todo plan, then open and read [Program.py](#) to locate errors.

✓ Reviewed Program.py and updated task statuses

Marking file-read step complete: next I'll patch [Program.py](#) to add validation, type hints, and examples.

✓ Updated Program.py

Run [pwsh](#) command?

& "C:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assistant_Coding/.venv/Scripts/python.exe" -c "from Program import calc; print(calc(2,3,'add')); print(calc(5,2,'sub')); print(calc(3,4,'mul')); print(calc(10,2,'div'))"

Allow Skip

Fix code and run quick checks (3/4)

Program.py

OUTPUT :

```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding>
eDrive/Desktop/LEARNING COURSES/AI_Assistant_Coding/.venv/Scripts/python.exe" "
ve/Desktop/LEARNING COURSES/AI_Assistant_Coding/Program.py"
2 + 3 = 5
5 - 2 = 3
3 * 4 = 12
10 / 2 = 5.0
> (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI Assisant Coding>
```

JUSTIFICATION :

The original function lacks meaningful naming, reducing code clarity.

Absence of a docstring makes the function harder to understand and reuse.

No exception handling (e.g., division by zero) makes it unsafe.

AI refactoring improves maintainability through documentation and validation.

Structured improvements align the function with professional coding standards.

QUESTION-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.

PYTHON CODE ORIGINAL CODE :

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

AI REVIEWED CODE :

The screenshot shows a GitHub code review interface. On the left, the code file 'Program.py' is displayed with syntax highlighting and line numbers. The code implements a prime number checker using the `isqrt` function from the `math` module. It handles edge cases and uses a quick烟检 (smoke check) for small numbers. On the right, the 'CODE REVIEW REQUEST' pane is open, showing a list of changes made:

- What I changed:** Replaced a fragile `Checkprime` with `is_prime`:
 - Added type checking and docstring.
 - Handles $n < 2$ correctly (returns False for 0 and 1).
 - Uses `isqrt(n)` and checks only odd divisors (faster).
 - Added smoke-checks under `if __name__ == "__main__":`.
- Verification:** `is_prime(17) → True, is_prime(18) → False` (import and run verified).

Below the changes, there are notes and suggestions, a 'Todos' section, and a 'Describe what to build next' field.

OUTPUT :

```
-> ./Program.py
/Desktop/LEARNING COURSES/AI_Assistant_Coding/.venv/Scripts/python.exe "c:/Users/DELL/OneDrive/DESKTOP/LEARNING COURSES/AI_Assistant_Coding/Program.py"
0 False
1 False
2 True
3 True
4 False
17 True
18 False
19 True
20 False
97 True
```

JUSTIFICATION :

The function violates **PEP8 naming conventions** (e.g., `Checkprime` instead of `check_prime`).

Proper spacing and formatting improve readability and team collaboration.

AI tools can systematically detect style violations faster than manual review.

Refactoring preserves logic while improving compliance.

Automated reviews enhance consistency in large development teams.

QUESTION-4 :

AI as a Code Reviewer in Real Projects

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.

PYTHON ORIGINAL CODE :

```
⌚ Program.py > ...
1 ✘ def processData(d):
2     return [x*2 for x in d if x%2==0]
3
```

AI REVIEWED CODE :

The screenshot shows the Grafana AI-Powered Root Cause Assistant interface. On the left, there's a sidebar with a 'TASK' button, a progress bar, and a 'Start New Task' button. Below the progress bar, it says 'correct this code and review it'. Under 'Test Results', there are several test cases with their expected outputs. At the bottom of the sidebar, it says 'The code has been successfully corrected and reviewed with all improvements implemented.' On the right, the code editor shows the original code at the top, followed by a detailed AI-generated review. The review includes a docstring explaining the function's purpose, arguments, and returns. It also provides notes about float conversion and negative numbers. A red box highlights a syntax error in the original code: 'try: Statements must be separated by newlines or semicolons'. The code editor also shows imports at the top and various comments and print statements throughout the script.

```
⌚ Program.py > processData
1 from typing import List, Union
2 def processData(d: List[Union[int, float]]) -> List[int]:
3     """
4         Process data by doubling only even numbers.
5         Args:
6             d: A list of numbers (integers or floats)
7         Returns:
8             A list containing only the even numbers from the input,
9             each multiplied by 2.
10        Note:
11            - Float values are converted to int for the modulo check
12            - Negative even numbers are also included
13    """try: Statements must be separated by newlines or semicolons
14        return [int(x) * 2 for x in d if int(x) % 2 == 0]
15    except (TypeError, ValueError) as e:
16        raise TypeError(f"Input must be a list of numeric values: {e}")
17    # Example usage:
18    if __name__ == "__main__":
19        # Test with various inputs
20        test_data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
21        print(f"Input: {test_data}")
22        print(f"Output: {processData(test_data)}")
23
24    # Test with negative numbers
25    test_data_neg = [-4, -3, -2, -1, 0, 1, 2, 3, 4]
26    print(f"Input (with negatives): {test_data_neg}")
27    print(f"Output: {processData(test_data_neg)}")
28
29    # Test with empty list
30    print(f"Empty list: {processData([])}")
31
```

OUTPUT :

```
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Output: [4, 8, 12, 16, 20]
Input (with negatives): [-4, -3, -2, -1, 0, 1, 2, 3, 4]
Output: [-8, -4, 0, 4, 8]
Empty list: []
D:\venv\PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI Assistant>
```

JUSTIFICATION :

Poor naming (processData, d) reduces function clarity.

Lack of input validation may cause runtime errors.

AI suggests type hints, improving static analysis and maintainability.

Generalizing logic (configurable multiplier) increases reusability.

AI works best as an assistant reviewer, not a standalone replacement for human judgment.

QUESTION-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.

PYTHON ORIGINAL CODE :

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

AI REVIEWED CODE :

```

Program.py > sum_of_squares
1 from typing import List, Union
2 def sum_of_squares(numbers: List[Union[int, float]]) -> int:
3     """
4         Calculate the sum of squares of all numbers in a list.
5     """
6     Args:
7         numbers: A list of numbers (integers or floats)
8
9     Returns:
10        The sum of the squares of all numbers in the list
11    """
12
13    try:
14        return sum(num ** 2 for num in numbers)
15    except (TypeError, ValueError) as e:
16        raise TypeError(f"Input must be a list of numeric values: {e}")
17
18    # Example usage:
19    if __name__ == "__main__":
20        # Test with various inputs
21        test_data = [1, 2, 3, 4, 5]
22        print(f"Input: {test_data}")
23        print(f"Sum of squares: {sum_of_squares(test_data)}")
24
25    # Test with negative numbers
26    test_data_neg = [-1, -2, -3, 0, 1, 2, 3]
27    print(f"Input (with negatives): {test_data_neg}")
28    print(f"Sum of squares: {sum_of_squares(test_data_neg)}")
29
30    # Test with empty list
31    print(f"Empty list: {sum_of_squares([])}")
32

```

OUTPUT :

```

● (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/Python Scripts/python.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assistant_Coding/Program.py"
Input: [1, 2, 3, 4, 5]
Sum of squares: 55
Input (with negatives): [-1, -2, -3, 0, 1, 2, 3]
Sum of squares: 28
Empty list: 0
○ (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assistant_Coding> []

```

JUSTIFICATION :

The original loop has O(n) time complexity but can be written more efficiently.

AI recommends Pythonic constructs (generator expressions with sum()).

Optimized versions improve performance and memory efficiency.

Performance testing validates measurable improvements.

Trade-offs exist between readability, dependency use (e.g., NumPy), and optimization gains.

