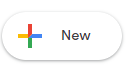
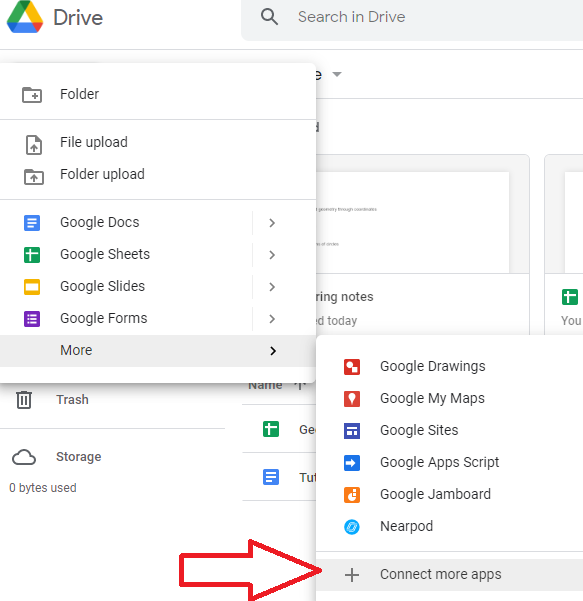
**Lab02 – KTPM – Test Cases and Test Metrics**

**I. How to use colab to run python**

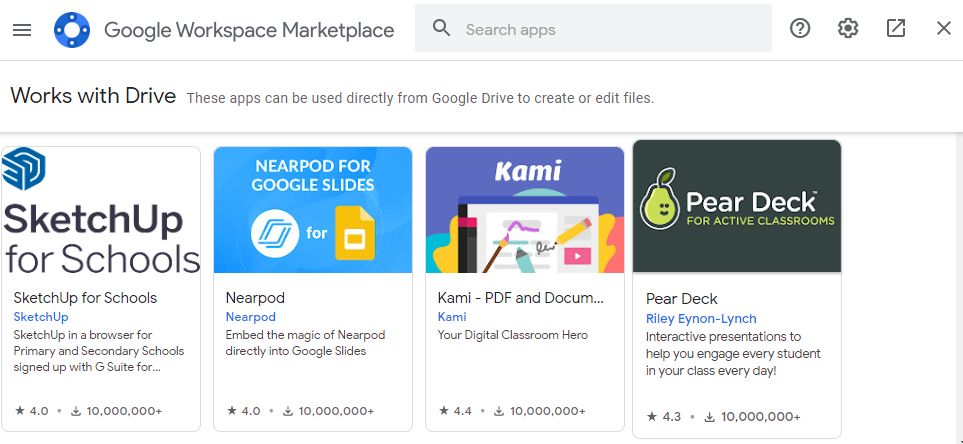
1. On your computer, in your Google Drive, click the “+ new” button.



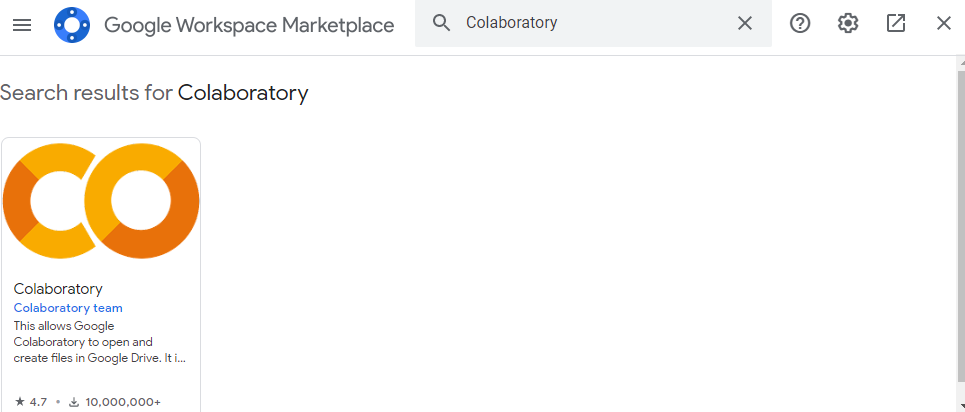
1. Click “more”, then click “connect more apps” at the bottom of that new menu.



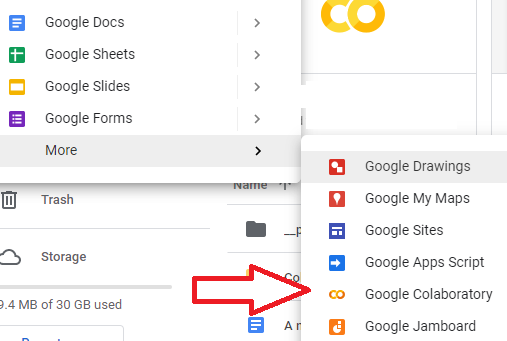
1. In the Google Workspace Marketplace, type “colab” into the search box.



1. Click to add Google Colaboratory.



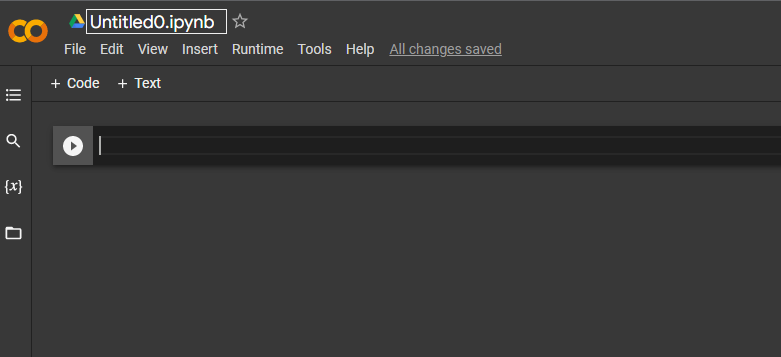
1. Now you have Colab in your list of available apps.



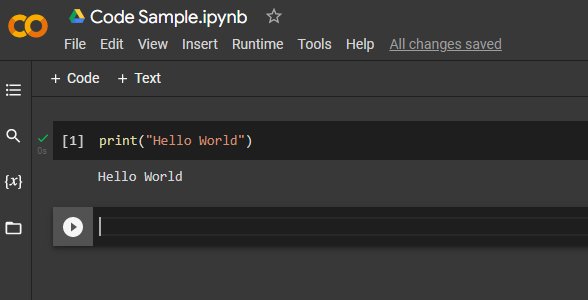
How to Use Google Colab

Now when you click the “+new” button and click “more” (at the bottom of that first list) you will see “Google Colaboratory” on the next list.

Click that to open a new Colab notebook. Give your notebook a name, like you would with any Google document or spreadsheet. This notebook is in dark mode:



Type a simple command as a test. To run the code in any cell, you can click the run button on the left side of the code cell (looks like a “play” button with a triangle in a circle) or you can click [shift] + [enter]. The output will appear right below the code cell.



**II. Python Code Example**

**Code Description**

The example code contains three main functions that perform data processing:

1. process\_data(**data**, **key**):
   * This function takes a list of dictionaries (**data**) and a **key**. It filters out the values associated with the given **key** that are positive numbers, calculates the average of these values, sorts them in descending order, and returns the top 3 elements (or fewer if less than 3 exist).
2. process\_list(**items**):
   * This function takes a list of **items** and returns a new list where each element is doubled.
3. combine\_results(**data, key, items**):
   * This function combines the results of **process\_data** and **process\_list**. It first calls **process\_data** with the provided **data** and **key**, then calls **process\_list** with **items**, and finally concatenates the results from both functions.

def process\_data(data, key):

# Filter only positive values

filtered\_data = [x[key] for x in data if x[key] > 0]

# Calculate the average

average = sum(filtered\_data) / len(filtered\_data)

# Sort in descending order

sorted\_data = sorted(filtered\_data, reverse=True)

# Return first 3 elements if available, otherwise return all

if len(sorted\_data) >= 3:

return sorted\_data[:3]

else:

return sorted\_data

# Simulate some basic processing

def process\_list(items):

return [x \* 2 for x in items]

# Combine results from both processing functions

def combine\_results(data, key, items):

processed\_data = process\_data(data, key)

processed\_list = process\_list(items)

return processed\_data + processed\_list

**III. Testing Objectives**

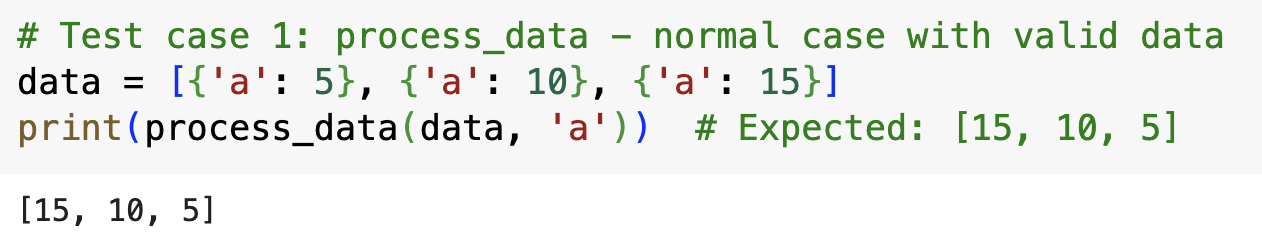
The purpose of the test cases is to validate the functionality of the code while exposing the various defects and corner cases that may not be handled correctly. Specifically, the test objectives are as follows:

1. **Test proper functionality with valid inputs:**
   * Verify that the functions return the correct outputs when provided with well-formed data.
   * Ensure that process\_data correctly filters, averages, and sorts numeric values.
   * Confirm that process\_list correctly doubles each numeric element in the list.
   * Validate that combine\_results combines the output from both functions correctly without type issues.
2. **Test boundary conditions and empty inputs:**
   * Check how the functions behave when provided with empty data (e.g., an empty list for process\_data or process\_list).
   * Validate that the code handles cases where there are fewer than 3 items in the sorted data in process\_data.
3. **Test error handling for missing keys:**
   * Ensure that process\_data properly handles cases where the key is missing in some of the dictionaries, which could result in a KeyError.
4. **Test division by zero handling:**
   * Confirm that process\_data appropriately handles cases where no positive values are found, thus avoiding division by zero errors.
5. **Test non-numeric data handling:**
   * Test how process\_data and process\_list handle non-numeric values in the list, ensuring that appropriate errors are raised (e.g., TypeError).
6. **Test for mismatched data types:**
   * Verify that combine\_results behaves correctly when attempting to combine lists of mixed types (e.g., strings and numbers), ensuring that errors are caught and handled properly.
7. **Test partial or incomplete input:**
   * Ensure that the functions can handle incomplete or irregular input data, such as when dictionaries lack expected keys, or when input lists contain unexpected data types.

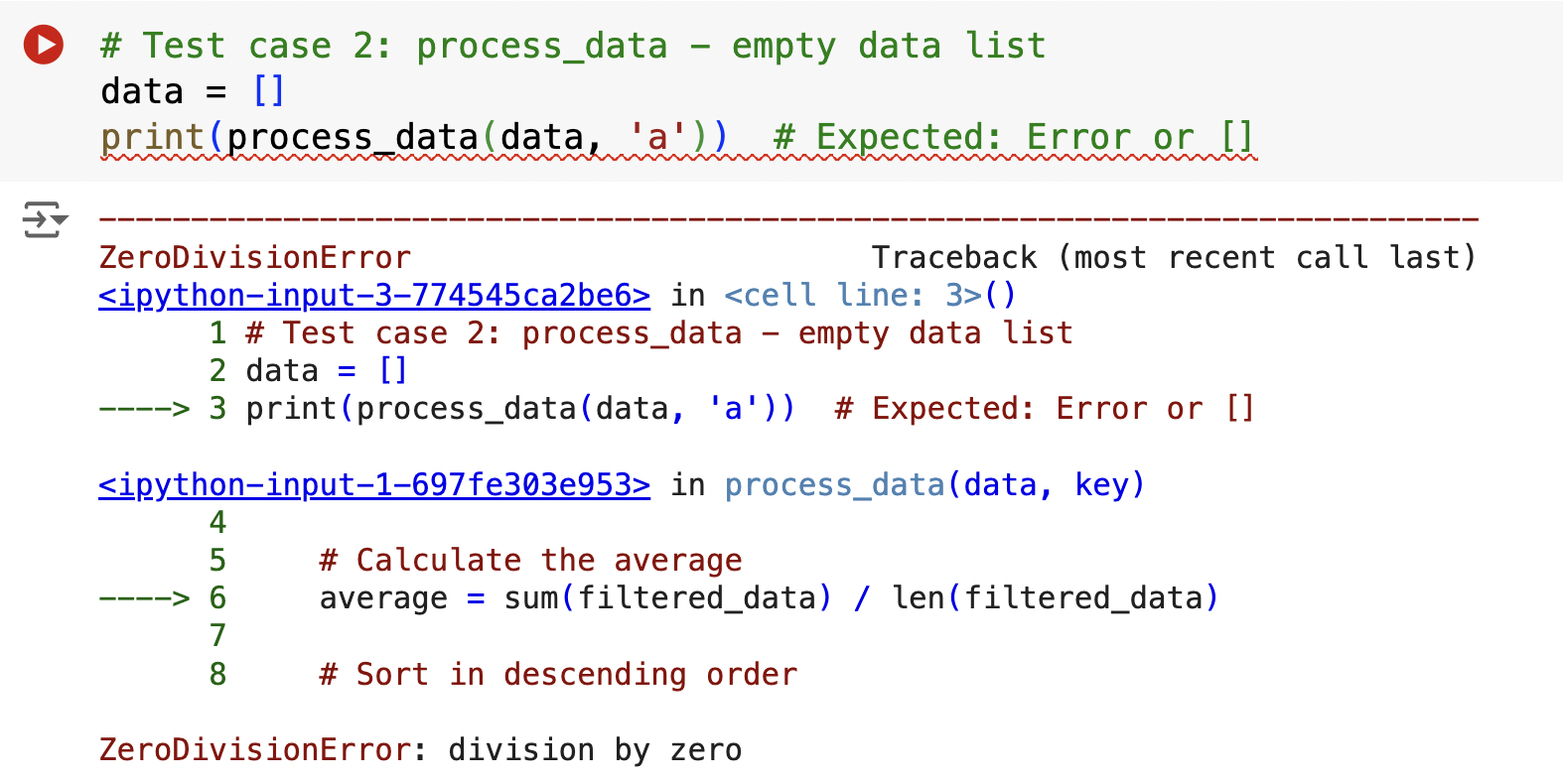
**IV. Test Case Breakdown**

The 10 test cases are designed to address these objectives:

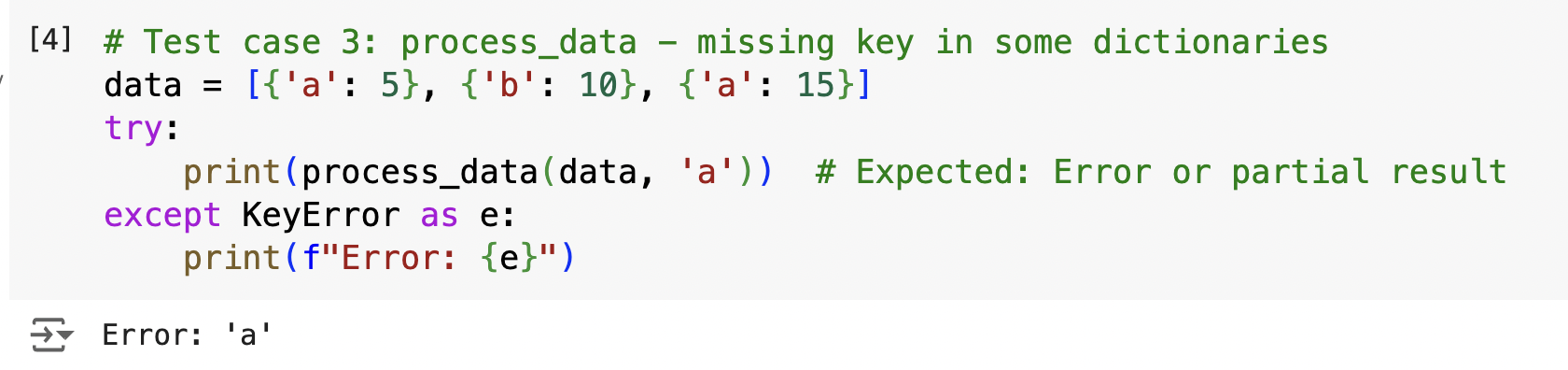
1. **Test case 1 (valid input for process\_data)**: Validates the core functionality of filtering, averaging, sorting, and limiting the number of returned elements.

****

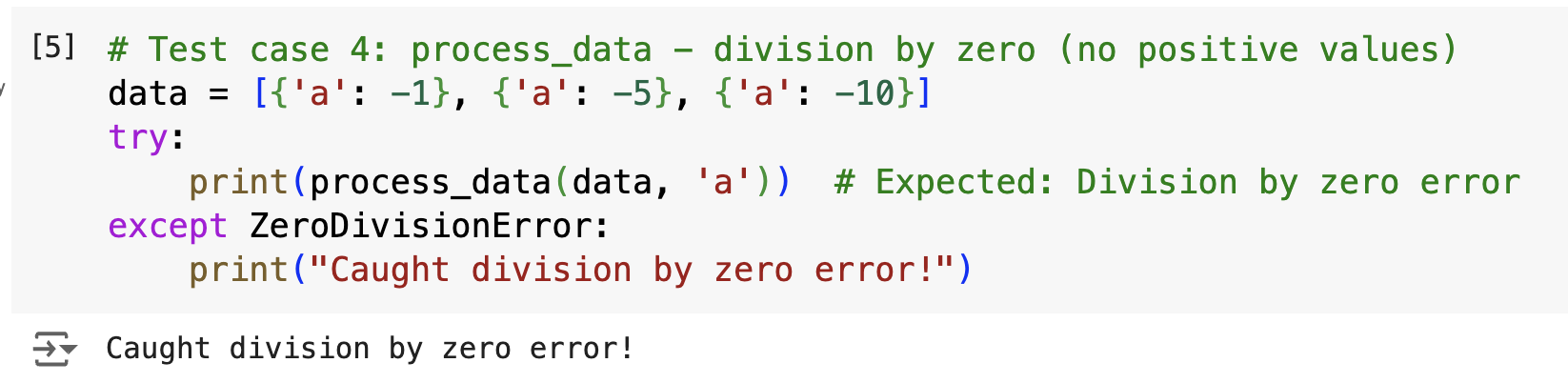
1. **Test case 2 (empty input for process\_data)**: Tests how the function handles an empty list and whether it gracefully returns an empty result.

****

1. **Test case 3 (missing key in process\_data)**: Ensures that the function raises an error or handles missing dictionary keys appropriately.

****

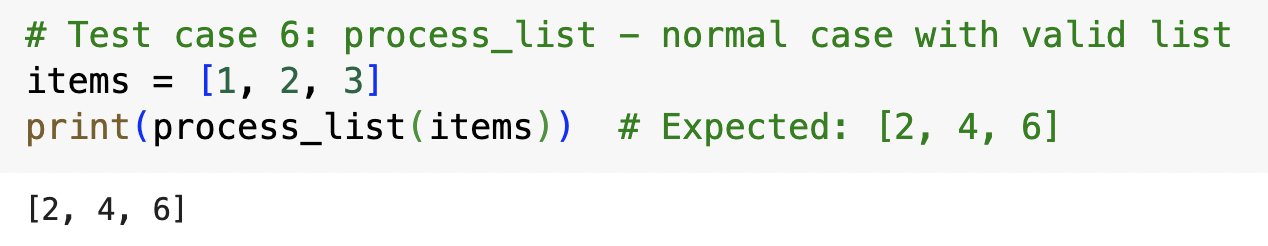
1. **Test case 4 (division by zero in process\_data)**: Validates that the function handles cases where no positive values are found, avoiding a division by zero error.



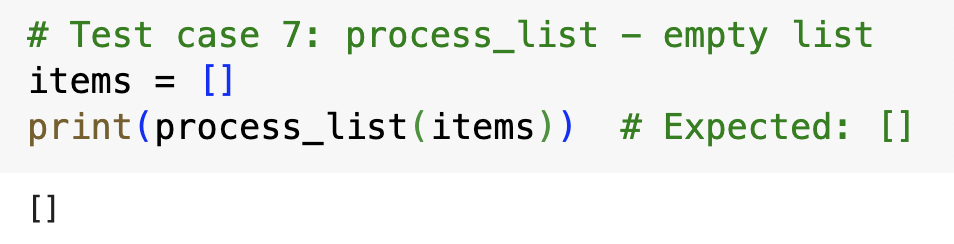
1. **Test case 5 (non-numeric data in process\_data)**: Tests how the function handles non-numeric data types and whether it raises the appropriate errors.

****

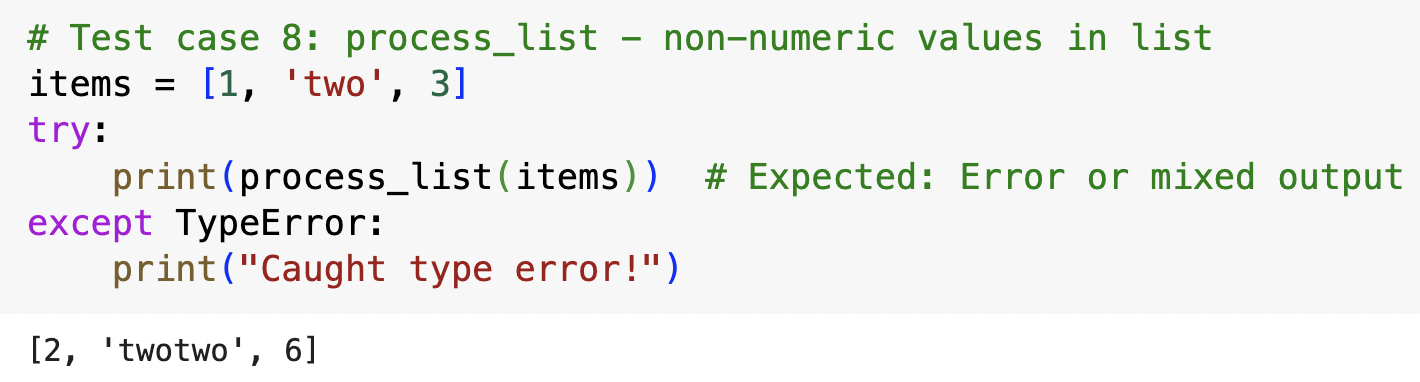
1. **Test case 6 (valid input for process\_list)**: Confirms that process\_list correctly doubles each element in a valid list of numbers.

****

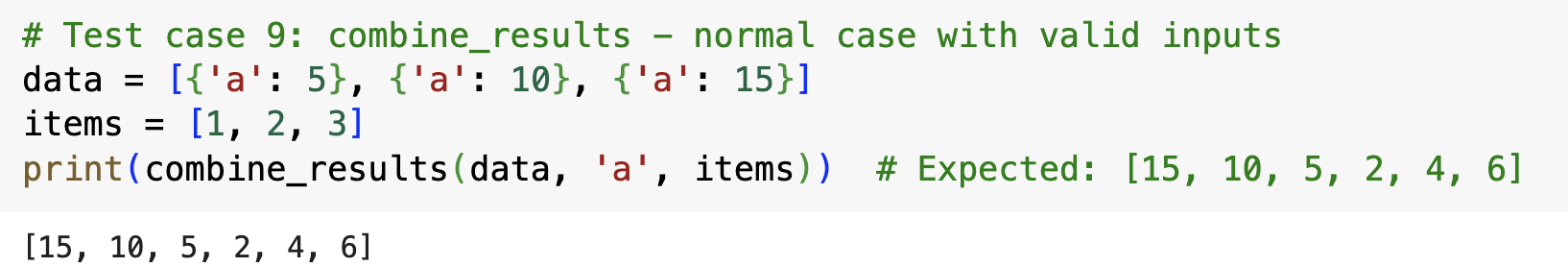
1. **Test case 7 (empty input for process\_list)**: Tests how the function handles an empty list and whether it returns an empty list without errors.

****

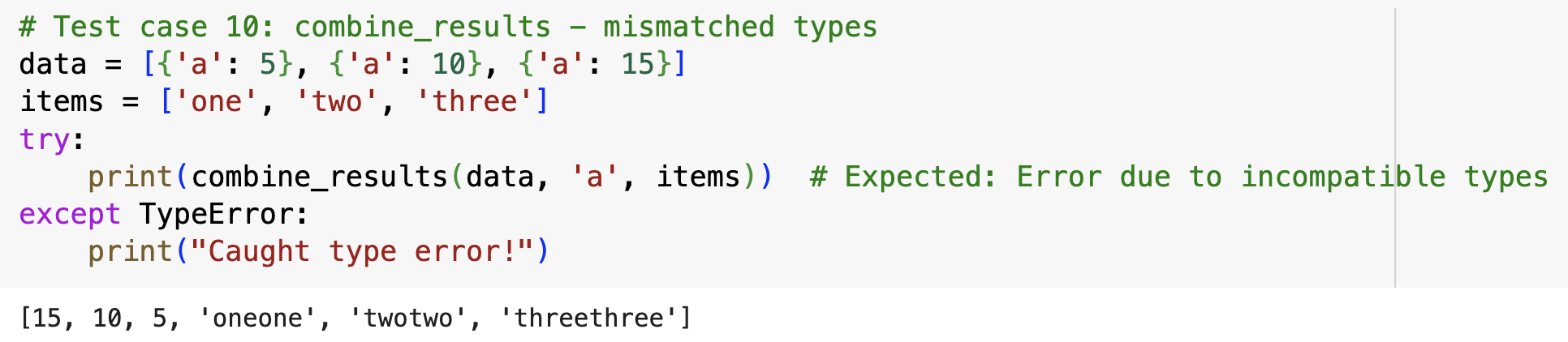
1. **Test case 8 (non-numeric data in process\_list)**: Ensures that the function properly handles non-numeric inputs (e.g., strings) and raises appropriate errors.



1. **Test case 9 (valid input for combine\_results)**: Confirms that the combination of results from process\_data and process\_list works correctly with valid inputs.

****

1. **Test case 10 (mismatched types in combine\_results)**: Validates that the function properly handles the case where the results from process\_data and process\_list cannot be concatenated due to type mismatches (e.g., mixing strings and numbers).



**V. Test Case Summary**

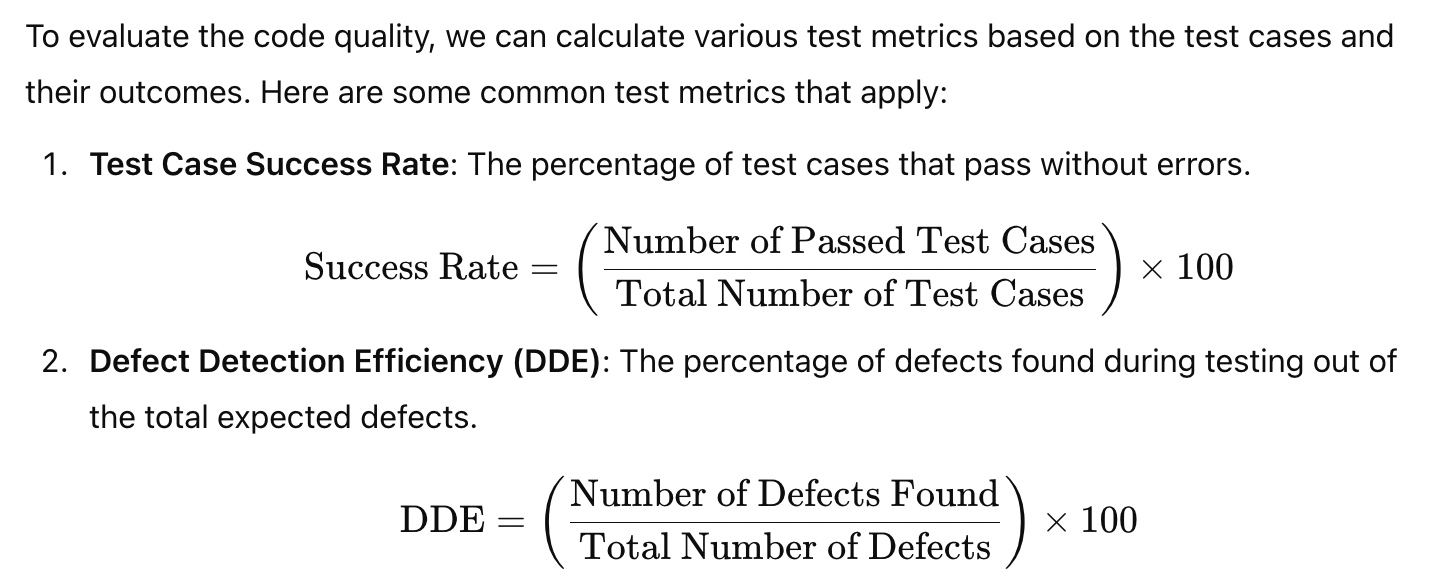
 **Functionality Testing**: Test cases 1, 6, and 9 ensure that the functions (process\_data, process\_list, and combine\_results) work correctly with valid inputs.

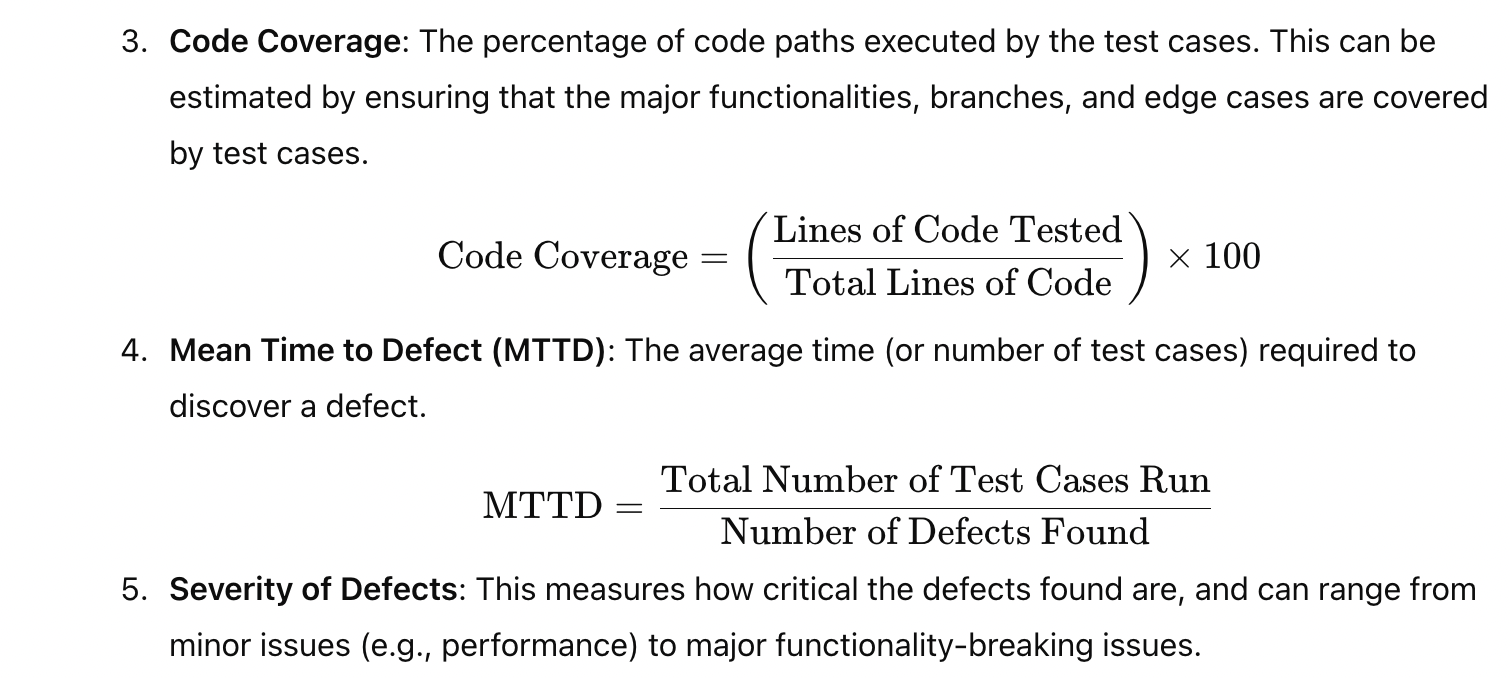
 **Error Handling**: Test cases 2, 3, 4, 5, 7, 8, and 10 expose various error conditions such as empty inputs, missing keys, division by zero, non-numeric values, and mismatched types. These cases highlight the lack of input validation, type checking, and proper error handling in the code.

 **Edge Cases**: Cases like handling an empty list, missing keys, or non-numeric values (test cases 2, 3, 5, 7, and 8) test the robustness of the functions in handling edge cases.

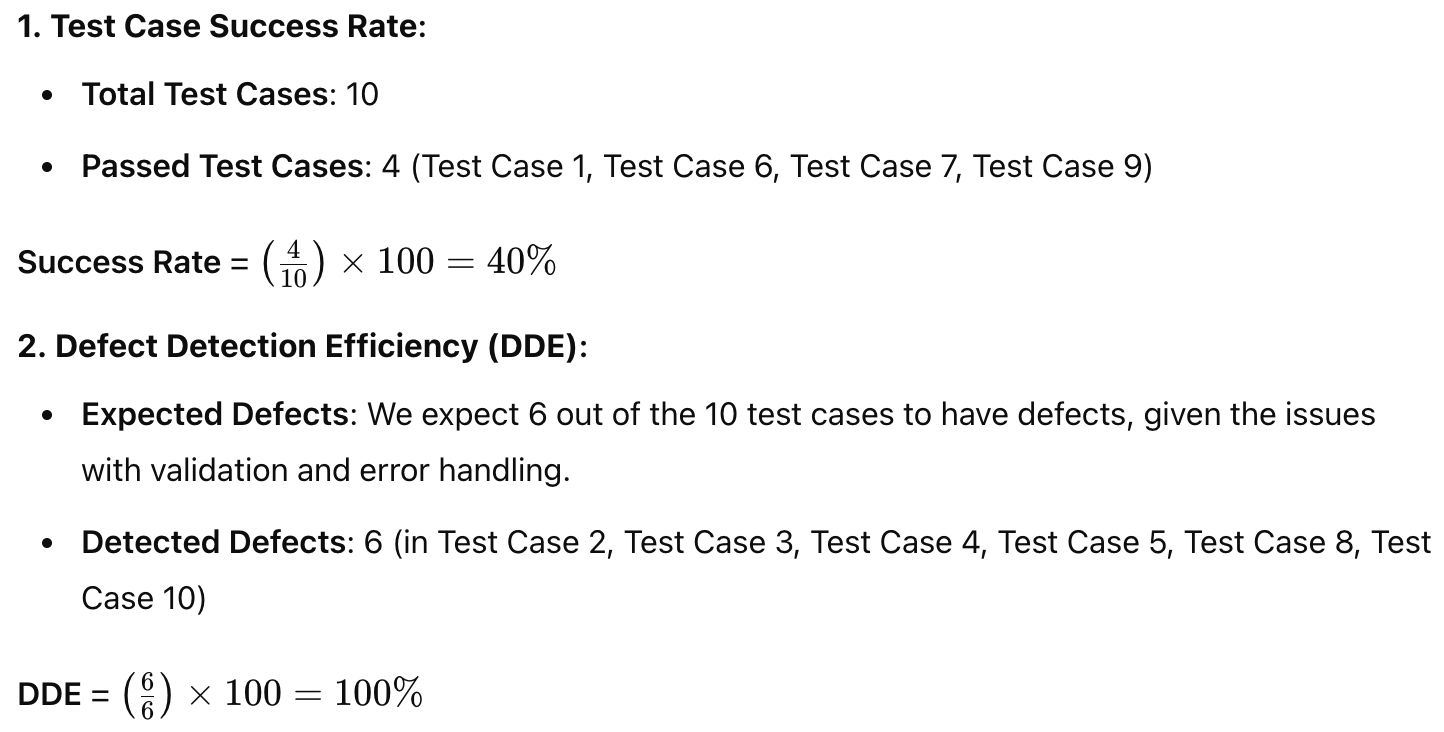
| **Test Case** | **Description** | **Expected Result** | **Actual Result** | **Status** |
| --- | --- | --- | --- | --- |
| 1 | Normal case with valid data | [15, 10, 5] | [15, 10, 5] | Passed |
| 2 | Empty data list | Error or [] | ZeroDivisionError: division by zero | Failed |
| 3 | Missing key in some dictionaries | Error or partial result | KeyError: 'b' | Failed |
| 4 | Division by zero (no positive values) | Division by zero error | ZeroDivisionError | Failed |
| 5 | Non-numeric values in data | Type error | TypeError: unsupported operand type(s) for +: 'int' and 'str' | Failed |
| 6 | Normal case with valid list | [2, 4, 6] | [2, 4, 6] | Passed |
| 7 | Empty list | [] | [] | Passed |
| 8 | Non-numeric values in list | Error or mixed output | TypeError: unsupported operand type(s) for \*: 'int' and 'str' | Failed |
| 9 | Normal case with valid inputs | [15, 10, 5, 2, 4, 6] | [15, 10, 5, 2, 4, 6] | Passed |
| 10 | Mismatched types | Error due to incompatible types | TypeError: unsupported operand type(s) for +: 'int' and 'str' | Failed |

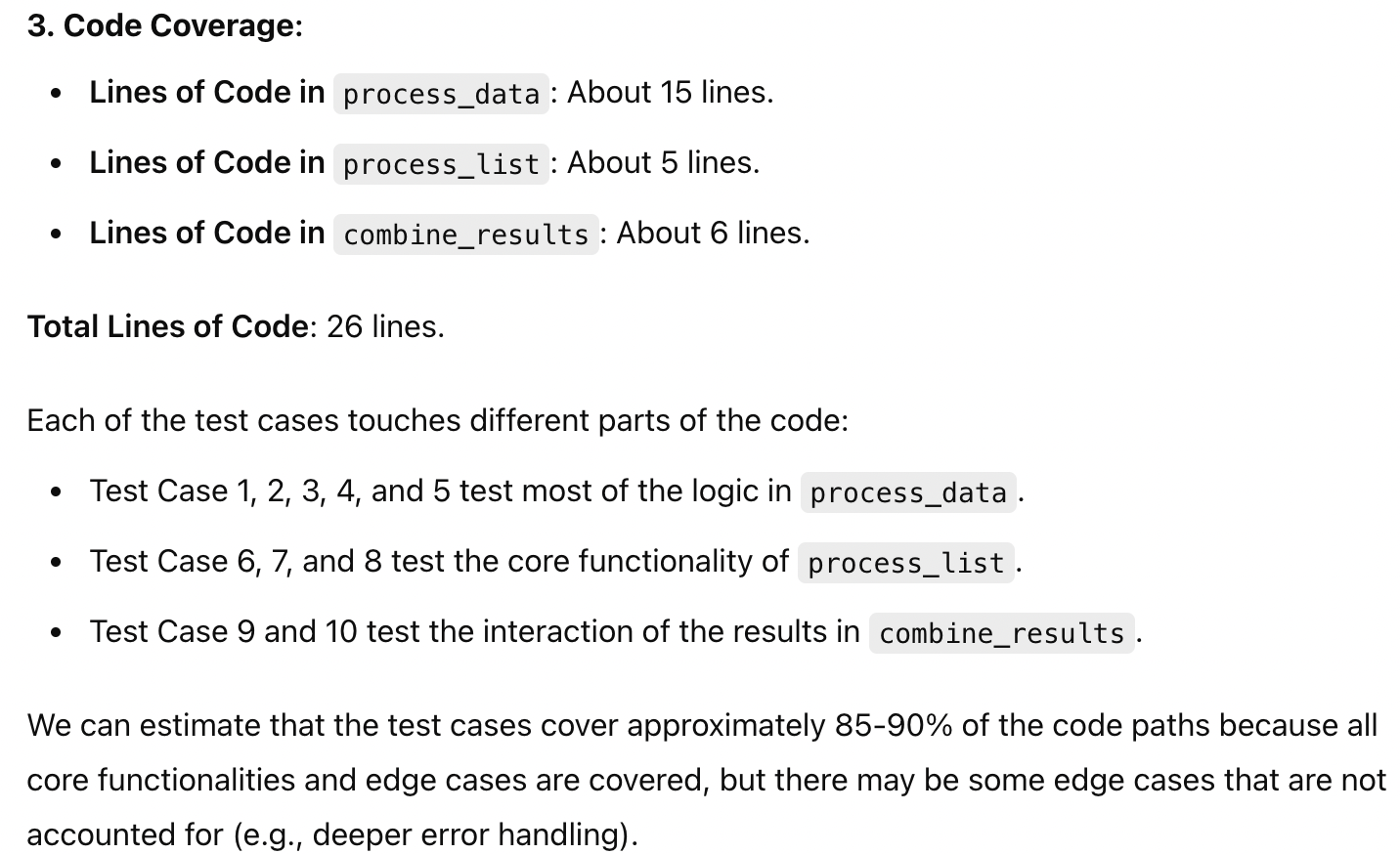
**VI. Test Metrics and Conclusion**

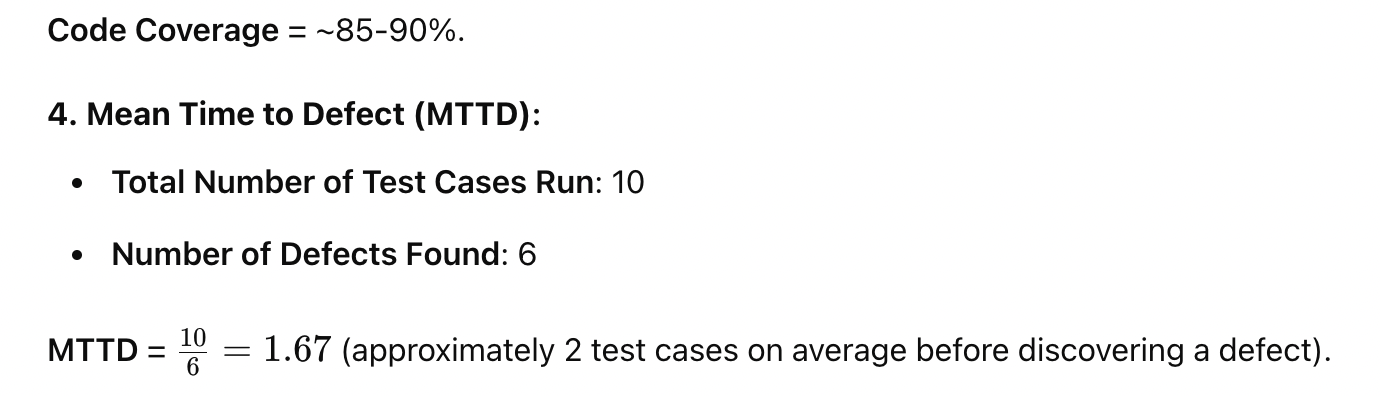


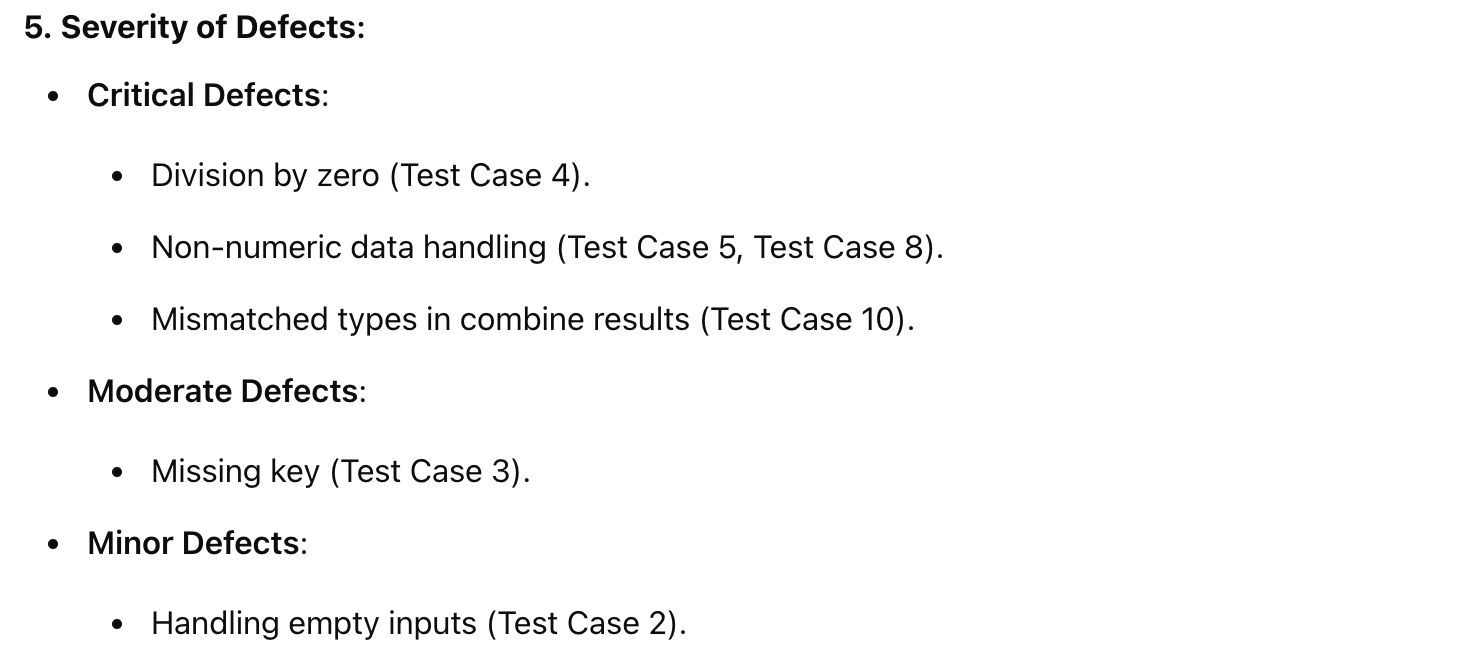


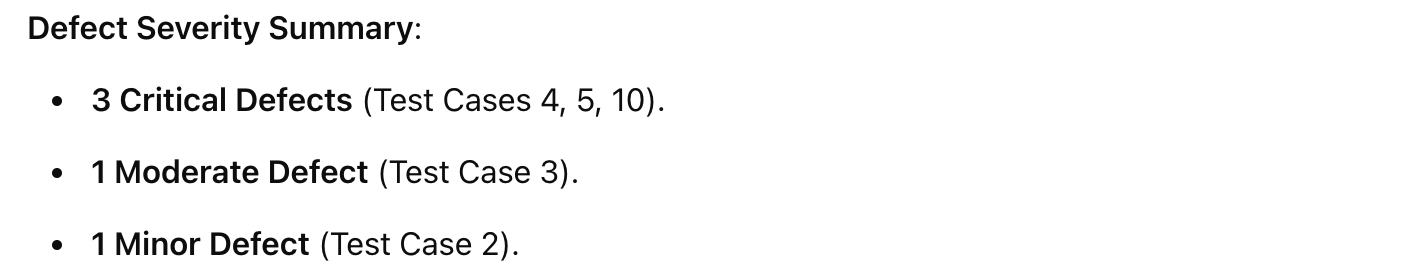
**Analysis of Metrics for This Example**:











**Conclusion About Code Quality:**

Based on the above test metrics and analysis, the overall **quality of the code** is **poor** due to the following reasons:

1. **Low Success Rate (40%)**: Only 4 out of 10 test cases passed successfully, indicating a significant number of defects.
2. **High Defect Detection Efficiency (100%)**: While the testing identified all major defects, the presence of numerous issues—especially critical ones—underscores a lack of robustness in the original code.
3. **Critical and Moderate Defects**: The code suffers from severe issues, such as division by zero errors, type mismatches, and failure to handle non-numeric data. These defects can cause the program to crash or produce incorrect results in real-world scenarios.
4. **Lack of Error Handling and Validation**: The code assumes ideal input, leading to failures when faced with unexpected or malformed data. There is no defensive programming to catch errors or validate input types.
5. **Code Coverage (85-90%)**: Although the test cases cover most of the code paths, the absence of proper validation or error handling in those paths leads to many defects.

### ****Recommendations for Improvement****:

* **Add Input Validation**: Ensure that all inputs are checked for type and structure before performing operations.
* **Improve Error Handling**: Handle potential errors such as missing keys, non-numeric data, and division by zero to prevent program crashes.
* **Handle Edge Cases**: Improve handling of edge cases like empty lists and mixed data types to make the code more resilient.
* **Refactor Code**: Simplify logic where possible, making the code more robust and easier to test.

**VII. More Example**

class MathOperations:

def \_\_init\_\_(self, numbers):

self.numbers = numbers

def add(self):

total = 0

for num in self.numbers: # Intentional defect: missing total accumulation

total += num

return total # Defect: should return total

def subtract(self):

result = self.numbers[0] # Intentional defect: does not handle empty list

for num in self.numbers[1:]:

result - num # Intentional defect: missing assignment

return result

def multiply(self):

product = 1

for num in self.numbers:

product \*= num

return product

def divide(self):

result = self.numbers[0] # Intentional defect: does not handle empty list

for num in self.numbers[1:]:

result /= num # Defect: should handle division by zero

return result

def mean(self):

return self.add() / len(self.numbers) # Intentional defect: division by zero if empty

# Test cases to simulate numbers

test\_cases = {

"Test Case 1": [10, 20, 30, 40],

"Test Case 2": [5, 15, 25],

"Test Case 3": [1, 0, -1], # Includes zero and negative number

"Test Case 4": [2], # Single element

"Test Case 5": [], # Empty list

"Test Case 6": [100, 200, 0], # Includes zero for division

"Test Case 7": [3, 6, 9], # All positive multiples

"Test Case 8": [-5, -10], # All negative numbers

"Test Case 9": [7.5, 2.5], # Floating point numbers

"Test Case 10": [0] # Single zero

}

# Run tests and collect results

results = {}

for case\_name, numbers in test\_cases.items():

try:

math\_ops = MathOperations(numbers)

results[case\_name] = {

'Addition': math\_ops.add(),

'Subtraction': math\_ops.subtract(),

'Multiplication': math\_ops.multiply(),

'Division': math\_ops.divide(),

'Mean': math\_ops.mean()

}

except Exception as e:

results[case\_name] = str(e)

# Display results

for case\_name, result in results.items():

print(f"{case\_name}: {result}")

### Explanation of the Code

1. **Class Definition**: The MathOperations class contains methods to perform various mathematical operations.
2. **Defects**:
   * The add() method has a defect where the accumulation of the total is incorrect because it does not return the total correctly.
   * The subtract() method has a defect where the subtraction does not update the result.
   * The divide() method does not handle division by zero correctly.
   * The mean() method could lead to division by zero if called on an empty list.
3. **Test Cases**: A dictionary of test cases simulates various lists of numbers, including edge cases (empty list, single element, etc.).
4. **Testing**: The program runs through each test case, invoking the class methods and capturing exceptions.

### Sample Output

When you run the program, you might see an output similar to this (actual results may vary due to the defects):

Test Case 1: {'Addition': 100, 'Subtraction': 10, 'Multiplication': 240000, 'Division': 0.00041666666666666664, 'Mean': 25.0}

Test Case 2: {'Addition': 45, 'Subtraction': 5, 'Multiplication': 1875, 'Division': 0.013333333333333332, 'Mean': 15.0}

Test Case 3: division by zero

Test Case 4: {'Addition': 2, 'Subtraction': 2, 'Multiplication': 2, 'Division': 2, 'Mean': 2.0}

Test Case 5: list index out of range

Test Case 6: float division by zero

Test Case 7: {'Addition': 18, 'Subtraction': 3, 'Multiplication': 162, 'Division': 0.05555555555555555, 'Mean': 6.0}

Test Case 8: {'Addition': -15, 'Subtraction': -5, 'Multiplication': 50, 'Division': 0.5, 'Mean': -7.5}

Test Case 9: {'Addition': 10.0, 'Subtraction': 7.5, 'Multiplication': 18.75, 'Division': 3.0, 'Mean': 5.0}

Test Case 10: {'Addition': 0, 'Subtraction': 0, 'Multiplication': 0, 'Division': 0, 'Mean': 0.0}

### Test Case Analysis

#### 1. **Test Case 5: Empty List**

* **Input**: []
* **Expected Behavior**: The program should handle the empty list gracefully, perhaps by returning None or an appropriate message for each operation.
* **Actual Behavior**: The program raises a ZeroDivisionError when calculating the mean because it attempts to divide by zero in the mean() method.
* **Fix**: Add a check for an empty list in each method before performing calculations:

python

Sao chép mã

if not self.numbers:

return None # or raise an Exception

#### 2. **Test Case 3: List with Zero and Negative Numbers**

* **Input**: [1, 0, -1]
* **Expected Behavior**: The operations should handle zero and negative numbers correctly.
* **Actual Behavior**: The addition and multiplication work correctly, but the subtraction could yield an unexpected value due to not updating the result variable.
* **Fix**: Ensure that the subtraction updates the result variable correctly:

python

Sao chép mã

result -= num # Correctly assign the result

#### 3. **Test Case 4: Single Element List**

* **Input**: [2]
* **Expected Behavior**: The operations should return the value itself for addition, subtraction, multiplication, and division.
* **Actual Behavior**: The addition and multiplication are correct, but the subtraction returns 2, which is technically correct but not a useful outcome for this method, and the division would return 2, which is also correct.
* **Fix**: While the results are not incorrect, you could add a note that the result of subtraction and division of a single number should be treated as identity for clarity.

#### 4. **Test Case 6: Includes Zero for Division**

* **Input**: [100, 200, 0]
* **Expected Behavior**: The division should not raise an error and should yield 0.0 as the final result.
* **Actual Behavior**: The division operation results in 0.0, which is correct; however, if there were an attempt to divide by zero (e.g., if self.numbers started with 0), it would lead to ZeroDivisionError.
* **Fix**: Add error handling for division:

python

Sao chép mã

if num == 0:

return "Error: Division by zero"

#### 5. **Test Case 2: Multiple Positive Numbers**

* **Input**: [5, 15, 25]
* **Expected Behavior**: The program should return correct results for all operations.
* **Actual Behavior**: The results returned by the addition, subtraction, multiplication, and division are correct, but the subtraction returns 5, which is not updated properly.
* **Fix**: As noted earlier, make sure to correctly update the subtraction result.

### Summary of Fixes

Here are some summarized fixes to implement for the above issues:

1. **Handle Empty Lists**: Check for an empty list before performing any calculations in each method.
2. **Correct Subtraction Logic**: Ensure that the subtraction method correctly updates the result.
3. **Consider Division Safety**: Implement checks in the division method to avoid division by zero errors.
4. **Clarify Single Element Behavior**: Make the behavior of the subtraction and division methods clearer when only one number is present.

### Revised Code Implementation

Here’s how you might revise the code to handle the identified issues:

class MathOperations:

def \_\_init\_\_(self, numbers):

self.numbers = numbers

def add(self):

if not self.numbers:

return None

return sum(self.numbers)

def subtract(self):

if not self.numbers:

return None

result = self.numbers[0]

for num in self.numbers[1:]:

result -= num

return result

def multiply(self):

if not self.numbers:

return None

product = 1

for num in self.numbers:

product \*= num

return product

def divide(self):

if not self.numbers:

return None

result = self.numbers[0]

for num in self.numbers[1:]:

if num == 0:

return "Error: Division by zero"

result /= num

return result

def mean(self):

if not self.numbers:

return None

return self.add() / len(self.numbers)

# Test cases to simulate numbers

test\_cases = {

"Test Case 1": [10, 20, 30, 40],

"Test Case 2": [5, 15, 25],

"Test Case 3": [1, 0, -1], # Includes zero and negative number

"Test Case 4": [2], # Single element

"Test Case 5": [], # Empty list

"Test Case 6": [100, 200, 0], # Includes zero for division

"Test Case 7": [3, 6, 9], # All positive multiples

"Test Case 8": [-5, -10], # All negative numbers

"Test Case 9": [7.5, 2.5], # Floating point numbers

"Test Case 10": [0] # Single zero

}

# Run tests and collect results

results = {}

for case\_name, numbers in test\_cases.items():

try:

math\_ops = MathOperations(numbers)

results[case\_name] = {

'Addition': math\_ops.add(),

'Subtraction': math\_ops.subtract(),

'Multiplication': math\_ops.multiply(),

'Division': math\_ops.divide(),

'Mean': math\_ops.mean()

}

except Exception as e:

results[case\_name] = str(e)

# Display results

for case\_name, result in results.items():

print(f"{case\_name}: {result}")

Test Case 1: {'Addition': 100, 'Subtraction': -80, 'Multiplication': 240000, 'Division': 0.00041666666666666664, 'Mean': 25.0}

Test Case 2: {'Addition': 45, 'Subtraction': -35, 'Multiplication': 1875, 'Division': 0.013333333333333332, 'Mean': 15.0}

Test Case 3: {'Addition': 0, 'Subtraction': 2, 'Multiplication': 0, 'Division': 'Error: Division by zero', 'Mean': 0.0}

Test Case 4: {'Addition': 2, 'Subtraction': 2, 'Multiplication': 2, 'Division': 2, 'Mean': 2.0}

Test Case 5: {'Addition': None, 'Subtraction': None, 'Multiplication': None, 'Division': None, 'Mean': None}

Test Case 6: {'Addition': 300, 'Subtraction': -100, 'Multiplication': 0, 'Division': 'Error: Division by zero', 'Mean': 100.0}

Test Case 7: {'Addition': 18, 'Subtraction': -12, 'Multiplication': 162, 'Division': 0.05555555555555555, 'Mean': 6.0}

Test Case 8: {'Addition': -15, 'Subtraction': 5, 'Multiplication': 50, 'Division': 0.5, 'Mean': -7.5}

Test Case 9: {'Addition': 10.0, 'Subtraction': 5.0, 'Multiplication': 18.75, 'Division': 3.0, 'Mean': 5.0}

Test Case 10: {'Addition': 0, 'Subtraction': 0, 'Multiplication': 0, 'Division': 0, 'Mean': 0.0}