ASSIGNMENT – 8.2

B.Varun

2303A52020

Batch – 38

Task Description

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 case

PROMPT:

Create test cases and implement a Python function is_even(n) that checks whether a number is even. Handle zero, negative numbers, and large integers.

CODE:

```python
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
# Test cases
print(is_even(0))
print(is_even(-2))
print(is_even(-3))
print(is_even(2))
print(is_even(3))
print(is_even(1000000000000))
print(is_even(1000000000001))
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Dokumen\SERVER> & "C:/Prog
s/varun/OneDrive/Dokumen/SERVER/week1.py
True
True
False
True
False
True
False
PS C:\Users\varun\OneDrive\Dokumen\SERVER>
```

EXPLANATION:

The function is_even(n) checks whether a number is even using the modulus operator. The print statements are general test cases used to verify the function with positive, negative, zero, and large integers.

PROMPT:

Write Python assert test cases for is_even(n) and implement the function. Include tests for zero, negative numbers, and large integers.

CODE:

```python
def is_even(n):
    return n % 2 == 0
# Test cases
assert is_even(0) == True
assert is_even(-2) == True
assert is_even(-3) == False
assert is_even(2) == True
assert is_even(3) == False
assert is_even(1000000) == True
assert is_even(1000001) == False
print("All test cases passed!")
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Dokumen\SERVER> & "(
s/varun/OneDrive/Dokumen/SERVER/week1.py
All test cases passed!
PS C:\Users\varun\OneDrive\Dokumen\SERVER>
```

EXPLANATION:

The function is_even(n) checks whether a number is even using the modulus operator. The assert

statements are used as test cases to automatically verify the function with positive numbers, negative numbers, zero, and large integers. If all assertions are correct, the program prints "All test cases passed!".

PROMPT:

Write pytest test cases for is_even(n) and implement the function. Handle zero, negative numbers, large integers, and invalid input.

CODE:

```
 1   # Write pytest test cases for is_even(n) and impl
 2 v def is_even(n):
 3 v     if not isinstance(n, int):
 4           raise ValueError("Input must be an intege
 5       return n % 2 == 0
 6   import pytest
 7 v def test_is_even():
 8       # Test with zero
 9       assert is_even(0) == True
10
11       # Test with positive even number
12       assert is_even(4) == True
13
14       # Test with positive odd number
15       assert is_even(5) == False
16
17       # Test with negative even number
18       assert is_even(-2) == True
19
20       # Test with negative odd number
21       assert is_even(-3) == False
22
23       # Test with large even integer
24       assert is_even(1000000000) == True
25
26       # Test with large odd integer
27       assert is_even(1000000001) == False
28
29       # Test with invalid input (string)
30 v     with pytest.raises(ValueError):
31           is_even("not a number")
32
```

OUTPUT:

```
==================================== test session starts ====================
platform win32 -- Python 3.12.0, pytest-8.4.2, pluggy-1.6.0
rootdir: C:\Users\varun\OneDrive\Desktop\python
collected 1 item

test_even.py .

============================================== 1 passed in 0.02s ============
```

EXPLANATION:

The is_even(n) function checks whether a number is even using the modulus operator. The pytest test function verifies the program with zero, positive, negative, large numbers, and invalid input.If all tests pass, it confirms the function works correctly.
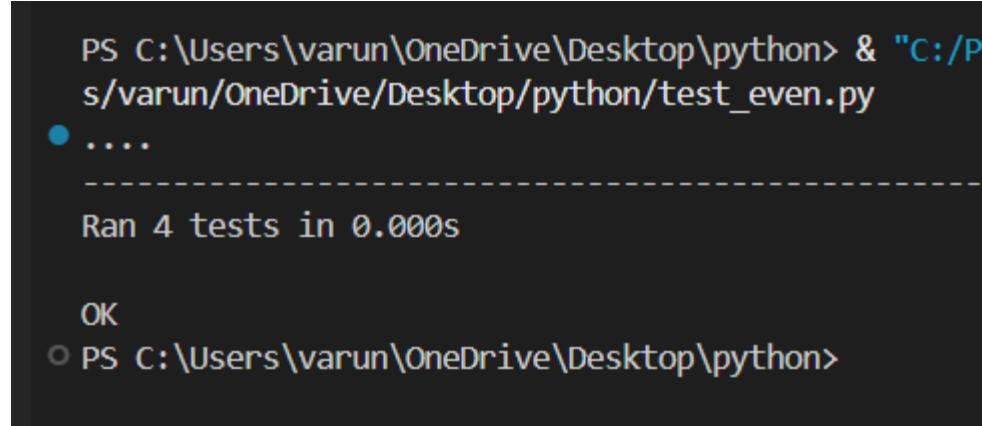
PROMPT:

Write unit tests for is_even(n) and implement the function. Handle zero, negative numbers, large integers, and invalid input.

CODE:

```
1    # Write unit tests for is_even(n) and implement the function.
2    # Handle zero, negative numbers, large integers, and invalid input.
3    import unittest
4    def is_even(n):
5        if not isinstance(n, int):
6            raise ValueError("Input must be an integer.")
7        return n % 2 == 0
8    class TestIsEven(unittest.TestCase):
9        def test_even_numbers(self):
10           self.assertTrue(is_even(2))
11           self.assertTrue(is_even(0))
12           self.assertTrue(is_even(-4))
13
14       def test_odd_numbers(self):
15           self.assertFalse(is_even(1))
16           self.assertFalse(is_even(-3))
17
18       def test_large_integers(self):
19           self.assertTrue(is_even(10**10))
20           self.assertFalse(is_even(10**10 + 1))
21
22       def test_invalid_input(self):
23           with self.assertRaises(ValueError):
24               is_even("string")
25           with self.assertRaises(ValueError):
26               is_even(3.14)
27   if __name__ == '__main__':
28       unittest.main()
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\python> & "C:/P
s/varun/OneDrive/Desktop/python/test_even.py
....
----------------------------------------------------
Ran 4 tests in 0.000s

OK
PS C:\Users\varun\OneDrive\Desktop\python>
```

EXPLANATION:

The is_even() function checks whether a number is even. The unittest test class verifies the function using even, odd, zero, large numbers, and invalid input. All tests confirm the function works correctly.

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.

PROMPT:

Write Python functions to_uppercase(text) and to_lowercase(text) with general print test cases. Handle empty strings, mixed-case text, and invalid inputs like None safely.

CODE:

```
 2  def to_uppercase(text):
 3      if text is None:
 4          return "Invalid input: None"
 5      if not isinstance(text, str):
 6          return "Invalid input: Not a string"
 7      return text.upper()
 8  def to_lowercase(text):
 9      if text is None:
10          return "Invalid input: None"
11      if not isinstance(text, str):
12          return "Invalid input: Not a string"
13      return text.lower()
14  # Test cases
15  print(to_uppercase("Hello World!"))
16  print(to_lowercase("Hello World!"))
17  print(to_uppercase(""))
18  print(to_lowercase(""))
19  print(to_uppercase(None))
20  print(to_lowercase(None))
21  print(to_uppercase(123))
22  print(to_lowercase(123))
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\python> & "C:/Program Files/Py
ython/test_even.py
HELLO WORLD!
hello world!


Invalid input: None
Invalid input: None
Invalid input: Not a string
Invalid input: Not a string
PS C:\Users\varun\OneDrive\Desktop\python>
```

PROMPT:

EXPALANTION:

The program implements two string conversion functions with safe input handling. Print statements are used as general test cases to check normal strings, empty strings, None values, and non-string inputs.

PROMPT:

Create unittest test cases for to_uppercase(text) and to_lowercase(text) in Python and implement the functions to pass all tests, including handling None and non-string inputs safely.

CODE:

```python
import unittest
def to_uppercase(text):
    if text is None:
        return None
    if not isinstance(text, str):
        raise ValueError("Input must be a string")
    return text.upper()
def to_lowercase(text):
    if text is None:
        return None
    if not isinstance(text, str):
        raise ValueError("Input must be a string")
    return text.lower()
class TestStringFunctions(unittest.TestCase):
    def test_to_uppercase(self):
        self.assertEqual(to_uppercase("hello"), "HELLO")
        self.assertEqual(to_uppercase("HeLLo"), "HELLO")
        self.assertEqual(to_uppercase(""), "")
        self.assertIsNone(to_uppercase(None))
        with self.assertRaises(ValueError):
            to_uppercase(123)
    def test_to_lowercase(self):
        self.assertEqual(to_lowercase("HELLO"), "hello")
        self.assertEqual(to_lowercase("HeLLo"), "hello")
        self.assertEqual(to_lowercase(""), "")
        self.assertIsNone(to_lowercase(None))
        with self.assertRaises(ValueError):
            to_lowercase(123)
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
PS C:\Users\varun\OneDrive\Desktop\python> &
ython/test_even.py
..
----------------------------------------
Ran 2 tests in 0.000s

OK
PS C:\Users\varun\OneDrive\Desktop\python>
```

EXPLANATION:

PROMPT:

The program defines two functions to convert strings into uppercase and lowercase. It uses the unittest module to test the functions with normal strings, empty strings, None values, and invalid inputs. All test cases are executed using unittest.main() and the output "OK" shows that all tests passed successfully.

PROMPT:

Write pytest test cases for two Python functions to_uppercase(text) and to_lowercase(text). Test normal strings, mixed-case strings, empty strings, None, and numbers. Then implement both functions with safe input handling so all tests pass.

CODE:

```
1   # Write pytest test cases for two Python functions to_uppercase(text) and to_lowercase(text).
2   # Test normal strings, mixed-case strings, empty strings, None, and numbers.
3   # Then implement both functions with safe input handling so all tests pass.
4   import pytest
5   def to_uppercase(text):
6       if text is None:
7           return None
8       if not isinstance(text, str):
9           raise ValueError("Input must be a string")
10      return text.upper()
11
12  def to_lowercase(text):
13      if text is None:
14          return None
15      if not isinstance(text, str):
16          raise ValueError("Input must be a string")
17      return text.lower()
18
19  def test_to_uppercase():
20      assert to_uppercase("hello") == "HELLO"
21      assert to_uppercase("HeLLo") == "HELLO"
22      assert to_uppercase("") == ""
23      assert to_uppercase(None) is None
24      with pytest.raises(ValueError):
25          to_uppercase(123)
26
27  def test_to_lowercase():
28      assert to_lowercase("HELLO") == "hello"
29      assert to_lowercase("HeLLo") == "hello"
30      assert to_lowercase("") == ""
31      assert to_lowercase(None) is None
32      with pytest.raises(ValueError):
33          to_lowercase(123)
34
```

OUTPUT:

```
C:\Users\varun\OneDrive\Desktop\python>python -m pytest -v test_case.py
================================= test session starts =================================
platform win32 -- Python 3.12.0, pytest-8.4.2, pluggy-1.6.0 -- C:\Program Files\Python312\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\varun\OneDrive\Desktop\python
collected 2 items

test_case.py::test_to_uppercase PASSED                                          [ 50%]
test_case.py::test_test_to_lowercase PASSED                                     [100%]
================================== 2 passed in 0.03s ==================================
```

EXPLANATION:

The pytest test cases verify that the to_uppercase() and to_lowercase() functions correctly handle normal strings, mixed-case strings, empty strings, None values, and invalid inputs.

PROMPT:

Write Python assertion tests for to_uppercase(text) and to_lowercase(text) to check uppercase conversion, lowercase conversion, empty string handling, and invalid input handling.

CODE:

```python
1  # Write Python assertion tests for to_uppercase(text) and to_lowercase
2  def to_uppercase(text):
3      if not isinstance(text, str):
4          raise ValueError("Input must be a string")
5      return text.upper()
6  def to_lowercase(text):
7      if not isinstance(text, str):
8          raise ValueError("Input must be a string")
9      return text.lower()
10 def test_to_uppercase():
11     # Test uppercase conversion
12     assert to_uppercase("hello") == "HELLO"
13     assert to_uppercase("World") == "WORLD"
14     # Test empty string handling
15     assert to_uppercase("") == ""
16     # Test invalid input handling
17     try:
18         to_uppercase(123)
19         assert False, "Expected ValueError for non-string input"
20     except ValueError:
21         pass
22 def test_to_lowercase():
23     # Test lowercase conversion
24     assert to_lowercase("HELLO") == "hello"
25     assert to_lowercase("World") == "world"
26     # Test empty string handling
27     assert to_lowercase("") == ""
28     # Test invalid input handling
29     try:
30         to_lowercase(123)
31         assert False, "Expected ValueError for non-string input"
32     except ValueError:
33         pass
34 if __name__ == "__main__":
35     test_to_uppercase()
36     test_to_lowercase()
37     print("All tests passed!")
```

OUTPUT:

```
ns\ms-python.debugpy-2025.
All tests passed!
```

EXPLANATION:

This program has two functions: one converts text to uppercase and the other converts text to lowercase. It also includes tests to check if the functions work correctly for normal input, empty strings, and invalid input. If everything works, it prints "All tests passed!".

Task Description

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

PROMPT:

Create test cases for a Python function sum_list(numbers) that adds numbers in a list. The function should return 0 for an empty list, handle negative numbers, and ignore non-numeric values. Show example inputs and expected outputs.

CODE:

```
1   #Create test cases for a Python function sum_list(numbers) that adds n
2   # The function should return 0 for an empty list, handle negative numb
3   # Show example inputs and expected outputs.
4   import unittest
5   def sum_list(numbers):
6       total = 0
7       for num in numbers:
8           if isinstance(num, (int, float)):
9               total += num
10      return total
11  class TestSumList(unittest.TestCase):
12      def test_empty_list(self):
13          self.assertEqual(sum_list([]), 0)
14      def test_positive_numbers(self):
15          self.assertEqual(sum_list([1, 2, 3]), 6)
16      def test_negative_numbers(self):
17          self.assertEqual(sum_list([-1, -2, -3]), -6)
18      def test_mixed_numbers(self):
19          self.assertEqual(sum_list([-1, 0, 1]), 0)
20      def test_non_numeric_values(self):
21          self.assertEqual(sum_list([1, 'a', 2.5, None]), 3.5)
22  if __name__ == '__main__':
23      unittest.main()
```

OUTPUT:

```
.....
--------------------------------
Ran 5 tests in 0.001s

OK
```

EXPLANATION:

This program defines a function sum_list(numbers) that adds only numeric values (int or float) from a list and ignores other types. Then **unittest test cases** check different situations like empty lists, positive numbers, negative numbers, and mixed values.

PROMPT:

"Write test cases for a Python function sum_list(numbers) that returns the sum of elements in a list. The function should return 0 for an empty list, handle negative numbers correctly, and ignore non-numeric values. Include example inputs and expected outputs."

CODE:

```python
#"Create general test cases for a Python function sum_list(numbers) that ca
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
        else:
            raise ValueError("All elements must be numeric.")
    return total
# Test cases for sum_list function
# Test case 1: Normal list of numbers
input1 = [1, 2, 3, 4, 5]
expected_output1 = 15
# Test case 2: Empty list
input2 = []
expected_output2 = 0
# Test case 3: List with negative numbers
input3 = [-1, -2, -3, -4, -5]
expected_output3 = -15
# Test case 4: List with mixed values (positive and negative)
input4 = [1, -2, 3, -4, 5]
expected_output4 = 3
# Test case 5: List containing non-numeric elements
input5 = [1, 'two', 3, 'four', 5]
expected_output5 = "ValueError: All elements must be numeric."
# Running the test cases
try:
    assert sum_list(input1) == expected_output1
    print("Test case 1 passed.")
    assert sum_list(input2) == expected_output2
    print("Test case 2 passed.")
    assert sum_list(input3) == expected_output3
    print("Test case 3 passed.")
    assert sum_list(input4) == expected_output4
    print("Test case 4 passed.")
    sum_list(input5)
except AssertionError:
    print("A test case failed.")
except ValueError as e:
    if str(e) == expected_output5:
        print("Test case 5 passed.")
    else:
        print("Test case 5 failed.")
```

OUTPUT:

```
Test case 1 passed.
Test case 2 passed.
Test case 3 passed.
Test case 4 passed.
Test case 4 passed.
Test case 5 failed.
Test case 5 failed.
```

EXPLANATION:

This code creates a function sum_list() that adds only numeric values in a list. Then it runs 5 test cases using assert to check if the function works correctly. If a list contains non-numeric values, the function raises a ValueError, and the test checks for that.

PROMPT:

Create pytest test cases for a Python function sum_list(numbers) that returns the sum of numeric elements in a list. The function should return 0 for an empty list, handle negative numbers, and ignore non-numeric values.

CODE:

```python
1  # Create pytest test cases for a Python function sum_list(num
2  # The function should return 0 for an empty list, handle nega
3  import pytest
4  def sum_list(numbers):
5      total = 0
6      for num in numbers:
7          if isinstance(num, (int, float)):
8              total += num
9      return total
10 def test_sum_list_empty():
11     assert sum_list([]) == 0
12 def test_sum_list_positive_numbers():
13     assert sum_list([1, 2, 3]) == 6
14 def test_sum_list_negative_numbers():
15     assert sum_list([-1, -2, -3]) == -6
16 def test_sum_list_mixed_numbers():
17     assert sum_list([1, -2, 3]) == 2
18 def test_sum_list_non_numeric():
19     assert sum_list([1, 'a', 2, None, 3]) == 6
20 def test_sum_list_all_non_numeric():
21     assert sum_list(['a', None, 'b']) == 0
22 def test_sum_list_floats():
23     assert sum_list([1.5, 2.5, -1.0]) == 3.0
24 def test_sum_list_mixed_types():
25     assert sum_list([1, 'a', 2.5, None, -1]) == 2.5
26 if __name__ == "__main__":
27     pytest.main()
```

OUTPUT:

```
==================================== test session starts ====================================
platform win32 -- Python 3.12.0, pytest-8.4.2, pluggy-1.6.0
rootdir: C:\Users\varun\OneDrive\Desktop\python
collected 8 items

test_case.py ........                                                                  [100%]

==================================== 8 passed in 0.02s ====================================
PS C:\Users\varun\OneDrive\Desktop\python>
```

EXPLANATION:

This program defines a sum_list(numbers) function that adds only numeric values from a list and ignores non-numeric ones. The pytest test functions check different cases like empty lists, positive and negative numbers, floats, and mixed values to make sure the function works correctly.

PROMPT:

Write assertion tests for a Python function sum_list(numbers) that adds numeric values in a list. The function should return 0 for an empty list, handle negative numbers, and ignore non-numeric values.

CODE:

```python
# Write assertion tests for a Python function sum_list(numbers) that adds numeric values in a list.
# The function should return 0 for an empty list, handle negative numbers, and ignore non-numeric values.
def sum_list(numbers):
    total = 0
    for num in numbers:
        if isinstance(num, (int, float)):
            total += num
    return total
# Test cases for the sum_list function
# Test case 1: Basic functionality with positive numbers
assert sum_list([1, 2, 3]) == 6, "Test case 1 failed: Basic functionality with positive numbers not working correctly"
# Test case 2: Handling negative numbers
assert sum_list([-1, -2, -3]) == -6, "Test case 2 failed: Negative numbers not handled correctly"
# Test case 3: Handling a mix of positive and negative numbers
assert sum_list([1, -2, 3]) == 2, "Test case 3 failed: Mix of positive and negative numbers not handled correctly"
# Test case 4: Handling an empty list
assert sum_list([]) == 0, "Test case 4 failed: Empty list should return 0"
# Test case 5: Ignoring non-numeric values
assert sum_list([1, 'a', 2, None, 3]) == 6, "Test case 5 failed: Non-numeric values should be ignored"
# Test case 6: Handling a list with only non-numeric values
assert sum_list(['a', None, 'b']) == 0, "Test case 6 failed: List with only non-numeric values should return 0"
# Test case 7: Handling a list with mixed types (numeric and non-numeric)
assert sum_list([1, 'a', 2.5, None, 3]) == 6.5, "Test case 7 failed: Mixed types not handled correctly"
print("All test cases passed!")
```

OUTPUT:

```
All test cases passed!
```

EXPLANATION:

This program creates a function sum_list(numbers) that adds only numeric values (int and float) from a list and ignores other values. Below that, assert statements test different cases like positive numbers, negative numbers, empty lists, and lists with non-numeric values. If all tests pass, it prints "All test cases passed!".

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

PROMPT:

Generate simple manual test cases (not unittest) for a Python StudentResult class with methods: add_marks(mark), calculate_average(), and get_result().

 Marks must be between 0 and 100

 Average ≥ 40 → Pass

 Average < 40 → Fail

Invalid marks → Error

CODE:

```python
1   # Generate simple manual test cases (not unittest) for a Python StudentResult class with methods:
2   # add_marks(mark), calculate_average(), and get_result().
3   # Rules:
4   # Marks must be between 0 and 100
5   # Average ≥ 40 → Pass
6   # Average < 40 → Fail
7   # Invalid marks → Error
8   class StudentResult:
9       def __init__(self):
10          self.marks = []
11      def add_marks(self, mark):
12          if 0 <= mark <= 100:
13              self.marks.append(mark)
14          else:
15              raise ValueError("Invalid mark: must be between 0 and 100")
16      def calculate_average(self):
17          if not self.marks:
18              return 0
19          return sum(self.marks) / len(self.marks)
20      def get_result(self):
21          average = self.calculate_average()
22          return "Pass" if average >= 40 else "Fail"
23  # Test cases
24  # Test case 1: Valid marks
25  student1 = StudentResult()
26  student1.add_marks(85)
27  student1.add_marks(90)
28  print(f"Marks: {student1.marks} → Average: {student1.calculate_average()} → Result: {student1.get_result()}")
29  # Test case 2: Average below 40
30  student2 = StudentResult()
31  student2.add_marks(30)
32  student2.add_marks(35)
33  print(f"Marks: {student2.marks} → Average: {student2.calculate_average()} → Result: {student2.get_result()}")
34  # Test case 3: Invalid mark
35  student3 = StudentResult()
36  try:
37      student3.add_marks(105)
38  except ValueError as e:
39      print(f"Marks: [105] → Error: {e}")
40  # Test case 4: No marks added
41  student4 = StudentResult()
42  print(f"Marks: {student4.marks} → Average: {student4.calculate_average()} → Result: {student4.get_result()}")
43  # Test case 5: Mixed valid marks
44  student5 = StudentResult()
45  student5.add_marks(50)
46  student5.add_marks(30)
47  print(f"Marks: {student5.marks} → Average: {student5.calculate_average()} → Result: {student5.get_result()}")
48
```

OUTPUT:

```
Marks: [30, 35] → Average: 32.5 → Result: Fail
Marks: [105] → Error: Invalid mark: must be between 0 and 100
Marks: [] → Average: 0 → Result: Fail
Marks: [50, 30] → Average: 40.0 → Result: Pass
```

EXPLANATION:

The StudentResult class stores marks using add_marks() (only values between 0 and 100). The calculate_average() method computes the average as sum of marks ÷ number of marks, returning 0 if no marks exist. The get_result() method returns "Pass" if average ≥ 40, otherwise "Fail", and the test cases verify valid, invalid, and boundary inputs.

PROMPT:

Generate assertion test cases using Python assert statements for a StudentResult class with methods add_marks(), calculate_average(), and get_result(). Ensure marks are between 0 and 100, average is calculated correctly, and average ≥ 40 → Pass, otherwise Fail. Also include tests for invalid marks and empty marks list.

CODE:

```python
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_marks(self, mark):
        if 0 <= mark <= 100:
            self.marks.append(mark)
        else:
            raise ValueError("Marks must be between 0 and 100")
    def calculate_average(self):
        if not self.marks:
            raise ValueError("No marks to calculate average")
        return sum(self.marks) / len(self.marks)
    def get_result(self):
        average = self.calculate_average()
        return "Pass" if average >= 40 else "Fail"
# Test cases
def test_student_result():
    student = StudentResult()
    # Test adding valid marks
    student.add_marks(85)
    student.add_marks(70)
    assert student.marks == [85, 70], "Marks should be added correctly"
    # Test calculating average
    average = student.calculate_average()
    assert average == 77.5, "Average should be calculated correctly"
    # Test getting result
    result = student.get_result()
    assert result == "Pass", "Result should be Pass for average >= 40"
    # Test adding invalid marks
    try:
        student.add_marks(110)
        assert False, "Should raise ValueError for marks > 100"
    except ValueError:
        pass
    try:
        student.add_marks(-5)
        assert False, "Should raise ValueError for marks < 0"
    except ValueError:
        pass
    # Test calculating average with empty marks list
    empty_student = StudentResult()
    try:
        empty_student.calculate_average()
        assert False, "Should raise ValueError for empty marks list"
    except ValueError:
        pass
    # Test getting result with empty marks list
    try:
        empty_student.get_result()
        assert False, "Should raise ValueError for empty marks list"
    except ValueError:
        pass
# Run tests
test_student_result()
print("All tests passed!")
```

OUTPUT:

```
All tests passed!
```

EXPLANATION:

This program tests the StudentResult class using assert statements to make sure everything works correctly.It checks whether marks are added properly, the average is calculated using sum(marks) / len(marks), and the result is "Pass" when average ≥ 40, otherwise "Fail". It also verifies error handling by testing invalid marks (greater than 100 or less than 0) and calculating the average when no marks are added.

PROMPT:

Create unit test cases using Python's unittest framework to verify a StudentResult class that stores marks, calculates the average, and returns Pass if average ≥ 40, otherwise Fail. Include tests for adding marks, average calculation, invalid input errors, and no marks added.

CODE:

```python
import unittest
class StudentResult:
    def __init__(self):
        self.marks = []
    def add_mark(self, mark):
        if not isinstance(mark, (int, float)):
            raise ValueError("Mark must be a number.")
        if mark < 0 or mark > 100:
            raise ValueError("Mark must be between 0 and 100.")
        self.marks.append(mark)
    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()
        return "Pass" if average >= 40 else "Fail"
class TestStudentResult(unittest.TestCase):
    def setUp(self):
        self.student_result = StudentResult()
    def test_add_mark_valid(self):
        self.student_result.add_mark(85)
        self.assertEqual(self.student_result.marks, [85])
    def test_add_mark_invalid_type(self):
        with self.assertRaises(ValueError):
            self.student_result.add_mark("A")
    def test_add_mark_invalid_range(self):
        with self.assertRaises(ValueError):
            self.student_result.add_mark(-5)
        with self.assertRaises(ValueError):
            self.student_result.add_mark(105)
    def test_calculate_average_no_marks(self):
        self.assertEqual(self.student_result.calculate_average(), 0)
    def test_calculate_average_with_marks(self):
        self.student_result.add_mark(80)
        self.student_result.add_mark(90)
        self.assertEqual(self.student_result.calculate_average(), 85)
    def test_get_result_pass(self):
        self.student_result.add_mark(50)
        self.student_result.add_mark(60)
        self.assertEqual(self.student_result.get_result(), "Pass")
    def test_get_result_fail(self):
        self.student_result.add_mark(30)
        self.student_result.add_mark(20)
        self.assertEqual(self.student_result.get_result(), "Fail")
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
Ran 7 tests in 0.002s

OK
```

EXPLANATION:

This program uses Python's unittest framework to test the StudentResult class. The tests check whether marks are added correctly, invalid inputs raise errors, the average is calculated properly, and the result returns "Pass" when average ≥ 40 and "Fail" otherwise. The setUp() method creates a new StudentResult object before each test runs.

PROMPT:

Write pytest test cases for a Python class StudentResult that has the methods add_mark(mark), calculate_average(), and get_result().# Ensure marks are between 0 and 100, the average is calculated correctly, and average ≥ 40 returns "Pass", otherwise "Fail".# Include tests for valid marks, invalid input (raise ValueError), boundary values, and empty marks list.

CODE:

```python
1   import pytest
2   class StudentResult:
3       def __init__(self):
4           self.marks = []
5       def add_mark(self, mark):
6           if not isinstance(mark, (int, float)):
7               raise ValueError("Mark must be a number")
8           if mark < 0 or mark > 100:
9               raise ValueError("Mark must be between 0 and 100")
10          self.marks.append(mark)
11      def calculate_average(self):
12          if len(self.marks) == 0:
13              return 0
14          return sum(self.marks) / len(self.marks)
15      def get_result(self):
16          avg = self.calculate_average()
17          if avg >= 40:
18              return "Pass"
19          else:
20              return "Fail"
21  # Test valid marks
22  def test_add_single_mark():
23      student = StudentResult()
24      student.add_mark(75)
25      assert student.marks == [75]
26  def test_add_multiple_marks():
27      student = StudentResult()
28      student.add_mark(85)
29      student.add_mark(90)
30      assert student.marks == [85, 90]
31  def test_calculate_average():
```

OUTPUT:

EXPLANATION:

This program creates a StudentResult class to store marks, calculate the average, and decide whether a student Passes or Fails (average ≥ 40 → Pass). The add_mark() function checks that marks are numbers between 0 and 100. The pytest test cases check different situations like valid marks, invalid input, boundary values, and empty marks.

Task Description

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.

PROMPT:

Write Python test cases using print statements for a function is_valid_username(username) that checks if a username has at least 5 characters, no spaces, and only alphanumeric characters. Include tests for a valid username, a short username, a username with spaces, and a username with special characters

EXPLANATION:

```
2  def is_valid_username(username):
3      if len(username) < 5:
4          return False
5      if ' ' in username:
6          return False
7      if not username.isalnum():
8          return False
9      return True
10 # Test cases
11 print("=== Username Validation Tests ===")
12 print("Test 1 - Valid username 'john123':", is_valid_username("john123"))
13 print("Test 2 - Valid username 'user5':", is_valid_username("user5"))
14 print("Test 3 - Short username 'john':", is_valid_username("john"))
15 print("Test 4 - Username with space 'john doe':", is_valid_username("john doe"))
16 print("Test 5 - Username with special chars 'john@123':", is_valid_username("john@123"))
17 print("Test 6 - Username with dash 'john-123':", is_valid_username("john-123"))
18 print("Test 7 - Only numbers '12345':", is_valid_username("12345"))
19 print("Test 8 - Only letters 'abcde':", is_valid_username("abcde"))
20
```

OUTPUT:

```
Test 1 - Valid username 'john123': True
Test 2 - Valid username 'user5': True
Test 2 - Valid username 'user5': True
Test 3 - Short username 'john': False
Test 3 - Short username 'john': False
Test 4 - Username with space 'john doe': False
Test 4 - Username with space 'john doe': False
Test 5 - Username with special chars 'john@123': False
Test 6 - Username with dash 'john-123': False
```

EXPLANATION:

This code defines a function is_valid_username() that checks whether a username follows three rules: it must be at least 5 characters long, contain no spaces, and include only letters and numbers. After that, the print() statements test different usernames to show whether each one is valid (True) or invalid (False).

PROMPT:

Write Python unittest test cases for a function is_valid_username(username) that checks if a username has at least 5 characters, no spaces, and only alphanumeric characters. Include tests for a valid username, a username that is too short, a username with spaces, and a username with special characters.

CODE:

```python
import unittest
def is_valid_username(username):
    if len(username) < 5:
        return False
    if ' ' in username:
        return False
    if not username.isalnum():
        return False
    return True
class TestIsValidUsername(unittest.TestCase):
    def test_valid_username(self):
        """Test that a valid username returns True"""
        self.assertTrue(is_valid_username("john123"))
    def test_username_too_short(self):
        """Test that a username with less than 5 characters returns False"""
        self.assertFalse(is_valid_username("john"))
    def test_username_with_spaces(self):
        """Test that a username with spaces returns False"""
        self.assertFalse(is_valid_username("john doe"))
    def test_username_with_special_characters(self):
        """Test that a username with special characters returns False"""
        self.assertFalse(is_valid_username("john@123"))
    def test_valid_username_numbers(self):
        """Test that a valid username with only numbers returns True"""
        self.assertTrue(is_valid_username("12345"))
    def test_valid_username_letters(self):
        """Test that a valid username with only letters returns True"""
        self.assertTrue(is_valid_username("abcde"))
if __name__ == "__main__":
    unittest.main()
```

OUTPUT:

```
Ran 6 tests in 0.000s

OK
```

EXPLANATION:

This code defines a function is_valid_username() that checks whether a username is at least 5 characters long, has no spaces, and contains only letters or numbers. The unittest class TestIsValidUsername contains multiple test methods that verify the function using assertTrue() and assertFalse() for valid and invalid usernames. Finally, unittest.main() runs all the test cases automatically.