AI Assisted Coding

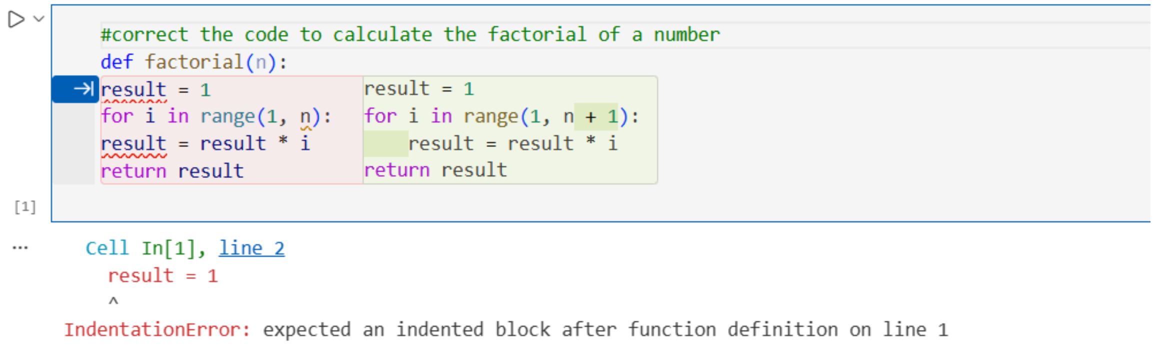
Name : Vineeth-Chidurala **Date :** 25-02-2026

Ht.No. : 2303A52447

Task - 1 : AI-Assisted Bug Detection
Scenario: A junior developer wrote the following Python function to calculate factorials:

Prompt : correct the code to calculate the factorial of a number

Screenshots :

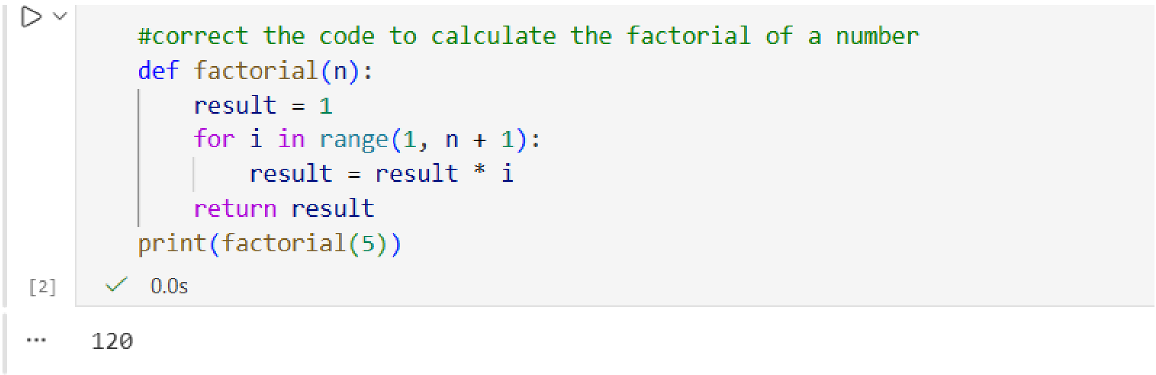


```
#correct the code to calculate the factorial of a number
def factorial(n):
result = 1
for i in range(1, n):
result = result * i
return result
```

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

```
Cell In[1], line 2
  result = 1
  ^
IndentationError: expected an indented block after function definition on line 1
```

Output :



```
#correct the code to calculate the factorial of a number
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result = result * i
    return result
print(factorial(5))
```

```
120
```

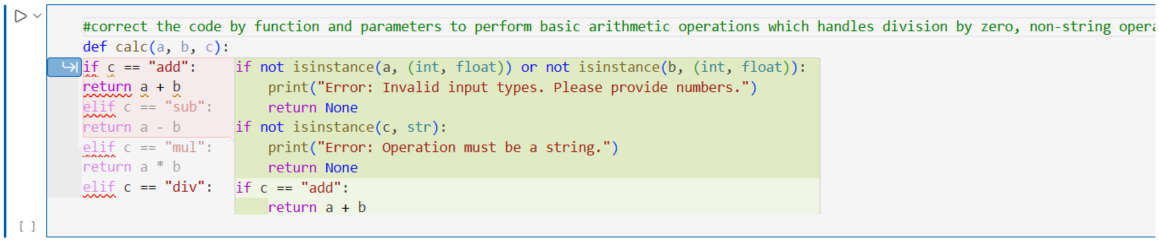
Task – 2 : Improving Readability &

Documentation

Scenario:The following code works but is poorly written:

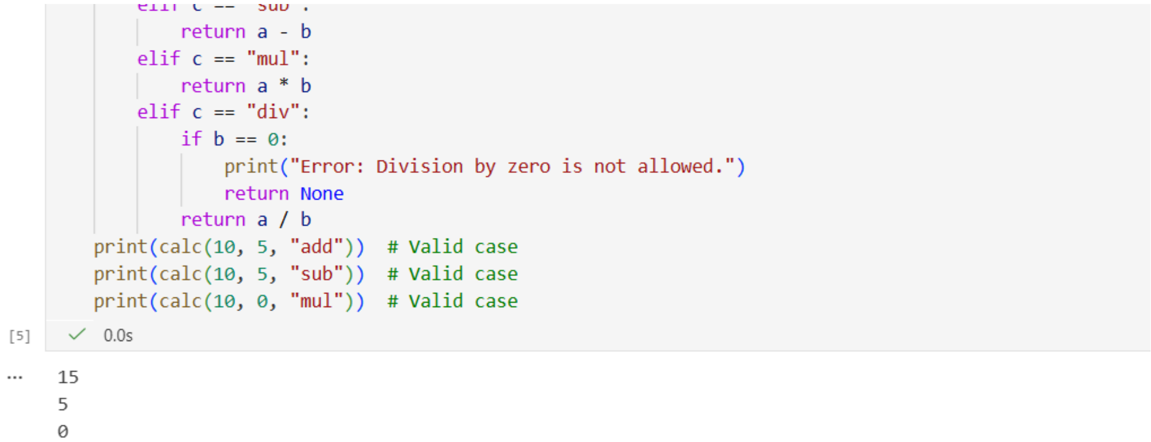
Prompt : correct the code by function and parameters to perform basic arithmetic operations which handles division by zero, non-string operation.print output for vaild and invalid cases

ScreenShots :



```
#correct the code by function and parameters to perform basic arithmetic operations which handles division by zero, non-string opera
def calc(a, b, c):
if c == "add":      if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
return a + b            print("Error: Invalid input types. Please provide numbers.")
elif c == "sub":        return None
return a - b        if not isinstance(c, str):
elif c == "mul":        print("Error: Operation must be a string.")
return a * b            return None
elif c == "div":    if c == "add":
                        return a + b
[ ]
```

Output :



```
    elif c == "sub":
        return a - b
    elif c == "mul":
        return a * b
    elif c == "div":
        if b == 0:
            print("Error: Division by zero is not allowed.")
            return None
        return a / b
print(calc(10, 5, "add"))  # Valid case
print(calc(10, 5, "sub"))  # Valid case
print(calc(10, 0, "mul"))  # Valid case
```

[5] ✓ 0.0s

```
15
5
0
```

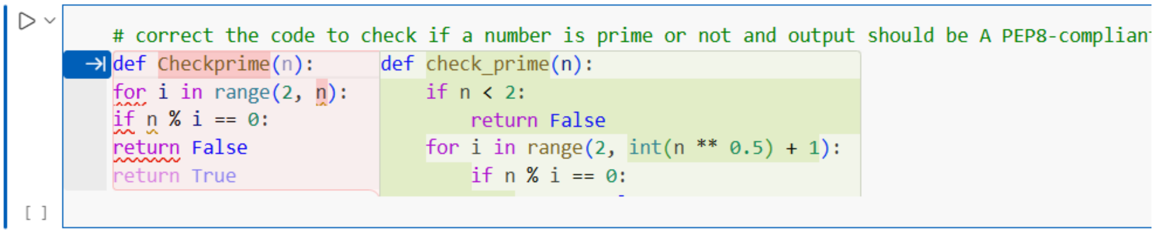
Task-3 : Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer

submits:

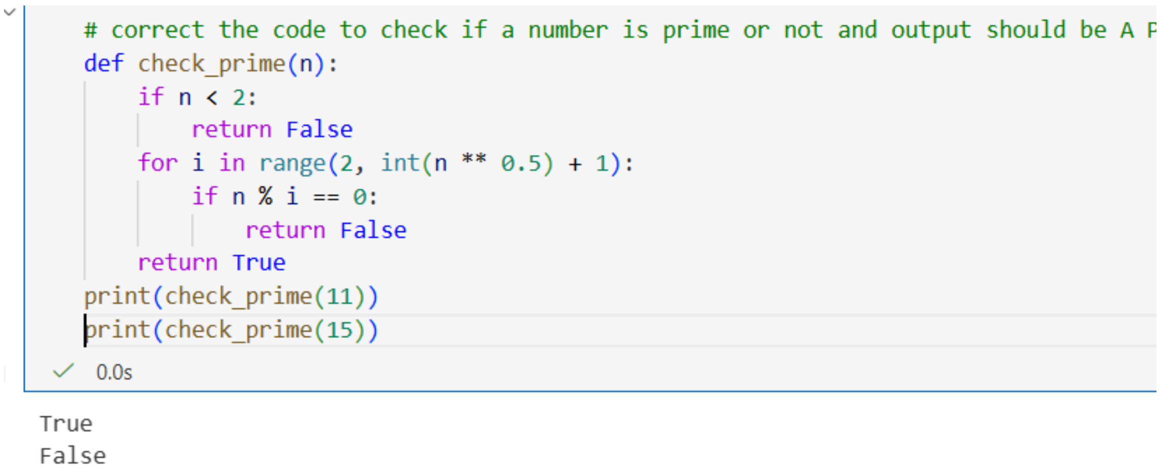
Prompt : correct the code to check if a number is prime or not and output should be A PEP8-compliant version of the function

ScreenShots :



```
# correct the code to check if a number is prime or not and output should be A PEP8-complian
def Checkprime(n):      def check_prime(n):
for i in range(2, n):       if n < 2:
if n % i == 0:                  return False
return False                for i in range(2, int(n ** 0.5) + 1):
return True                     if n % i == 0:
[ ]
```

Output :



```
# correct the code to check if a number is prime or not and output should be A P
def check_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
print(check_prime(11))
print(check_prime(15))
```

✓ 0.0s

```
True
False
```

Task-4 : AI as a Code Reviewer in Real Projects

Scenario:

In a GitHub project, a teammate submits:

Prompt : Act as a professional Python code reviewer.

Review the following function:

def processData(d):

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

1. Critique its readability, naming, reusability, and edge cases.

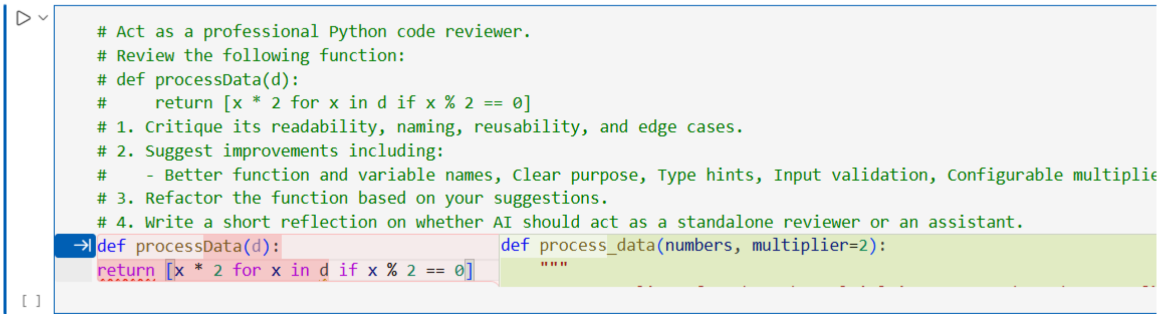
2. Suggest improvements including:

  - Better function and variable names, Clear purpose, Type hints, Input validation, Configurable multiplier instead of hardcoded 2

3. Refactor the function based on your suggestions.

4. Write a short reflection on whether AI should act as a standalone reviewer or an assistant.

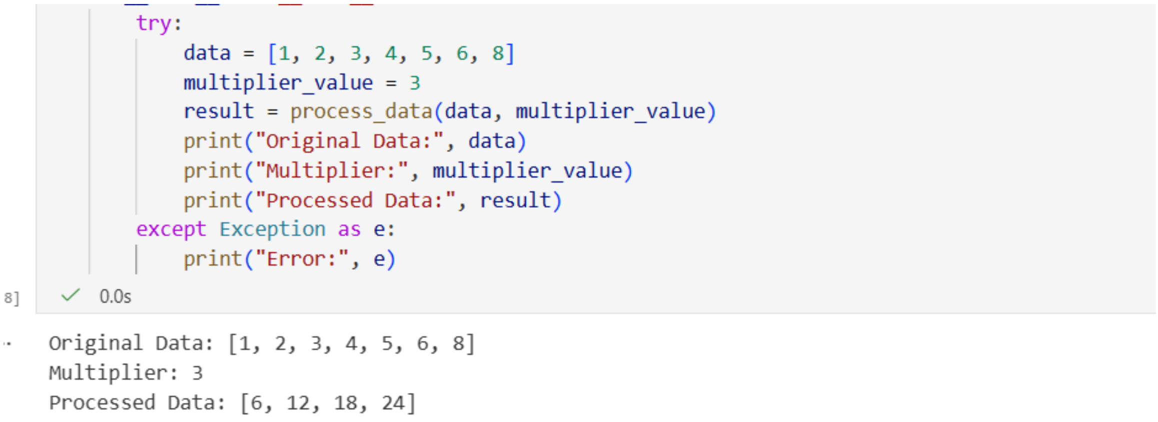
Screenshots :



```
# Act as a professional Python code reviewer.
# Review the following function:
# def processData(d):
#     return [x * 2 for x in d if x % 2 == 0]
# 1. Critique its readability, naming, reusability, and edge cases.
# 2. Suggest improvements including:
#    - Better function and variable names, Clear purpose, Type hints, Input validation, Configurable multiplie
# 3. Refactor the function based on your suggestions.
# 4. Write a short reflection on whether AI should act as a standalone reviewer or an assistant.
def processData(d):
return [x * 2 for x in d if x % 2 == 0]
```

```
def process_data(numbers, multiplier=2):
    """
```

Output :



```
    try:
        data = [1, 2, 3, 4, 5, 6, 8]
        multiplier_value = 3
        result = process_data(data, multiplier_value)
        print("Original Data:", data)
        print("Multiplier:", multiplier_value)
        print("Processed Data:", result)
    except Exception as e:
        print("Error:", e)
```

```
Original Data: [1, 2, 3, 4, 5, 6, 8]
Multiplier: 3
Processed Data: [6, 12, 18, 24]
```

Task-5 : AI-Assisted Performance Optimization

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

Prompt : Act as a Python performance optimization expert.

Given the function:

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

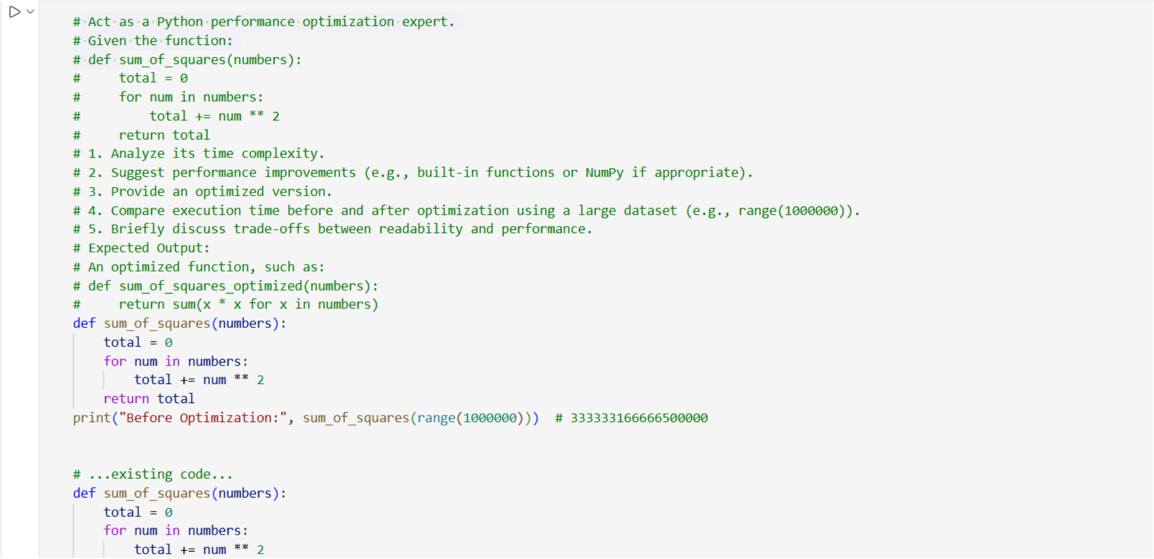
1. Analyze its time complexity.
2. Suggest performance improvements (e.g., built-in functions or NumPy if appropriate).
3. Provide an optimized version.
4. Compare execution time before and after optimization using a large dataset (e.g., range(1000000)).
5. Briefly discuss trade-offs between readability and performance.

Expected Output:

An optimized function, such as:

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

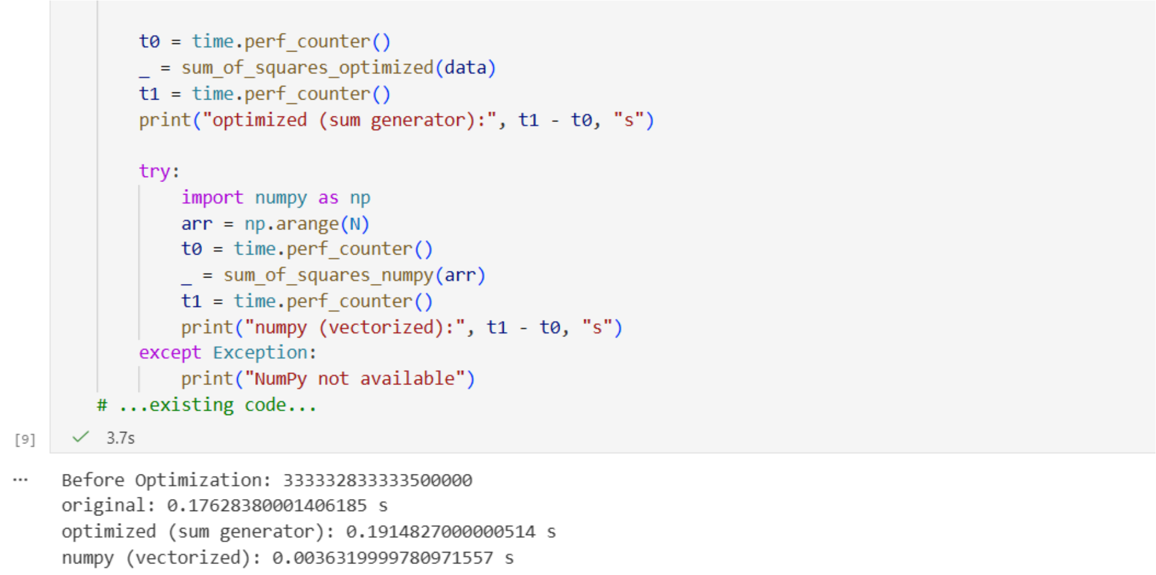
Screenshots :



```
# Act as a Python performance optimization expert.
# Given the function:
# def sum_of_squares(numbers):
#     total = 0
#     for num in numbers:
#         total += num ** 2
#     return total
# 1. Analyze its time complexity.
# 2. Suggest performance improvements (e.g., built-in functions or NumPy if appropriate).
# 3. Provide an optimized version.
# 4. Compare execution time before and after optimization using a large dataset (e.g., range(1000000)).
# 5. Briefly 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)
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
    return total
print("Before Optimization:", sum_of_squares(range(1000000)))  # 333333166666500000


# ...existing code...
def sum_of_squares(numbers):
    total = 0
    for num in numbers:
        total += num ** 2
```

Output :



```
    t0 = time.perf_counter()
    _ = sum_of_squares_optimized(data)
    t1 = time.perf_counter()
    print("optimized (sum generator):", t1 - t0, "s")

    try:
        import numpy as np
        arr = np.arange(N)
        t0 = time.perf_counter()
        _ = sum_of_squares_numpy(arr)
        t1 = time.perf_counter()
        print("numpy (vectorized):", t1 - t0, "s")
    except Exception:
        print("NumPy not available")
# ...existing code...
```

```
Before Optimization: 333332833333500000
original: 0.17628380001406185 s
optimized (sum generator): 0.1914827000000514 s
numpy (vectorized): 0.0036319999780971557 s
```