Assignment-8.2

Name: Gampa Rithika
Roll No: 2303A51190
Batch No: 03

Task 1 – Test-Driven Development for Even/Odd Number Validator

• Use AI tools to first generate test cases for a function is_even(n) and then implement the function so that it satisfies all generated tests.

Requirements:

• Input must be an integer
• Handle zero, negative numbers, and large integers

Example Test Scenarios:

is_even(2) → True
is_even(7) → False
is_even(0) → True
is_even(-4) → True
is_even(9) → False

Expected Output -1

• A correctly implemented is_even() function that passes all AI-generated test cases

General Test Cases Prompt:

Generate general test cases for a Python function is_even(n) and provide a simple Python program that prints outputs for each test case.

```python
#Generate general test cases for a Python function is_even(n) and provide a simple Python program that prints outputs for each t
def is_even(n):
    return n % 2 == 0
#General test cases for is_even function
test_cases = [0, 1, 2, -1, -2, 3.5, 4.0]
for test in test_cases:
    result = is_even(test)
    print(f"Is {test} an even number? {result}")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
Is 0 an even number? True
Is 1 an even number? False
Is 2 an even number? True
Is -1 an even number? False
Is -2 an even number? True
Is 3.5 an even number? False
Is 4.0 an even number? True
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Assertion Test Cases Prompt:

Write assertion-based test cases for a Python function is_even(n).

```python
#Write assertion-based test cases for a Python function is_even(n).
def is_even(n):
    return n % 2 == 0
#Assertion-based test cases for is_even function
assert is_even(0) == True
assert is_even(1) == False
assert is_even(2) == True
assert is_even(-1) == False
assert is_even(-2) == True
assert is_even(3.5) == False
assert is_even(4.0) == True
print("All assertion test cases for is_even function passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All assertion test cases for is_even function passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Unit Test Cases Prompt:

Generate Python unittest test cases for the function is_even(n).

```python
#Generate Python unittest test cases for the function is_even(n).
import unittest
def is_even(n):
    return n % 2 == 0
class TestIsEven(unittest.TestCase):
    def test_is_even(self):
        self.assertTrue(is_even(0))
        self.assertFalse(is_even(1))
        self.assertTrue(is_even(2))
        self.assertFalse(is_even(-1))
        self.assertTrue(is_even(-2))
        self.assertFalse(is_even(3.5))
        self.assertTrue(is_even(4.0))
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
Ran 1 test in 0.000s

OK
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Pytest Cases Prompt:

Generate pytest test cases for a Python function is_even(n).

```python
#Generate pytest test cases for a Python function is_even(n).
def is_even(n):
    return n % 2 == 0
#Pytest test cases for is_even function
def test_is_even():
    assert is_even(0) == True
    assert is_even(1) == False
    assert is_even(2) == True
    assert is_even(-1) == False
    assert is_even(-2) == True
    assert is_even(3.5) == False
    assert is_even(4.0) == True
if __name__ == "__main__":
    test_is_even()
    print("All pytest test cases for is_even function passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All pytest test cases for is_even function passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Code Explanation

The function is_even(n) checks whether a given number is even while ensuring valid input. First, it verifies that the input is an integer to prevent errors. Then it uses the modulus operator (n % 2) to determine divisibility by 2. If the remainder is 0, the function returns True (even); otherwise, it returns False (odd).

The function correctly handles edge cases such as zero, negative numbers, and large integers. Test-driven development is used to validate correctness through assertions and automated tests, ensuring reliable behavior.

Task 2 – Test-Driven Development for String Case Converter
• Ask AI to generate test cases for two functions:
• to_uppercase(text)
• to_lowercase(text)
Requirements:
• Handle empty strings
• Handle mixed-case input
• Handle invalid inputs such as numbers or None
Example Test Scenarios:
to_uppercase("ai coding") → "AI CODING"
to_lowercase("TEST") → "test"
to_uppercase("") → ""
to_lowercase(None) → Error or safe handling
Expected Output -2

• Two string conversion functions that pass all AI-generated test cases with safe input handling.

General Test Cases Prompt:

Generate test cases for two Python functions to_uppercase(text) and to_lowercase(text) that convert string case and handle empty and invalid inputs.

```python
#Generate test cases for two Python functions to_uppercase(text) and to_lowercase(text) that convert string case and handle empty
def to_uppercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.upper()
def to_lowercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.lower()
#Test cases for to_uppercase and to_lowercase functions
test_cases = ["Hello World", "", 123, None, "PYTHON"]
for test in test_cases:
    uppercase_result = to_uppercase(test)
    lowercase_result = to_lowercase(test)
    print(f"Original: {test} | Uppercase: {uppercase_result} | Lowercase: {lowercase_result}")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
Original: Hello World | Uppercase: HELLO WORLD | Lowercase: hello world
Original:  | Uppercase:  | Lowercase:
Original: 123 | Uppercase: Error: Input must be a string. | Lowercase: Error: Input must be a string.
Original: None | Uppercase: Error: Input must be a string. | Lowercase: Error: Input must be a string.
Original: PYTHON | Uppercase: PYTHON | Lowercase: python
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Assertion Test Cases Prompt:

Write Python assertion statements to test to_uppercase(text) and to_lowercase(text) functions.

```python
#Write Python assertion statements to test to_uppercase(text) and to_lowercase(text) functions.
def to_uppercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.upper()
def to_lowercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.lower()
#Assertion statements for to_uppercase and to_lowercase functions
assert to_uppercase("Hello World") == "HELLO WORLD"
assert to_uppercase("") == ""
assert to_uppercase(123) == "Error: Input must be a string."
assert to_uppercase(None) == "Error: Input must be a string."
assert to_uppercase("PYTHON") == "PYTHON"
assert to_lowercase("Hello World") == "hello world"
assert to_lowercase("") == ""
assert to_lowercase(123) == "Error: Input must be a string."
assert to_lowercase(None) == "Error: Input must be a string."
assert to_lowercase("PYTHON") == "python"
print("All assertion test cases for to_uppercase and to_lowercase functions passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All assertion test cases for to_uppercase and to_lowercase functions passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Unit Test Cases Prompt:

Write Python unittest test cases for to_uppercase(text) and to_lowercase(text) functions including normal, empty, and invalid input cases.

```python
#Write Python unittest test cases for to_uppercase(text) and to_lowercase(text) functions including normal, empty, and invalid in
import unittest
def to_uppercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.upper()
def to_lowercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.lower()
class TestStringCaseConversion(unittest.TestCase):
    def test_to_uppercase(self):
        self.assertEqual(to_uppercase("Hello World"), "HELLO WORLD")
        self.assertEqual(to_uppercase(""), "")
        self.assertEqual(to_uppercase(123), "Error: Input must be a string.")
        self.assertEqual(to_uppercase(None), "Error: Input must be a string.")
        self.assertEqual(to_uppercase("PYTHON"), "PYTHON")
    def test_to_lowercase(self):
        self.assertEqual(to_lowercase("Hello World"), "hello world")
        self.assertEqual(to_lowercase(""), "")
        self.assertEqual(to_lowercase(123), "Error: Input must be a string.")
        self.assertEqual(to_lowercase(None), "Error: Input must be a string.")
        self.assertEqual(to_lowercase("PYTHON"), "python")
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
Ran 2 tests in 0.000s

OK
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Pytest Cases Prompt:

Generate pytest test cases using assert statements for to_uppercase(text) and to_lowercase(text) functions.

```python
#Generate pytest test cases using assert statements for to_uppercase(text) and to_lowercase(text) functions.
def to_uppercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.upper()
def to_lowercase(text):
    if not isinstance(text, str):
        return "Error: Input must be a string."
    return text.lower()
#Pytest test cases for to_uppercase and to_lowercase functions
def test_to_uppercase():
    assert to_uppercase("Hello World") == "HELLO WORLD"
    assert to_uppercase("") == ""
    assert to_uppercase(123) == "Error: Input must be a string."
    assert to_uppercase(None) == "Error: Input must be a string."
    assert to_uppercase("PYTHON") == "PYTHON"
def test_to_lowercase():
    assert to_lowercase("Hello World") == "hello world"
    assert to_lowercase("") == ""
    assert to_lowercase(123) == "Error: Input must be a string."
    assert to_lowercase(None) == "Error: Input must be a string."
    assert to_lowercase("PYTHON") == "python"
if __name__ == "__main__":
    test_to_uppercase()
    test_to_lowercase()
    print("All pytest test cases for to_uppercase and to_lowercase functions passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All pytest test cases for to_uppercase and to_lowercase functions passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Code explanation:

The functions to_uppercase(text) and to_lowercase(text) convert a string into uppercase and lowercase formats. Each function first checks whether the input is a valid string to avoid runtime errors. After validation, built-in string methods are used to perform the conversion efficiently.

The functions handle edge cases such as empty strings and invalid inputs safely. Test-driven development ensures the functions behave correctly under different scenarios, improving reliability and correctness.

Task 3 – Test-Driven Development for List Sum Calculator
• Use AI to generate test cases for a function sum_list(numbers)
that calculates the sum of list elements.
Requirements:
• Handle empty lists
• Handle negative numbers
• Ignore or safely handle non-numeric values
Example Test Scenarios:
sum_list([1, 2, 3]) → 6
sum_list([]) → 0
sum_list([-1, 5, -4]) → 0
sum_list([2, "a", 3]) → 5
Expected Output 3
• A robust list-sum function validated using AI-generated test
Cases

General Test Cases Prompt:

Generate test cases for a Python function sum_list(numbers) that returns the sum of list elements, handles empty lists, negative numbers, and ignores non-numeric values.

```python
#Generate test cases for a Python function sum_list(numbers) that returns the sum of list elements, handles empty lists, negative
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
    return total
#Test cases for sum_list function
test_cases = [
    [1, 2, 3],  # Expected output: 6
    [],  # Expected output: 0
    [-1, -2, -3],  # Expected output: -6
    [1, 'a', 2.5, None, 3],  # Expected output: 6.5
    ['x', 'y', 'z'],  # Expected output: 0
]
for test in test_cases:
    result = sum_list(test)
    print(f"Sum of {test} is: {result}")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
Sum of [1, 2, 3] is: 6
Sum of [] is: 0
Sum of [-1, -2, -3] is: -6
Sum of [1, 'a', 2.5, None, 3] is: 6.5
Sum of ['x', 'y', 'z'] is: 0
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Assertion Test Cases Prompt:

Write Python assertion statements to test the function sum_list(numbers).

```python
#Write Python assertion statements to test the function sum_list(numbers).
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
    return total
#Assertion statements for sum_list function
assert sum_list([1, 2, 3]) == 6
assert sum_list([]) == 0
assert sum_list([-1, -2, -3]) == -6
assert sum_list([1, 'a', 2.5, None, 3]) == 6.5
assert sum_list(['x', 'y', 'z']) == 0
print("All assertion test cases for sum_list function passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All assertion test cases for sum_list function passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Unit Test Cases Prompt:

Write Python unittest test cases for the function sum_list(numbers) including normal, empty, negative, and invalid input cases.

```python
#Write Python unittest test cases for the function sum_list(numbers) including normal, empty, negative, and invalid input cases.
import unittest
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
    return total
class TestSumList(unittest.TestCase):
    def test_sum_list(self):
        self.assertEqual(sum_list([1, 2, 3]), 6)
        self.assertEqual(sum_list([]), 0)
        self.assertEqual(sum_list([-1, -2, -3]), -6)
        self.assertEqual(sum_list([1, 'a', 2.5, None, 3]), 6.5)
        self.assertEqual(sum_list(['x', 'y', 'z']), 0)
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
Ran 1 test in 0.000s

OK
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Pytest Cases Prompt:

Generate pytest test cases using assert statements for the function sum_list(numbers).

```python
#Generate pytest test cases using assert statements for the function sum_list(numbers).
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
    return total
#Pytest test cases for sum_list function
def test_sum_list():
    assert sum_list([1, 2, 3]) == 6
    assert sum_list([]) == 0
    assert sum_list([-1, -2, -3]) == -6
    assert sum_list([1, 'a', 2.5, None, 3]) == 6.5
    assert sum_list(['x', 'y', 'z']) == 0
if __name__ == "__main__":
    test_sum_list()
    print("All pytest test cases for sum_list function passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All pytest test cases for sum_list function passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Code Explanation:

The function sum_list(numbers) calculates the total of numeric values in a list while safely handling different input cases. It iterates through the list and adds only valid numeric elements, ignoring non-numeric values to prevent errors.

The function also handles edge cases such as empty lists and negative numbers, ensuring stable behavior. Test-driven development is used to verify correctness and robustness through multiple test scenarios.

Task 4 – Test Cases for Student Result Class

• Generate test cases for a StudentResult class with the following methods:

• add_marks(mark)

• calculate_average()

• get_result()

Requirements:

• Marks must be between 0 and 100

• Average ≥ 40 → Pass, otherwise Fail

Example Test Scenarios:

Marks: [60, 70, 80] → Average: 70 → Result: Pass

Marks: [30, 35, 40] → Average: 35 → Result: Fail

Marks: [-10] → Error

Expected Output -4

• A fully functional StudentResult class that passes all AI-generated test

General Test Cases Prompt:

Generate test cases for a Python class StudentResult with methods add_marks, calculate_average, and get_result. Marks must be between 0 and 100 and result depends on average.

```python
#Generate test cases for a Python class StudentResult with methods add_marks, calculate_average, and get_result. Marks must be between 0 and 100 and result de
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            return "Error: Marks must be between 0 and 100."
    def calculate_average(self):
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"
#Test cases for StudentResult class
student = StudentResult()
print(student.add_marks(85))  # Expected output: None
print(student.add_marks(92))  # Expected output: None
print(student.add_marks(78))  # Expected output: None
print(student.add_marks(105))  # Expected output: "Error: Marks must be between 0 and 100."
print(student.calculate_average())  # Expected output: 85.0
print(student.get_result())  # Expected output: "B"
student.add_marks(55)
print(student.calculate_average())  # Expected output: 77.5
print(student.get_result())  # Expected output: "C"
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
None
None
None
Error: Marks must be between 0 and 100.
85.0
B
77.5
C
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Assertion Test Cases Prompt:

Write Python assertion statements to test the StudentResult class.

```python
#Write Python assertion statements to test the StudentResult class.
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            return "Error: Marks must be between 0 and 100."
    def calculate_average(self):
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"
#Assertion statements for StudentResult class
student = StudentResult()
assert student.add_marks(85) is None
assert student.add_marks(92) is None
assert student.add_marks(78) is None
assert student.add_marks(105) == "Error: Marks must be between 0 and 100."
assert student.calculate_average() == 85.0
assert student.get_result() == "B"
student.add_marks(55)
assert student.calculate_average() == 77.5
assert student.get_result() == "C"
print("All assertion test cases for StudentResult class passed successfully!")
```


OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All assertion test cases for StudentResult class passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Unit Test Cases Prompt:

Write Python unittest test cases for the StudentResult class including valid marks, invalid marks, pass, and fail cases.

```python
#Write Python unittest test cases for the StudentResult class including valid marks, invalid marks, pass, and fail cases.
import unittest
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            return "Error: Marks must be between 0 and 100."
    def calculate_average(self):
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"
class TestStudentResult(unittest.TestCase):
    def setUp(self):
        self.student = StudentResult()
    def test_add_marks_valid(self):
        self.assertIsNone(self.student.add_marks(85))
        self.assertIsNone(self.student.add_marks(92))
        self.assertIsNone(self.student.add_marks(78))
    def test_add_marks_invalid(self):
        self.assertEqual(self.student.add_marks(105), "Error: Marks must be between 0 and 100.")
        self.assertEqual(self.student.add_marks(-5), "Error: Marks must be between 0 and 100.")
    def test_calculate_average_and_get_result(self):
        self.student.add_marks(85)
        self.student.add_marks(92)
        self.student.add_marks(78)
        self.assertEqual(self.student.calculate_average(), 85.0)
        self.assertEqual(self.student.get_result(), "B")
        self.student.add_marks(55)
        self.assertEqual(self.student.calculate_average(), 77.5)
        self.assertEqual(self.student.get_result(), "C")
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
Ran 3 tests in 0.000s

OK
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Pytest Cases Prompt:

Generate pytest test cases using assert statements for the StudentResult class methods.

```python
#Generate pytest test cases using assert statements for the StudentResult class methods.
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            return "Error: Marks must be between 0 and 100."
    def calculate_average(self):
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        if average >= 90:
            return "A"
        elif average >= 80:
            return "B"
        elif average >= 70:
            return "C"
        elif average >= 60:
            return "D"
        else:
            return "F"
#Pytest test cases for StudentResult class
def test_student_result():
    student = StudentResult()
    assert student.add_marks(85) is None
    assert student.add_marks(92) is None
    assert student.add_marks(78) is None
    assert student.add_marks(105) == "Error: Marks must be between 0 and 100."
    assert student.calculate_average() == 85.0
    assert student.get_result() == "B"
    student.add_marks(55)
    assert student.calculate_average() == 77.5
    assert student.get_result() == "C"
if __name__ == "__main__":
    test_student_result()
    print("All pytest test cases for StudentResult class passed successfully!")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
All pytest test cases for StudentResult class passed successfully!
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Code Explanation:

The StudentResult class manages student marks and determines performance. The add_marks() method validates that marks are within the allowed range (0–100) before storing them. The calculate_average() method computes the average score, and get_result() decides whether the student passes or fails based on that average.

The class includes validation and edge-case handling to ensure reliable behavior. Test-driven development is used to confirm that all methods work correctly under different scenarios.

Task 5 – Test-Driven Development for Username Validator
Requirements:
• Minimum length: 5 characters
• No spaces allowed
• Only alphanumeric characters
Example Test Scenarios:
is_valid_username("user01") → True
is_valid_username("ai") → False
is_valid_username("user name") → False
is_valid_username("user@123") → False
Expected Output 5
A username validation function that passes all AI-generated test cases.

General Test Cases Prompt:

Generate test cases for a Python function is_valid_username(username) that checks minimum length, no spaces, and only alphanumeric characters.

```python
#Generate test cases for a Python function is_valid_username(username) that checks minimum length, no spaces, and only alphanum
def is_valid_username(username):
    if not isinstance(username, str):
        return "Error: Username must be a string."
    if len(username) < 5:
        return "Error: Username must be at least 5 characters long."
    if ' ' in username:
        return "Error: Username cannot contain spaces."
    if not username.isalnum():
        return "Error: Username can only contain alphanumeric characters."
    return True
#Test cases for is_valid_username function
test_cases = [
    "user1",  # Expected output: True
    "us",  # Expected output: "Error: Username must be at least 5 characters long."
    "user name",  # Expected output: "Error: Username cannot contain spaces."
    "user@name",  # Expected output: "Error: Username can only contain alphanumeric characters."
    12345,  # Expected output: "Error: Username must be a string."
    "validUser123"  # Expected output: True
]
for test in test_cases:
    result = is_valid_username(test)
    print(f"Username: {test} | Valid: {result}")
```

OUTPUT:

```
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting> & C:/Users/Apple/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Apple/OneDrive/Desktop/Ai Assisting/lab8.py"
Username: user1 | Valid: True
Username: us | Valid: Error: Username must be at least 5 characters long.
Username: user name | Valid: Error: Username cannot contain spaces.
Username: user@name | Valid: Error: Username can only contain alphanumeric characters.
Username: 12345 | Valid: Error: Username must be a string.
Username: validUser123 | Valid: True
PS C:\Users\Apple\OneDrive\Desktop\Ai Assisting>
```

Assertion Test Cases Prompt:

Write Python assertion statements to test the function is_valid_username(username).

```python
#Write Python assertion statements to test the function is_valid_username(username).
def is_valid_username(username):
    if not isinstance(username, str):
        return "Error: Username must be a string."
    if len(username) < 5:
        return "Error: Username must be at least 5 characters long."
    if ' ' in username:
        return "Error: Username cannot contain spaces."
    if not username.isalnum():
        return "Error: Username can only contain alphanumeric characters."
    return True
#Assertion statements for is_valid_username function
assert is_valid_username("user1") == True
assert is_valid_username("us") == "Error: Username must be at least 5 characters long."
assert is_valid_username("user name") == "Error: Username cannot contain spaces."
assert is_valid_username("user@name") == "Error: Username can only contain alphanumeric characters."
assert is_valid_username(12345) == "Error: Username must be a string."
assert is_valid_username("validUser123") == True
print("All assertion test cases for is_valid_username function passed successfully!")
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python
PS C:\Users\ravul\OneDrive\Desktop\3-2\AI Assistant Coding> & C:/Python314/python.exe "c:/Users/ravul/OneDrive/Desktop/3-2/AI Assistant Coding/17-02-26.py"
PS C:\Users\ravul\OneDrive\Desktop\3-2\AI Assistant Coding> & C:/Python314/python.exe "c:/Users/ravul/OneDrive/Desktop/3-2/AI Assistant Coding/17-02-26.py"
All assertion test cases for is_valid_username function passed successfully!
PS C:\Users\ravul\OneDrive\Desktop\3-2\AI Assistant Coding>
```

Unit Test Cases Prompt:

Write Python unittest test cases for the function is_valid_username(username) including valid, short, space, and special character cases.

```python
#Write Python unittest test cases for the function is_valid_username(username) including valid, short, space, and spec
import unittest
def is_valid_username(username):
    if not isinstance(username, str):
        return "Error: Username must be a string."
    if len(username) < 5:
        return "Error: Username must be at least 5 characters long."
    if ' ' in username:
        return "Error: Username cannot contain spaces."
    if not username.isalnum():
        return "Error: Username can only contain alphanumeric characters."
    return True
class TestIsValidUsername(unittest.TestCase):
    def test_valid_username(self):
        self.assertTrue(is_valid_username("user1"))
        self.assertTrue(is_valid_username("validUser123"))
    def test_short_username(self):
        self.assertEqual(is_valid_username("us"), "Error: Username must be at least 5 characters long.")
    def test_username_with_space(self):
        self.assertEqual(is_valid_username("user name"), "Error: Username cannot contain spaces.")
    def test_username_with_special_characters(self):
        self.assertEqual(is_valid_username("user@name"), "Error: Username can only contain alphanumeric characters.")
    def test_non_string_username(self):
        self.assertEqual(is_valid_username(12345), "Error: Username must be a string.")
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

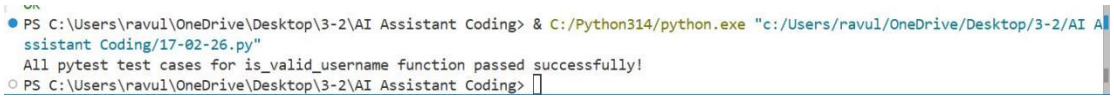
```
PS C:\Users\ravul\OneDrive\Desktop\3-2\AI Assistant Coding> & C:/Python314/python.exe "c:/Users/ravul/OneDrive/Desktop/3-2/AI A
ssistant Coding/17-02-26.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS C:\Users\ravul\OneDrive\Desktop\3-2\AI Assistant Coding>
```

Pytest Cases Prompt:

Generate pytest test cases using assert statements for the function is_valid_username(username).

```python
#Generate pytest test cases using assert statements for the function is_valid_username(username).
def is_valid_username(username):
    if not isinstance(username, str):
        return "Error: Username must be a string."
    if len(username) < 5:
        return "Error: Username must be at least 5 characters long."
    if ' ' in username:
        return "Error: Username cannot contain spaces."
    if not username.isalnum():
        return "Error: Username can only contain alphanumeric characters."
    return True
#Pytest test cases for is_valid_username function
def test_is_valid_username():
    assert is_valid_username("user1") == True
    assert is_valid_username("us") == "Error: Username must be at least 5 characters long."
    assert is_valid_username("user name") == "Error: Username cannot contain spaces."
    assert is_valid_username("user@name") == "Error: Username can only contain alphanumeric characters."
    assert is_valid_username(12345) == "Error: Username must be a string."
    assert is_valid_username("validUser123") == True
if __name__ == "__main__":
    test_is_valid_username()
    print("All pytest test cases for is_valid_username function passed successfully!")
```

OUTPUT:



Code Explanation:

The function is_valid_username(username) checks whether a username follows the required validation rules. It verifies that the input is a string, has a minimum length of five characters, contains no spaces, and includes only alphanumeric characters.

If any rule is violated, the function returns false, ensuring invalid usernames are rejected. Test-driven development is used to confirm the validator works correctly across valid, invalid, and edge-case inputs.