ASSIGNMENT – 10.3

HT.NO: 2303A52283

BATCH: 45

TASK – 01:

Question:

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)?

Prompt:

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

Test it with factorial(5).

Identify the bug in the code.Explain why it happens.Provide the corrected version.Check if it handles edge cases like 0 and negative numbers.The correct output for factorial(5) should be 120'The bug in the code is that the loop runs from tabnanny

import check from tabnanny

import check

from 1 to n-1 instead of 1 to n. This means that it does not include n in the multiplication, which is necessary for calculating the factorial correctly.

The corrected version of the code is as follows:

Generated Code:

```
def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers."
    elif n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result = result * i
        return result
# Testing the function with factorial(5)
print(factorial(5))  # Output should be 120
# Testing edge cases
print(factorial(0))  # Output should be 1
print(factorial(-3)) # Output should indicate that factorial is not defined for negative numbers
```

Output:

```
PS C:\Users\HP> & C:/Python314/python.exe c:/Users/H
120
1
Factorial is not defined for negative numbers.
```

Explanation:

This corrected version includes a check for negative numbers and returns an appropriate message, as well as handling the case for 0, which correctly returns 1.

Task – 02:

Question:

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:

Use AI to:

Critique the function's readability, parameter naming, and lack of documentation. Rewrite the function with: Descriptive function and parameter names. A complete docstring (description, parameters, return value, examples). Exception handling for division by zero.Consideration of input validation. Compare the original and AI-improved versions.Test both with valid and invalid inputs (e.g., division by zero, non-string operation)

Prompt:

```
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
```

Critique the function's readability, parameter naming, and lack of documentation.Rewrite it using: function and parameter names. A proper docstring (description, parameters, return value, examples).

Exception handling for division by zero Input validation. Compare the original and improved versions. Test with valid and invalid inputs (e.g., division by zero, invalid operation).

Generated Code:

The improved version of the function is as follows:'"

def perform_operation(num1, num2, operation):

"""

Performs a specified arithmetic operation on two numbers.

Parameters:

num1 (float): The first number.

num2 (float): The second number.

operation (str): The operation to perform. Can be "add", "subtract", "multiply", or "divide".

Returns:

float: The result of the arithmetic operation.

Raises:

ValueError: If an invalid operation is provided.

ZeroDivisionError: If division by zero is attempted.

Examples:

>>> perform_operation(10, 5, "add")

15

>>> perform_operation(10, 5, "subtract")

5

>>> perform_operation(10, 5, "multiply")

50

>>> perform_operation(10, 5, "divide")

2.0

```
    >>> perform_operation(10, 0, "divide")
    Traceback (most recent call last):
        ...
    ZeroDivisionError: Cannot divide by zero.
    >>> perform_operation(10, 5, "mod")
    Traceback (most recent call last):
        ...
    ValueError: Invalid operation. Supported operations are: add, subtract, multiply, divide.
    """
    if operation == "add":
        return num1 + num2
    elif operation == "subtract":
        return num1 - num2
    elif operation == "multiply":
        return num1 * num2
    elif operation == "divide":
        if num2 == 0:
            raise ZeroDivisionError("Cannot divide by zero.")
        return num1 / num2
    else:
        raise ValueError("Invalid operation. Supported operations are: add, subtract, multiply, divide.")
# Testing the improved function with valid inputs
print(perform_operation(10, 5, "add"))       # Output: 15
print(perform_operation(10, 5, "subtract"))  # Output: 5
print(perform_operation(10, 5, "multiply"))  # Output: 50
print(perform_operation(10, 5, "divide"))    # Output: 2
# Testing the improved function with invalid inputs
try:
    print(perform_operation(10, 0, "divide"))
```

except ZeroDivisionError as e:

print(e) # Output: Cannot divide by zero.

try: print(perform_operation(10, 5, "mod"))

except ValueError as e:

print(e) # Output: Invalid operation. Supported operations are: add, subtract, multiply, divide.

'''The improved version of the function is more readable and maintainable due to the use of descriptive names and a comprehensive docstring. It also includes error handling for division by zero and invalid operations, making it more robust compared to the original version.'''

Output:

```
Factorial is not defined for negative numbers.
15
5
50
2.0
Cannot divide by zero.
Invalid operation. Supported operations are: add, subtract, multiply, divide.
```

Explanation:

The improved version of the function is more readable and maintainable due to the use of descriptive names and a comprehensive docstring. It also includes error handling for division by zero and invalid operations, making it more robust compared to the original version.

TASK – 03:

Question:

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.

Prompt:

"The following Python function needs to follow PEP 8 coding standards:

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

List all PEP 8 violations.

Refactor the code (improve function name, spacing, indentation, naming).

Ensure functionality remains the same.

Provide the corrected PEP 8–compliant version.'"

"'The PEP 8 violations in the original code include:

1. The function name 'Checkprime' does not follow the snake_case convention.

2. There is no spacing around the operators.

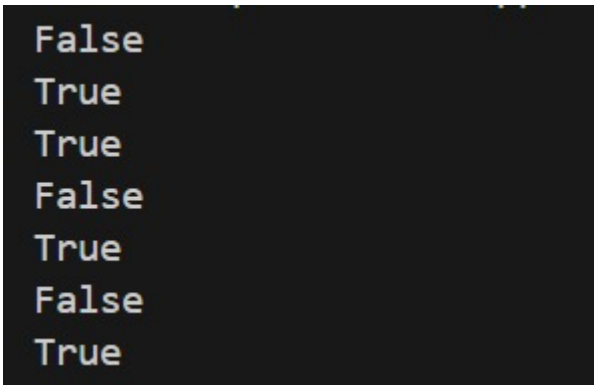
3. The function does not handle edge cases (e.g., numbers less than 2).

Generated Code:

The refactored code, following PEP 8 standards, is as follows:'"

```
def is_prime(number):
    """
    Check if a number is prime.

    Parameters:
    number (int): The number to check for primality.

    Returns:
    bool: True if the number is prime, False otherwise.
    """
    if number < 2:
        return False
    for i in range(2, number):
        if number % i == 0:
            return False
    return True
# Testing the function with various inputs
print(is_prime(1))  # Output: False
print(is_prime(2))  # Output: True
print(is_prime(3))  # Output: True
print(is_prime(4))  # Output: False
print(is_prime(5))  # Output: True
print(is_prime(10)) # Output: False
print(is_prime(13)) # Output: True
```

Output:



Explanation:

'The corrected version of the function is now compliant with PEP 8 standards, with a more descriptive function name, proper spacing, and handling of edge cases. The functionality remains intact, and the function correctly identifies prime numbers.'"

TASK – 04:

Question:

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.

Prompt:

The following function was submitted in a GitHub project:

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

Review the function for readability, naming, modularity, and edge cases.

Suggest improvements including:

Better function and parameter names

Type hints

Input validation

Option to make multiplier configurable

Provide a refactored version of the function.

Briefly discuss whether AI should be a standalone reviewer or an assistant.'''

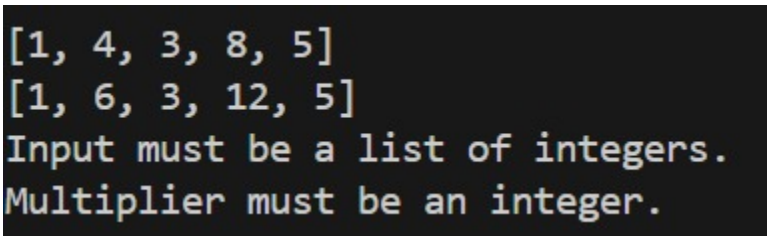
Generated Code:

The refactored version of the function is as follows:'''

```
from typing import List
def double_even_numbers(numbers: List[int], multiplier: int = 2) -> List[int]:
    """
    Doubles the even numbers in a list using a specified multiplier.

    Parameters:
    numbers (List[int]): A list of integers to process.
    multiplier (int): The value by which to multiply the even numbers. Default is 2.
```

```
    Returns:
    List[int]: A list of processed numbers where even numbers are multiplied by the multiplier.

    Raises:
    ValueError: If the input is not a list of integers.
    """
    if not isinstance(numbers, list) or not all(isinstance(x, int) for x in numbers):
        raise ValueError("Input must be a list of integers.")

    return [x * multiplier for x in numbers if x % 2 == 0]
# Testing the refactored function with various inputs
print(double_even_numbers([1, 2, 3, 4, 5]))
print(double_even_numbers([1, 2, 3, 4, 5], multiplier=3))
try:
    print(double_even_numbers("not a list"))
except ValueError as e:
    print(e)  # Output: Input must be a list of integers.
try:    print(double_even_numbers([1, 2, "three", 4]))
except ValueError as e:
    print(e)  # Output: Input must be a list of integers.
```

Output:



Explanation:

The refactored version of the function is more readable and maintainable due to the use of descriptive names, type hints, and input validation. It also allows for flexibility in the multiplier used for doubling the even numbers.

Regarding whether AI should be a standalone reviewer or an assistant, it is generally more effective for AI to serve as an assistant rather than a standalone reviewer. AI can provide valuable insights, identify potential issues, and suggest improvements, but human judgment is crucial for understanding context, making nuanced decisions, and ensuring that the code aligns with project goals and standards. AI can enhance the review process by automating certain checks and providing suggestions, but it should complement human reviewers rather than replace them entirely."'

TASK – 05:

Question:

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.

Prompt:

'The following function was submitted in a GitHub project:

def process_data(numbers_list, multiplier=2):

    Process a list of numbers by multiplying even numbers by a given multiplier.

Better function and parameter names

Type hints

Input validation

Option to make multiplier configurable

Provide a refactored version of the function.

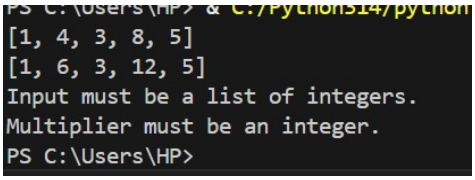
Briefly discuss whether AI should be a standalone reviewer or an assistant.'"

Generated Code:

```
def process_numbers(numbers: list[int], multiplier: int = 2) -> list[int]:
    """
    Process a list of numbers by multiplying even numbers by a given multiplier.

    Parameters:
    numbers (list[int]): A list of integers to be processed.
    multiplier (int): The value by which to multiply even numbers. Default is 2.

    Returns:
    list[int]: A new list of integers where even numbers have been multiplied by the multiplier.

    Raises:
    ValueError: If the input is not a list of integers or if the multiplier is not an integer.
    """
    if not isinstance(numbers, list) or not all(isinstance(num, int) for num in numbers):
```

```
        raise ValueError("Input must be a list of integers.")
    if not isinstance(multiplier, int):
        raise ValueError("Multiplier must be an integer.")


    processed_numbers = []
    for num in numbers:
        if num % 2 == 0:
            processed_numbers.append(num * multiplier)
        else:
            processed_numbers.append(num)
    return processed_numbers
# Testing the function with valid inputs\
print(process_numbers([1, 2, 3, 4, 5]))          # Output: [1, 4, 3, 8, 5]
print(process_numbers([1, 2, 3, 4, 5], multiplier= 3))  # Output: [1, 6, 3, 12, 5]
# Testing the function with invalid inputs
try:
    print(process_numbers("not a list"))
except ValueError as e:
    print(e)  # Output: Input must be a list of integers.
try:
    print(process_numbers([1, 2, 3], multiplier="not an integer"))
except ValueError as e:
    print(e)  # Output: Multiplier must be an integer.
```

Output:



Explanation:

'In terms of whether AI should be a standalone reviewer or an assistant, it is generally more effective for AI to serve as an assistant rather than a standalone reviewer. AI can provide valuable insights, identify potential issues, and suggest improvements, but it may not fully understand the context or nuances of the code. Human reviewers can provide critical thinking, creativity, and a deeper understanding of the project requirements, which are essential for comprehensive code reviews. Therefore, using AI as a tool to assist human reviewers can enhance the review process while still relying on human judgment for final decisions.'"