ASSIGNMENT-10.3

HT NO:2303A51585

Batch No:28

Course :AI Assisted Coding

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)?

Python Code:

1.Original Code and Output

```
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_Assisant_Coding>
URSES/AI_Assisant_Coding/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_
RNING COURSES/AI_Assisant_Coding/.venv/Scripts/python.exe" "c:/Users/DE
ant_Coding/fact.py"
24
```

2.AI Reviewed Code:

```
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))
20
```

CODE REVIEW REQUEST

COURSES/AI_Assisant_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

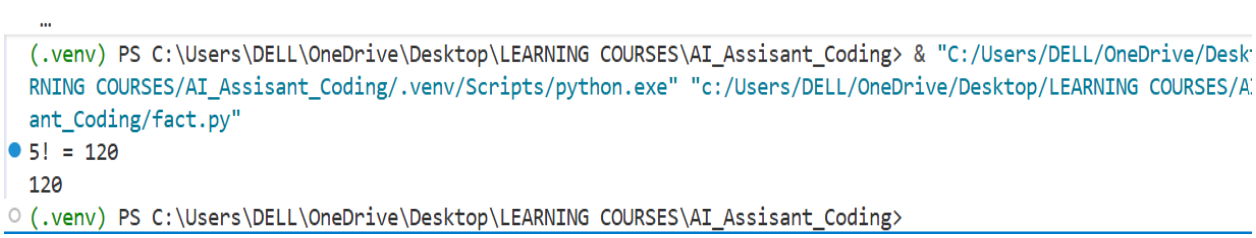
- `factorial` now:

Todos (3/3)

fact.py

Describe what to build next

Output:



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.

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).

Python Code:

1.Original Code:

```python
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
```

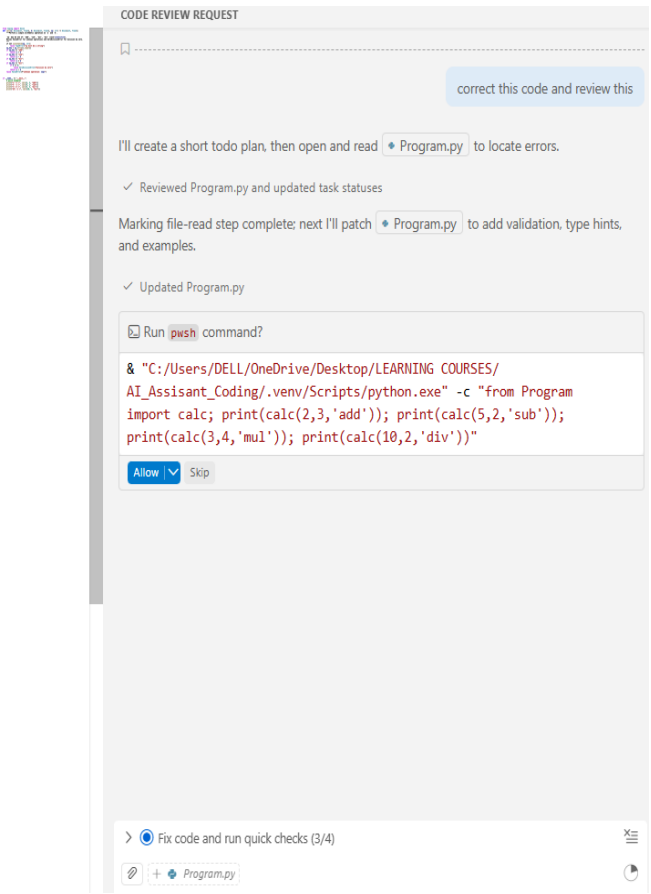
2.AI Reviewed Code:

Program.py > calc

```python
from typing import Union
def calc(a: Union[int, float], b: Union[int, float], op: str) -> Union[int, float]:
    """Perform a simple arithmetic operation on `a` and `b`.

    `op` may be one of: 'add', 'sub', 'mul', 'div' (case-insensitive).
    Raises ValueError for unknown operations and ZeroDivisionError for division by zero.
    """
    if not isinstance(op, str):
        raise TypeError("op must be a string")
    op_key = op.strip().lower()
    if op_key == "add":
        return a + b
    if op_key == "sub":
        return a - b
    if op_key == "mul":
        return a * b
    if op_key == "div":
        if b == 0:
            raise ZeroDivisionError("division by zero")
        return a / b
    raise ValueError(f"unknown operation: {op}")


if __name__ == "__main__":
    # quick examples
    print("2 + 3 =", calc(2, 3, "add"))
    print("5 - 2 =", calc(5, 2, "sub"))
    print("3 * 4 =", calc(3, 4, "mul"))
    print("10 / 2 =", calc(10, 2, "div"))
```



Output:

```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
eDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/.venv/Scripts/python.exe" "
ve/Desktop/LEARNING COURSES/AI_Assisant_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:

1.The original function lacks meaningful naming, reducing code clarity.

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

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

4.AI refactoring improves maintainability through documentation and validation.

5.Structured improvements align the function with professional coding standards.

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.

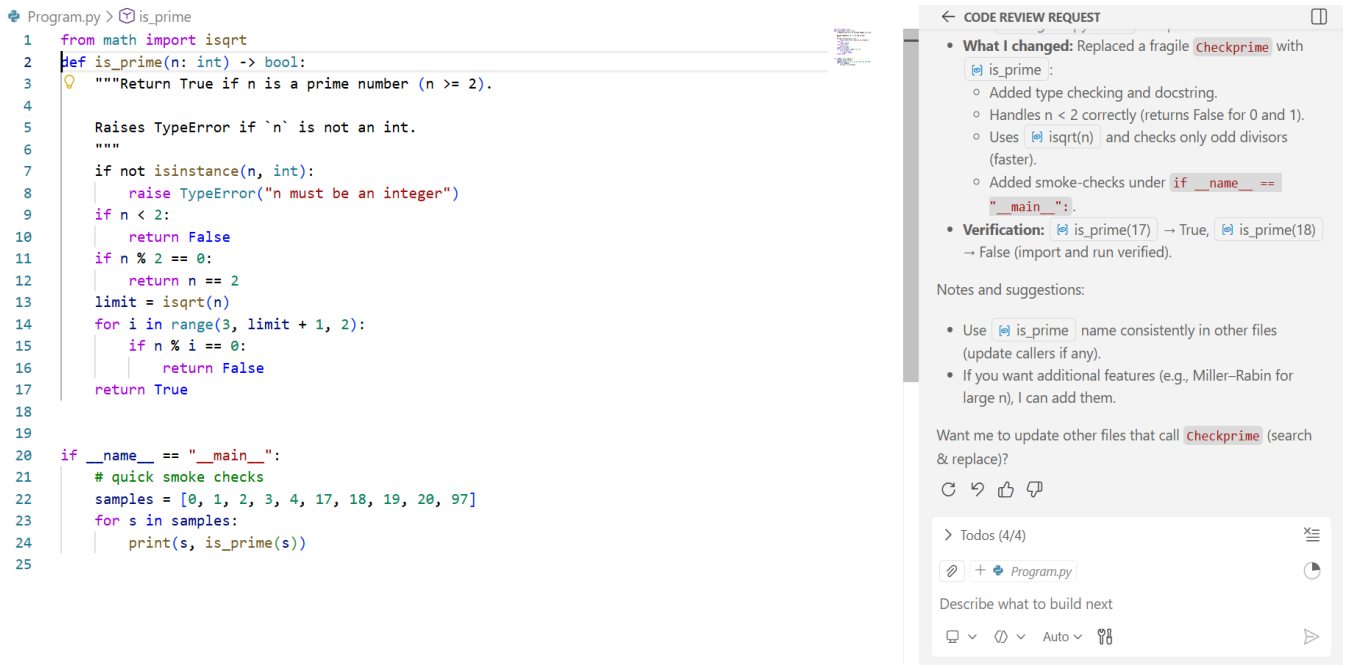
11. Write a short note on how automated AI reviews could streamline code reviews in large teams.

Python Code:

1.Original Code:

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

AI Reviewed Code:



Output:

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

Justification:

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

2.Proper spacing and formatting improve readability and team collaboration.

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

4.Refactoring preserves logic while improving compliance.

5.Automated reviews enhance consistency in large development teams.

Problem Statement 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 Code:

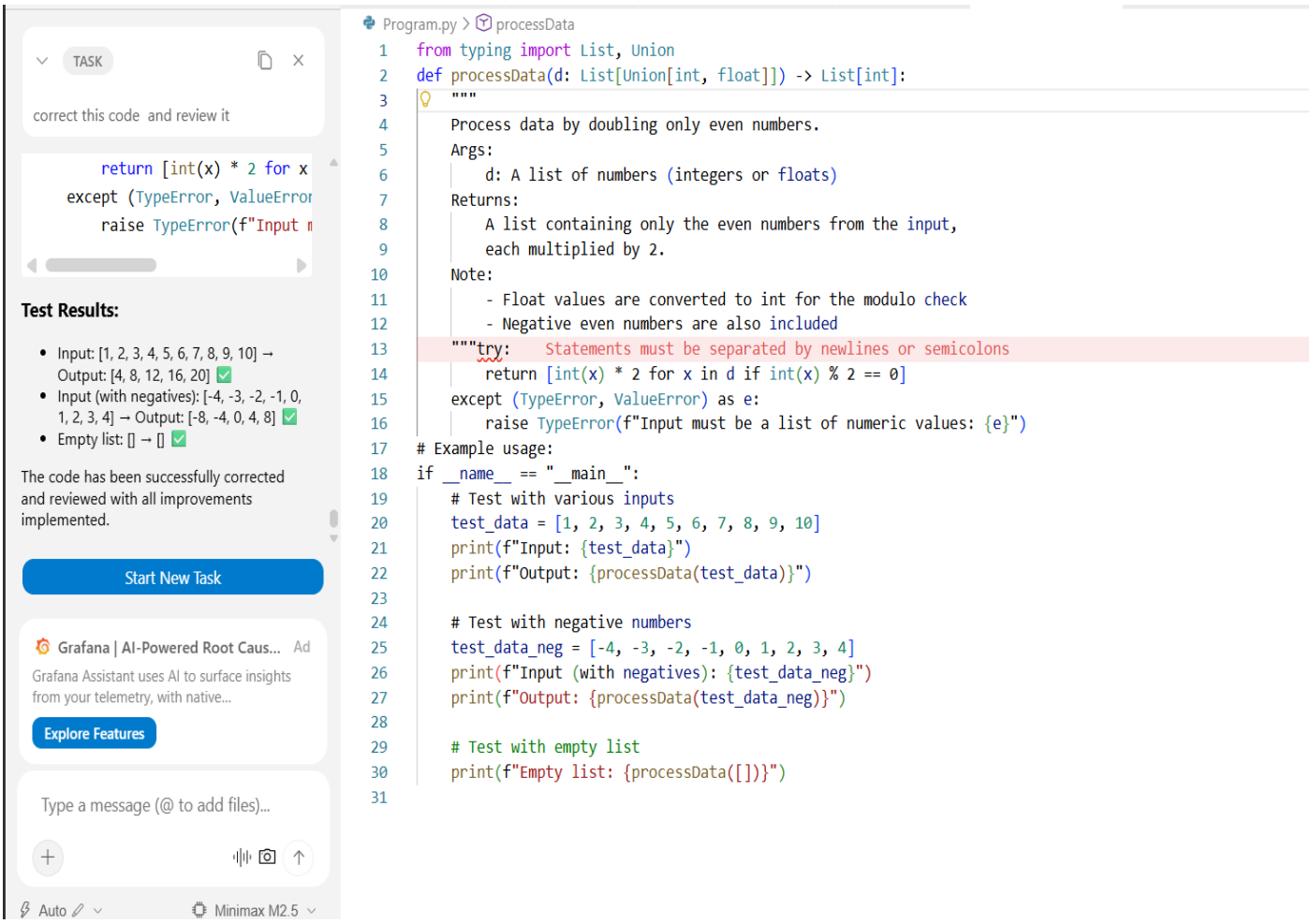
1.Original Code:

Program.py > ...

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

```

2.AI Reviewed Code:



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: []
```

Justification:

1.Poor naming (processData, d) reduces function clarity.

2.Lack of input validation may cause runtime errors.

3.AI suggests type hints, improving static analysis and maintainability.

4.Generalizing logic (configurable multiplier) increases reusability.

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

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.

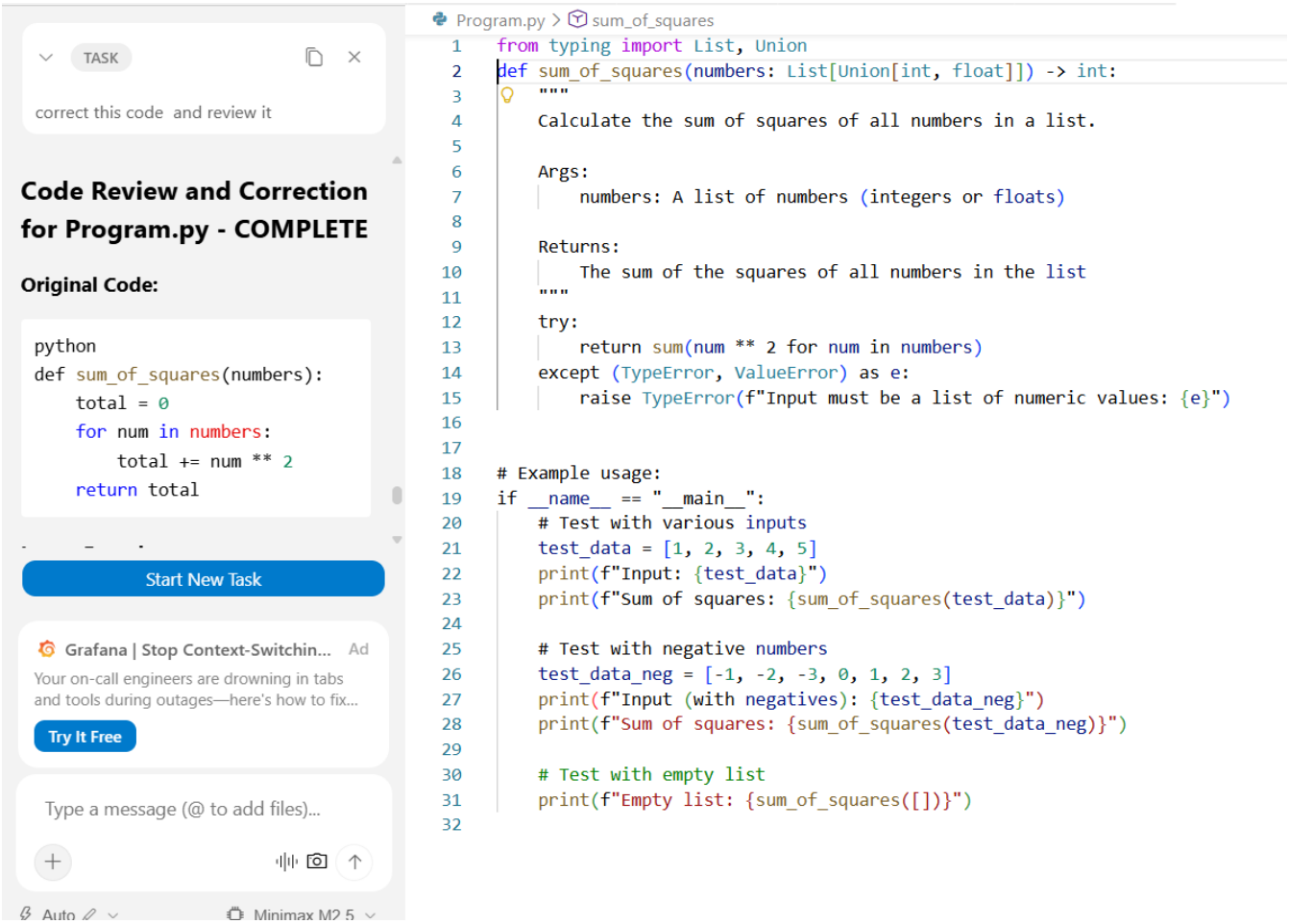
4. Discuss trade-offs between readability and performance.

Python Code:

1.Original Code:

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

2.AI Reviewed Code:



Output:

```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEA
v/Scripts/python.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_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_Assisant_Coding>
```

Justification:

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

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

3.Optimized versions improve performance and memory efficiency.

4.Performance testing validates measurable improvements.

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