21110324 _ Luong Dang Khoi _ SoftwareTesting _ Lab01

-- Part 1: --

```
from collections import Counter


class StatsProcessor:
    def __init__(self, numbers):
        self.numbers = numbers

    # Find the sum of all numbers in the list
    def add(self):
        total = 0
        for num in self.numbers:
            total += num
        return total

    # Find the average of the numbers
    def mean(self):
        if len(self.numbers) == 0:
            return 0
        return self.add() / len(self.numbers)

    # Find the median of the numbers
    def median(self):
        if len(self.numbers) == 0:
            return None
        sorted_nums = sorted(self.numbers)
        mid = len(sorted_nums) // 2
        if len(sorted_nums) % 2 == 0:
            return (sorted_nums[mid - 1] + sorted_nums[mid]) / 2
        else:
            return sorted_nums[mid]

    # Find the mode of the numbers
    def mode(self):
        if len(self.numbers) == 0:
            return None
        count = Counter(self.numbers)
        max_count = max(count.values())
        modes = [num for num, cnt in count.items() if cnt == max_count]
        if len(modes) == len(count):
            return None
        return modes[0]

    # Find the product of all numbers
    def multiply(self):
        if len(self.numbers) == 0:
            return None
        product = 1
        for num in self.numbers:
            product *= num
        return product
```

Code Decription:

The 'StatsProcessor' class processes a list of numbers and provides methods to compute basic statistics:

- **add():** Returns the sum of all numbers in the list.
- **mean():** Returns the average of the numbers.
- **median():** Returns the median of the numbers.
- **mode():** Returns the mode of the numbers.
- **multiply():** Returns the product of all numbers.

Testing Objectives:

- Verify the correct behavior of methods: add(), mean(), median(), mode(), and multiply() within the StatsProcessor class.
- Ensure the methods handle edge cases such as empty lists and lists with identical values.
- Confirm that mathematical operations (addition, mean, median, mode, multiplication) are calculated as expected.

Test Case Breakdown:

Test Case for Invalid Input:

```
# Test case to catch a ValueError when the input contains strings
try:
    sp = StatsProcessor(["a", "b", "c"])  # Invalid input
    print("test_invalid_input_string \t failed")
except ValueError as e:
    print(f"test_invalid_input_string passed: {e}")

# Test case to catch a ValueError when the input contains characters
try:
    sp = StatsProcessor([1, 2, "x", 4])  # Invalid input with a char
    print("test_invalid_input_characters \t failed")
except ValueError as e:
    print(f"test_invalid_input_characters passed: {e}")

# Test case to catch a TypeError when the input is not a list
try:
    sp = StatsProcessor("1234")  # Invalid input, not a list
    print("test_invalid_input_not_list \t failed")
except TypeError as e:
    print(f"test_invalid_input_not_list passed: {e}")
```

```
test_invalid_input_string        failed
test_invalid_input_characters    failed
test_invalid_input_not_list      failed
```

Test Case for Addition function:

```
# TC1: Test addition of positive numbers
sp = StatsProcessor([1, 2, 3])
assert sp.add() == 6, f"Expected 6, but got {sp.add()}"
print("test_add_positive \t passed")

# TC2: Test addition with negative numbers
sp = StatsProcessor([-1, -2, -3])
assert sp.add() == -6, f"Expected -6, but got {sp.add()}"
print("test_add_negative \t passed")

# TC3: Test addition with an empty list
sp = StatsProcessor([])
assert sp.median() is None, f"Expected None, but got {sp.median()}"
print("test_median_empty \t passed")
```

```
test_add_positive        passed
test_add_negative        passed
test_median_empty        passed
```

Test Case for Mean function:

```
# TC4: Test mean of a list of numbers
sp = StatsProcessor([2, 3, 3, 4])
assert sp.mode() == 3, f"Expected 3, but got {sp.mode()}"
print("test_mode_repeated \t passed")

# TC5:  Test mean with an empty list
sp = StatsProcessor([1, 2, 3])
assert sp.mode() is None, f"Expected None, but got {sp.mode()}"
print("test_mode_no_repeated \t passed")
```

```
test_mode_repeated       passed
test_mode_no_repeated    passed
```

Test Case for Median function:

```
# TC6:  Test median with odd number of elements
sp = StatsProcessor([3, 1, 2])
assert sp.median() == 2, f"Expected 2, but got {sp.median()}"
print("test_median_odd \t passed")

# TC7:  Test median with even number of elements
sp = StatsProcessor([1, 2, 3, 4])
assert sp.median() == 2.5, f"Expected 2.5, but got {sp.median()}"
print("test_median_even \t passed")

# TC8:  Test median with an empty list
sp = StatsProcessor([])
assert sp.median() is None, f"Expected None, but got {sp.median()}"
print("test_median_empty \t passed")
```

```
test_median_odd          passed
test_median_even         passed
test_median_empty        passed
```

Test Case for Mode function:

```
# TC9:  Test mode with repeated elements
sp = StatsProcessor([2, 3, 3, 4])
assert sp.mode() == 3, f"Expected 3, but got {sp.mode()}"
print("test_mode_repeated \t passed")

# TC10: Test mode with no repeated elements
sp = StatsProcessor([1, 2, 3])
assert sp.mode() is None, f"Expected None, but got {sp.mode()}"
print("test_mode_no_repeated \t passed")
```

```
test_mode_repeated       passed
test_mode_no_repeated    passed
```

Test Case for Multiply function:

```
# TC11: Test multiply positive numbers
sp = StatsProcessor([2, 3, 4])
assert sp.multiply() == 24, f"Expected 24, but got {sp.multiply()}"
print("test_multiply_positive \t passed")

# TC12: Test multiply with negative numbers
sp = StatsProcessor([-2, 3, 4])
assert sp.multiply() == -24, f"Expected -24, but got {sp.multiply()}"
print("test_multiply_negative \t passed")

# TC13: Test multiply with an empty list
sp = StatsProcessor([])
assert sp.multiply() is None, f"Expected None, but got {sp.multiply()}"
print("test_multiply_empty \t passed")
```

```
test_multiply_positive   passed
test_multiply_negative   passed
test_multiply_empty      passed
```

Test Case Summary:

- Total test cases: 16
- Covered operations: Addition, Mean, Median, Mode, Multiplication
- Edge cases: Invalid values, empty lists, identical values, negative numbers, and mixed number sets.

Test Metrics:

- Pass rate: 81.25% for basic functionality (Expected if all cases pass)
- Failures: **init()** does not validate input data. This will cause other methods in the class to fail.
- Code Coverage: Ensure all possible inputs (valid, invalid, and edge cases) are tested for each function.

Conclusion:

The test cases confirm that the StatsProcessor class works almost correctly for addition, average, median, mode, and multiplication operations.

Revised Code Implement:

Modify the Constructor for Input Validation

```
class StatsProcessor:
    def __init__(self, numbers):
        if not isinstance(numbers, list):
            raise TypeError("Input must be a list")
        if not all(isinstance(num, (int, float)) for num in numbers):
            raise ValueError("All elements must be integers or floats")
        self.numbers = numbers
```

Some defects in the methods and proposed solutions

```
# -- 1 -- Defect in mean() Method:
# Defect: Returns 0 for the mean of an empty list, which is misleading.
# Fix: Return None instead of 0 when the list is empty.
def mean(self):
    if len(self.numbers) == 0:
        return None  # Return None for empty list instead of 0
    return self.add() / len(self.numbers)

# -- 2 -- Defect in median() Method:
# Problem: If the list is empty, the method returns None, which is fine, but it could be explicitly documented as a design decision.
# However, the code does not handle lists with non-numeric values, which may raise an error during sorting.
# Additionally, it’s a good practice to handle non-list data types.
# Fix: Check if all elements are numeric types before proceeding with the sorting.
def median(self):
    if len(self.numbers) == 0:
        return None
    if not all(isinstance(num, (int, float)) for num in self.numbers):  # Check for non-numeric values
        raise ValueError("All elements must be numeric")
    sorted_nums = sorted(self.numbers)
    mid = len(sorted_nums) // 2
    if len(sorted_nums) % 2 == 0:
        return (sorted_nums[mid - 1] + sorted_nums[mid]) / 2
    else:
        return sorted_nums[mid]

# -- 3 -- Defect in multiply() Method:
# Defect: The multiply() method returns None for an empty list instead of 1.
# Fix: Change the multiply() method to return 1 when the list is empty.
def multiply(self):
    if len(self.numbers) == 0:
        return 1  # Return 1 for empty list (multiplicative identity)
    product = 1
    for num in self.numbers:
        product *= num
    return product

# -- 4 -- Defect in mode() Method:
# Defect: The mode() method returns None if all elements have the same frequency.
# Fix: Adjust the mode() method to return a list of modes when there are multiple values that share the highest frequency.
def mode(self):
    if len(self.numbers) == 0:
```

```
        return None
    count = Counter(self.numbers)
    max_count = max(count.values())
    modes = [num for num, cnt in count.items() if cnt == max_count]
    # If all numbers are equally frequent, return the modes list
    return modes if len(modes) > 1 else modes[0]  # Return a single mode or list of modes
```

-- Part 2: --

```
class StringManipulator:
    def __init__(self, text):
        self.text = text

    # Count vowels in string
    def count_vowels(self):
        vowels = 'aeiouAEIOU'
        count = 0
        for char in self.text:
            if char in vowels:
                count += 1
        return count

    # Reverse the string
    def reverse_string(self):
        return self.text[::-1]

    # Returns True if the string is symmetric
    def is_palindrome(self):
        if not self.text:
            return True
        reversed_text = self.reverse_string()
        return reversed_text.lower() == self.text.lower()

    # Find the most repeated letter in a string
    def most_frequent_char(self):
        if len(self.text) == 0:
            return None
        char_count = {}
        for char in self.text:
            if char.isalpha():
                if char in char_count:
                    char_count[char] += 1
                else:
                    char_count[char] = 1

        max_char = max(char_count, key=char_count.get)
        return max_char

    # Capitalize the first letter in a string
    def capitalize_words(self):
        if not self.text:
            return ""
        words = self.text.split()
        capitalized_words = [word.capitalize() for word in words]
        return ' '.join(capitalized_words)
```

Code Decription:

The StringManipulator class is designed to perform various string manipulation operations. It takes a string as input and provides several methods to interact with and analyze the string. The methods include:

- **count_vowels():** Counts the number of vowels (both uppercase and lowercase) in the given string.
- **reverse_string():** Returns the string in reverse order.
- **is_palindrome():** Checks whether the string is a palindrome (reads the same forwards and backwards), ignoring case and considering empty strings as palindromes.
- **most_frequent_char():** Identifies the most frequently occurring alphabetic character in the string, returning None for an empty string.
- **capitalize_words():** Capitalizes the first letter of each word in the string, returning an empty string for empty input.

Testing Objectives:

The testing objectives for the StringManipulator class can be defined as follows:

- **Correctness:** Ensure each method returns the expected results for a variety of input strings, including edge cases like empty strings and strings with mixed case.
- **Robustness:** Validate that methods handle invalid inputs gracefully, such as non-string types or strings containing special characters.
- **Performance:** Assess the efficiency of methods with large input strings to ensure they operate within acceptable time limits.
- **Edge Cases:** Test methods against specific edge cases (e.g., strings with no vowels, all vowels, palindromic phrases) to verify their correctness and stability.

Test Case Breakdown:

```
test_cases = {
    "Test Case 1": "",  # Empty string
    "Test Case 2": "hello world",  # Normal string with vowels and spaces
    "Test Case 3": "rhythm",  # String with no vowels
    "Test Case 4": "AEIOUaeiou",  # String with all vowels in mixed case
    "Test Case 5": "madam",  # Palindrome string
    "Test Case 6": "01210",  # Palindrome string but made of digits
    "Test Case 7": "=-\\.\\-=",  # Palindrome string but made of symbols
    "Test Case 8": "Racecar",  # Palindrome string, case insensitive
    "Test Case 9": "A man a plan a canal Panama",  # Palindrome with spaces and mixed case
    "Test Case 10": "hello,  world!",  # String with punctuation and extra spaces
    "Test Case 11": "hElLo wOrLd",  # Mixed case string
    "Test Case 12": "aaa",  # String where all characters are the same
    "Test Case 13": "abcde",  # All unique characters
    "Test Case 14": "123 32.4",  # Combination of digits and float
    "Test Case 15": ";..p.][].//''./@@\\T.T 123 asdfg",  # String with symbols and characters
    "Test Case 16": "@hello -world" # String starting from symbol
}



passed_cases = 0
failed_cases = 0
for key, value in test_cases.items():
    print(f'{key}: "{value}"')

    sm = StringManipulator(value)

    try:
        res = sm.count_vowels()
        print(f'Result = {res}')
        print("count_vowels \t\t Passed")
        passed_cases += 1
    except Exception as e:
        print(f"count_vowels \t\t Failed: {e}")
        failed_cases += 1

    try:
        res = sm.reverse_string()
        print(f'Result = {res}')
        print("reverse_string \t\t Passed")
        passed_cases += 1
    except Exception as e:
        print(f"reverse_string \t\t Failed: {e}")
        failed_cases += 1

    try:
        res = sm.is_palindrome()
        print(f'Result = {res}')
        print("is_palindrome \t\t Passed")
        passed_cases += 1
    except Exception as e:
        print(f"is_palindrome \t\t Failed: {e}")
        failed_cases += 1

    try:
        res = sm.most_frequent_char()
        print(f'Result = {res}')
        print("most_frequent_char \t Passed")
        passed_cases += 1
```

```
        passed_cases += 1
    except Exception as e:
        print(f"most_frequent_char \t Failed: {e}")
        failed_cases += 1

    try:
        res = sm.capitalize_words()
        print(f'Result = {res}')
        print("capitalize_words \t Passed")
        passed_cases += 1
    except Exception as e:
        print(f"capitalize_words \t Failed: {e}")
        failed_cases += 1

    print('\n')

print(f'Passed: {passed_cases}')
print(f'Failed: {failed_cases}')
```

```
most_frequent_char       Passed
Result = Aaa
capitalize_words         Passed


Test Case 13: "abcde"
Result = 2
count_vowels             Passed
Result = edcba
reverse_string           Passed
Result = False
is_palindrome            Passed
Result = a
most_frequent_char       Passed
Result = Abcde
capitalize_words         Passed


Test Case 14: "123 32.4"
Result = 0
count_vowels             Passed
Result = 4.23 321
reverse_string           Passed
Result = False
is_palindrome            Passed
most_frequent_char       Failed: max() arg is an empty sequence
Result = 123 32.4
capitalize_words         Passed


Test Case 15: ";..p.][].//''./@@\T.T 123 asdfg"
Result = 1
count_vowels             Passed
Result = gfdsa 321 T.T\@@/.''/.][].p..;
reverse_string           Passed
Result = False
is_palindrome            Passed
Result = T
most_frequent_char       Passed
Result = ;..p.][].//''./@@\t.t 123 Asdfg
capitalize_words         Passed


Test Case 16: "@hello -world"
Result = 3
count_vowels             Passed
Result = dlrow- olleh@
reverse_string           Passed
Result = False
is_palindrome            Passed
Result = l
most_frequent_char       Passed
Result = @hello -world
capitalize_words         Passed


Passed: 77
Failed: 3
```

Test Case Summary:

- **Total Test Cases**: 16
- **Total Methods Tested**: 5 methods
- **Total Executions**: 80 method executions in total.
- **Pass Percentage**: This represents the overall reliability of the methods.
- **Next Steps**: Failed test cases need further debugging for improvement.

Test Metrics:

Metric	Description	Value
Total Test Cases	Number of distinct test cases executed.	16
Methods Tested	Methods evaluated: `count_vowels()`, `reverse_string()`, `is_palindrome()`, `most_frequent_char()`, `capitalize_words()`.	5
Total Method Executions	Total executions across all test cases (15 test cases × 5 methods).	80
Passed Method Executions	Number of method executions that passed without errors.	X
Failed Method Executions	Number of executions that failed due to errors or edge cases.	Y
Pass Percentage	Percentage of method executions that passed successfully.	`96.25%`

Conclusion:

The StringManipulator class was thoroughly tested using 15 distinct test cases, evaluating the functionality of 5 core methods. A total of 75 method executions were performed, covering a wide range of inputs, including empty strings, palindrome strings, strings with mixed cases, and those containing symbols or numbers. The overall pass percentage reflects the class's robustness in handling common cases, but the failed executions highlight areas where edge cases, such as non-alphabetic characters or empty inputs, need further attention. Addressing these failures will improve the class's reliability, ensuring it works correctly for all valid and edge case inputs. Further refinements, especially around error handling and input validation, are recommended for improving the overall performance and accuracy of the class methods.

Defects and Revised Code Implementation :

```
# -- 1 -- most_frequent_char() does not handle cases where there are no alphabetic characters.
# Fix: Modify the method to return None when no alphabetic characters are present.
def most_frequent_char(self):
    if len(self.text) == 0:
        return None
    char_count = {}
    for char in self.text:
        if char.isalpha():  # Only consider alphabetic characters
            if char in char_count:
                char_count[char] += 1
            else:
                char_count[char] = 1
    if not char_count:  # No alphabetic characters found
        return None
    max_char = max(char_count, key=char_count.get)
    return max_char

# -- 2 -- is_palindrome() does not handle non-alphabetic characters properly.
# Fix: Strip non-alphabetic characters and compare only alphabetic ones.
def is_palindrome(self):
    if not self.text:
        return True
    # Remove non-alphabetic characters and ignore case
    cleaned_text = ''.join([char.lower() for char in self.text if char.isalpha()])
    reversed_text = cleaned_text[::-1]
    return cleaned_text == reversed_text

# -- 3 -- capitalize_words() does not handle cases where input text contains non-alphabetic characters or extra spaces.
# Fix: Modify the method to handle non-alphabetic characters and preserve their positions while capitalizing only the alphabetic words.
def capitalize_words(self):
    if not self.text:
        return ""
    words = self.text.split()
    capitalized_words = [word.capitalize() for word in words if word.isalpha()]
    return ' '.join(capitalized_words)

# -- 4 -- The class lacks input validation to ensure only strings are processed.
# Fix: Add input validation in the constructor to ensure that the text attribute is always a string.
def __init__(self, text):
```

```
    if not isinstance(text, str):
        raise ValueError("Input must be a string")
    self.text = text

# -- 5 -- count_vowels() does not handle non-alphabetic characters gracefully.
# Fix: Add a check to ignore non-alphabetic characters when counting vowels.
def count_vowels(self):
    vowels = 'aeiouAEIOU'
    count = 0
    for char in self.text:
        if char.isalpha() and char in vowels:
            count += 1
    return count
```