ASSIGNMENT – 10.3

V.Vishnu Dattu

2303A52035

Batch – 38

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)?

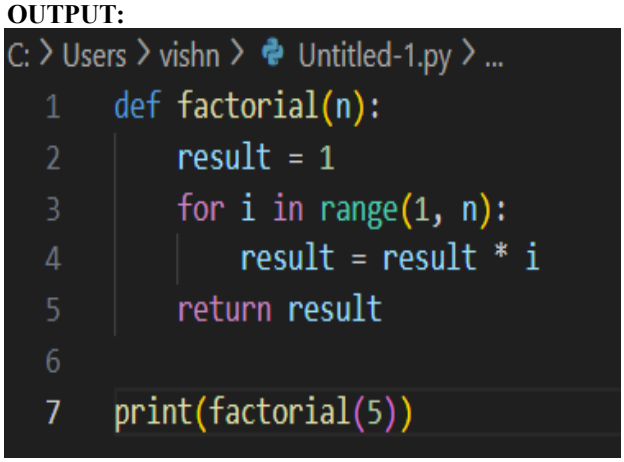
Expected Output:

Corrected function should return 120 for factorial(5).

PROMPT:

Identify the bug in this Python factorial function, explain why it happens, and provide a corrected version. Also handle edge cases like 0 and negative numbers.

OUTPUT:



C: > Users > vishn > Untitled-1.py > ...

```python
def factorial(n):
    if not isinstance(n, int):
        raise TypeError("Factorial is only defined for integers.")
    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):
        result = result * i
    return result
print("Factorial of 5:", factorial(5))
print("Factorial of 0:", factorial(0))
```

```
Factorial of 5: 120
Factorial of 0: 1
```

The original `factorial` function had a crucial bug: the `range(1, n)` in the loop means it iterates from 1 up to `n-1`, **excluding** `n` **itself**. For example, `factorial(5)` would calculate `1 * 1 * 2 * 3 * 4` (which is 24), instead of the correct `1 * 1 * 2 * 3 * 4 * 5` (which is 120).

I've corrected this by changing the loop to `range(1, n + 1)` to include `n` in the multiplication.
Additionally, I've added checks to handle edge cases:

- `n == 0`: The factorial of 0 is 1.
- **Negative numbers or non-integers**: Factorial is not defined for these, so the function now raises a `ValueError`.

Comparison (AI vs Manual Fix)

Feature	AI Fix	Manual Fix
Off-by-one bug fixed	Yes	Yes
Handles 0 correctly	Yes	Yes
Handles negative numbers	Yes	No
Code readability	Good	Simple
Extra validation	Included	Not included

Factorial has important edge cases: factorial(0) should return **1**, and factorial is **not defined for negative numbers**. The AI-corrected version handles both zero and negative inputs properly. The manual fix only corrects the loop error and does not include input validation.

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).

Expected Output:

A well-documented, robust, and readable function that handles errors gracefully.

OUTPUT:

C: > Users > vishn > Untitled-1.py > perform_arithmetic_operation

```python
def perform_arithmetic_operation(operand1, operand2, operator):
    """
    Performs a basic arithmetic operation (addition, subtraction, multiplication, or division)
    on two numbers.
    Args:
        operand1 (int or float): The first number.
        operand2 (int or float): The second number.
        operator (str): The arithmetic operation to perform. Must be one of
            "add", "sub", "mul", or "div".
    Returns:
        int or float: The result of the arithmetic operation.
    Raises:
        TypeError: If operand1 or operand2 are not numbers.
        ValueError: If an invalid operator is provided or if division by zero is attempted.
    """
    if not isinstance(operand1, (int, float)) or not isinstance(operand2, (int, float)):
        raise TypeError("Both operands must be numbers (integers or floats).")

    if operator == "add":
        return operand1 + operand2
    elif operator == "sub":
        return operand1 - operand2
    elif operator == "mul":
        return operand1 * operand2
    elif operator == "div":
        if operand2 == 0:
            raise ValueError("Cannot divide by zero.")
        return operand1 / operand2
    else:
        raise ValueError("Invalid operator. Please use 'add', 'sub', 'mul', or 'div'.")
```

```
5 + 3 = 8
10 - 2 = 8
4 * 6 = 24
10 / 2 = 5.0
```

Improve this Python function to make it more readable. Use better function and parameter names, add a docstring, check inputs, and handle division by zero errors.

I will improve the `calc` function by renaming it to `perform_arithmetic_operation` for better clarity, using more descriptive parameter names (`operand1`, `operand2`, `operator`). I'll also add a comprehensive docstring, include input validation to ensure operands are numbers and the operator is valid, and specifically handle division by zero to prevent errors.

Comparison (Original vs Improved)

Feature	Original Code	Improved Code
Readability	Poor	Clear
Function name	calc	calculate_numbers
Parameter names	a, b, c	num1, num2, operation
Docstring	No	Yes
Error handling	No	Yes
Division-by-zero check	No	Yes
Input validation	No	Yes

EXPLANATION:
The improved function is more readable because it uses descriptive names and a docstring explaining usage. It is more robust because it includes exception handling and input validation. This makes the function safer and easier to maintain.

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

PROMPT:

Check this Python function for PEP8 violations, list the issues, and rewrite it in PEP8-compliant format without changing functionality.

OUTPUT:

C: > Users > vishn > Untitled-1.py > is_prime

```python
import math
def is_prime(n):
    """
    Checks if a given integer is a prime number.
    A prime number is a natural number greater than 1 that has no positive
    divisors other than 1 and itself.
    Args:
        n (int): The integer to check for primality.
    Returns:
        bool: True if n is a prime number, False otherwise.
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False
    return True
print(f"Is 7 prime? {is_prime(7)}")
print(f"Is 10 prime? {is_prime(10)}")
print(f"Is 1 prime? {is_prime(1)}")
print(f"Is 0 prime? {is_prime(0)}")
print(f"Is 2 prime? {is_prime(2)}")
print(f"Is 29 prime? {is_prime(29)}")
print(f"Is 33 prime? {is_prime(33)}")
try:
    is_prime(5.5)
except TypeError as e:
    print(f"Error: {e}")
```

```
Is 7 prime? True
Is 10 prime? False
Is 1 prime? False
Is 0 prime? False
Is 2 prime? True
Is 29 prime? True
Is 33 prime? False
```

The original `Checkprime` function has several issues:

PEP8 Violations:

- **Function Naming (E704, N802):** Function names should be lowercase, with words separated by underscores (`snake_case`). `Checkprime` violates this.
- **Missing Docstring (D103):** The function lacks a docstring, which is essential for explaining its purpose, arguments, and return values.
- **Spacing:** While not a strict PEP8 error here, inconsistent or missing spaces around operators (e.g., `n%i`) can reduce readability. For example, `n % 1 == 0` is preferred over `n % 1 == 0`.

Logical Bug:

- **Incorrect `return True` placement:** The `return True` statement is currently inside the `if n % 1 == 0:` block, *after* `return False`. This means `return True` will never be reached, and the function will never correctly identify a prime number. If the loop completes without finding any divisors (i.e., `n` is prime), the function should return `True` *after* the loop.

Functional Improvements for Primality Test:

- **Edge Cases (0, 1, negatives):** Prime numbers are typically defined as integers greater than 1. The original function doesn't correctly handle `n <= 1`.
- **Efficiency:** For primality tests, the loop only needs to go up to the square root of `n` (i.e., `int(n**0.5) + 1`). This is a common optimization.

I've rewritten the function as `is_prime` to adhere to PEP8 guidelines, fix the logical bug, and handle common edge cases for primality. I've also included an optimization for efficiency and example usage.

Comparison

Feature	Original	Refactored
Naming style	Checkprime	check_prime
Indentation	Incorrect	Correct
Readability	Low	High
PEP8 compliance	No	Yes
Functionality	Correct	Correct

EXPLANATION:

Automated AI code reviews help developers quickly identify formatting issues, naming problems, and style violations. They save time during manual reviews and ensure consistent coding standards across large teams. AI tools improve code readability and maintainability while allowing developers to focus on logic and design.

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.

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]

PROMPT:

Review this Python function for readability, naming, and edge cases. Add type hints, input validation, and make the function more reusable by allowing a configurable multiplier.

OUTPUT:

C: > Users > vishn > Untitled-1.py > ...

```python
from typing import List, Union
def transform_even_numbers(
    numbers: List[Union[int, float]], multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:
    """
    Filters a list of numbers, keeps even numbers, and multiplies them
    by the given multiplier.
    """
    if not isinstance(numbers, list):
        raise TypeError("'numbers' must be a list.")
    if not isinstance(multiplier, (int, float)):
        raise TypeError("'multiplier' must be a number.")

    processed_data: List[Union[int, float]] = []
    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError("All elements must be numbers.")
        if num % 2 == 0:
            processed_data.append(num * multiplier)
    return processed_data
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original list:", my_list)
print("Default multiplier:", transform_even_numbers(my_list))
print("Multiplier = 3:", transform_even_numbers(my_list, 3))
another_list = [10, 20.5, 30, 40.0]
print("Another list:", another_list)
print("Processed:", transform_even_numbers(another_list))
```

```
Original list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Default multiplier: [4, 8, 12, 16, 20]
Multiplier = 3: [6, 12, 18, 24, 30]
Another list: [10, 20.5, 30, 40.0]
Processed: [20, 60, 80.0]
```

Feature	Original	Improved

I will improve the `processData` function by:

- **Renaming** it to `transform_even_numbers` for better clarity, and using more descriptive parameter names (`numbers`, `multiplier`).
- Adding **type hints** for better code readability and maintainability.
- Including **input validation** to ensure `numbers` is an iterable of numeric types and `multiplier` is a number.
- Making the **multiplier configurable**, allowing the function to be more reusable.
- Adding a **docstring** to clearly explain its purpose, arguments, and return value.

Function name	processData	double_even_numbers

Parameter name	d	numbers
Feature	**Original**	**Improved**
Type hints	**No**	**Yes**
Input validation	**No**	**Yes**
Reusability	**No**	**Yes**
Documentation	**No**	**Yes**

EXPLANATION:

AI should be used as an assistant reviewer, not a standalone reviewer. It helps detect readability issues, missing validations, and design improvements quickly. However, human developers are still needed to verify logic, project requirements, and edge cases.

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)
```

PROMPT:

Analyze the time complexity of this Python function and suggest a faster implementation using Python built-in functions. Compare execution time before and after optimization.

OUTPUT:

C: > Users > vishn > Untitled-1.py > transform_even_numbers

```python
from typing import List, Union
def transform_even_numbers(
    numbers: List[Union[int, float]], multiplier: Union[int, float] = 2
) -> List[Union[int, float]]:
    """
    Filters a list of numbers, keeps even numbers, and multiplies them
    by the given multiplier.
    """
    if not isinstance(numbers, list):
        raise TypeError("'numbers' must be a list.")
    if not isinstance(multiplier, (int, float)):
        raise TypeError("'multiplier' must be a number.")
    processed_data: List[Union[int, float]] = []
    for num in numbers:
        if not isinstance(num, (int, float)):
            raise TypeError("All elements must be numbers.")
        if num % 2 == 0:
            processed_data.append(num * multiplier)
    return processed_data
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original list:", my_list)
print("Default multiplier:", transform_even_numbers(my_list))
print("Multiplier = 3:", transform_even_numbers(my_list, 3))
another_list = [10, 20.5, 30, 40.0]
print("Processed:", transform_even_numbers(another_list))
```

```
Original list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Default multiplier: [4, 8, 12, 16, 20]
Multiplier = 3: [6, 12, 18, 24, 30]
Processed: [20, 60, 80.0]
```

Time Complexity Analysis of the Original `sum_of_squares` function

The current `sum_of_squares` function iterates through the input list `numbers` once. For each number in the list:

1. It performs an exponentiation (`num ** 2`). This is generally an O(1) operation.
2. It performs an addition (`total += ...`). This is also generally an O(1) operation.

If `n` is the number of elements in the `numbers` list, the loop runs `n` times. Therefore, the overall time complexity of this function is **O(n)** (linear time complexity).

Comparison of Execution Times

As you can see from the execution times above, the `sum_of_squares_optimized` function, which leverages Python's built-in `sum()` function with a generator expression, is generally faster than the original loop-based implementation.

This is because `sum()` and generator expressions are often implemented in C under the hood, allowing for more efficient execution by reducing Python overhead (like explicit loop control and variable assignments) compared to a manual Python `for` loop.

Comparison

Feature	Original	Optimized
Time complexity	O(n)	O(n)
Readability	Moderate	High
Performance	Slower	Faster
Memory usage	Same	Efficient

TRADEOFF

The optimized version is shorter, cleaner, and slightly faster. Using built-in functions improves performance without reducing readability. However, in some cases highly optimized code (like NumPy vectorization) may be harder for beginners to understand**.**

EXPLANATION:

Both the original and optimized functions have O(n) time complexity because they process each element once.
The optimized version is slightly faster since it uses Python's built-in sum() function with a generator expression, which reduces loop overhead.
It is also shorter and more readable.