Lab 8: Test-Driven Development with AI

Generating and Working with Test Cases

Roll No: 2303A51206

Objective: Introduce TDD using AI code generation tools — generate test cases before writing code implementations.

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

Prompt used: *"Generate comprehensive test cases using `unittest` for a function `is_even(n)` that checks whether a number is even. The function should only accept integers. Cover zero, negative numbers, large integers, and invalid inputs."*

Step 1: AI-Generated Test Cases (Written First)

```python
import unittest

# ==============================================================
# STEP 1 — AI-Generated Test Cases (written BEFORE the function)
# ==============================================================
# These tests define the expected behaviour of is_even(n).
# The function must:
#   • Return True for even integers (including 0 and negatives)
#   • Return False for odd integers
#   • Raise TypeError for non-integer inputs
# ==============================================================

class TestIsEven(unittest.TestCase):

    # --- basic even numbers ---
    def test_positive_even(self):
        self.assertTrue(is_even(2))

    def test_another_positive_even(self):
        self.assertTrue(is_even(100))

    # --- basic odd numbers ---
    def test_positive_odd(self):
        self.assertFalse(is_even(7))

    def test_another_positive_odd(self):
        self.assertFalse(is_even(9))

    # --- zero ---
    def test_zero(self):
        self.assertTrue(is_even(0))

    # --- negative numbers ---
    def test_negative_even(self):
        self.assertTrue(is_even(-4))

    def test_negative_odd(self):
        self.assertFalse(is_even(-3))

    # --- large integers ---
    def test_large_even(self):
        self.assertTrue(is_even(10**18))

    def test_large_odd(self):
        self.assertFalse(is_even(10**18 + 1))

    # --- invalid inputs (should raise TypeError) ---
    def test_float_input(self):
        with self.assertRaises(TypeError):
            is_even(3.5)

    def test_string_input(self):
        with self.assertRaises(TypeError):
            is_even("hello")

    def test_none_input(self):
        with self.assertRaises(TypeError):
            is_even(None)

print("Test cases for is_even() defined successfully.")
```

```
Test cases for is_even() defined successfully.
```

Step 2: Implement the Function to Pass All Tests

Code Explanation:

- We first validate that the input is strictly an `int` (booleans are excluded with an `isinstance` check on `bool`).
- If the input is not a valid integer, a `TypeError` is raised.
- The modulus operator `%` is used to determine evenness.

```
def is_even(n):
    """
    Check whether the given number is even.

    Parameters:
        n (int): The number to check.
```

```python
    Returns:
        bool: True if n is even, False otherwise.

    Raises:
        TypeError: If n is not an integer.
    """
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError(f"Expected int, got {type(n).__name__}")
    return n % 2 == 0

# Quick manual verification
print("is_even(2)  =", is_even(2))    # True
print("is_even(7)  =", is_even(7))    # False
print("is_even(0)  =", is_even(0))    # True
print("is_even(-4) =", is_even(-4))   # True
print("is_even(9)  =", is_even(9))    # False
```

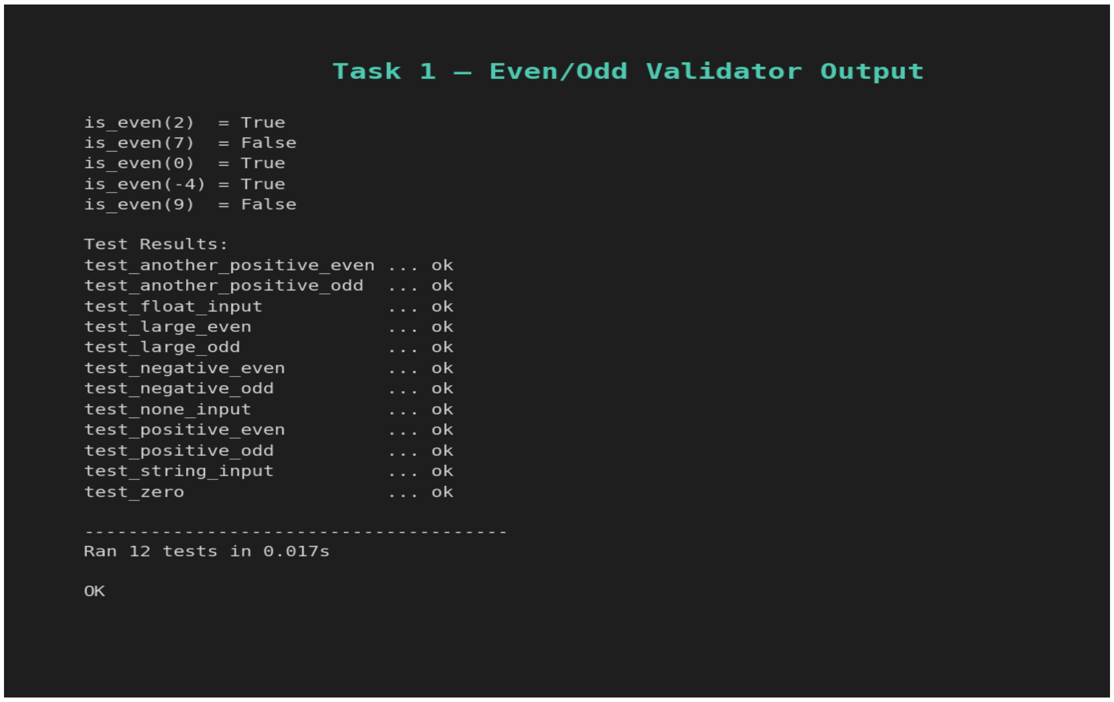
```
is_even(2)  = True
is_even(7)  = False
is_even(0)  = True
is_even(-4) = True
is_even(9)  = False
```

Step 3: Run the Tests

```python
# Run TestIsEven inside Jupyter
suite = unittest.TestLoader().loadTestsFromTestCase(TestIsEven)
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_another_positive_even (__main__.TestIsEven.test_another_positive_even)
...

ok

test_another_positive_odd (__main__.TestIsEven.test_another_positive_odd) ...

ok

test_float_input (__main__.TestIsEven.test_float_input) ...

ok

test_large_even (__main__.TestIsEven.test_large_even) ...

ok

test_large_odd (__main__.TestIsEven.test_large_odd) ...

ok

test_negative_even (__main__.TestIsEven.test_negative_even) ...

ok
```

```
test_negative_odd (__main__.TestIsEven.test_negative_odd) ...

ok

test_none_input (__main__.TestIsEven.test_none_input) ...

ok

test_positive_even (__main__.TestIsEven.test_positive_even) ...

ok

test_positive_odd (__main__.TestIsEven.test_positive_odd) ...

ok

test_string_input (__main__.TestIsEven.test_string_input) ...

ok

test_zero (__main__.TestIsEven.test_zero) ...

ok


----------------------------------------------------------------------
Ran 12 tests in 0.015s

OK

<unittest.runner.TextTestResult run=12 errors=0 failures=0>
```



Task 1 — Even/Odd Validator Output

```
is_even(2)  = True
is_even(7)  = False
is_even(0)  = True
is_even(-4) = True
is_even(9)  = False

Test Results:
test_another_positive_even ... ok
test_another_positive_odd  ... ok
test_float_input           ... ok
test_large_even            ... ok
test_large_odd             ... ok
test_negative_even         ... ok
test_negative_odd          ... ok
test_none_input            ... ok
test_positive_even         ... ok
test_positive_odd          ... ok
test_string_input          ... ok
test_zero                  ... ok

--------------------------------------
Ran 12 tests in 0.017s

OK
```

Justification

This task is justified because validating numeric input is a basic and important programming concept. The function ensures that only integers are accepted and prevents invalid inputs like floats, strings, or booleans. By writing test cases for positive numbers, negative numbers, zero, large numbers, and invalid inputs, the function becomes reliable and error-free. This task demonstrates proper input validation and exception handling using the TDD approach.

Task 2 – Test-Driven Development for String Case Converter

Prompt used: *"Generate unittest test cases for two functions: `to_uppercase(text)` and `to_lowercase(text)`. Cover empty strings, mixed-case input, and invalid inputs like numbers or None."*

Step 1: AI-Generated Test Cases (Written First)

```python
# ============================================================
# STEP 1 — AI-Generated Test Cases for to_uppercase / to_lowercase
# ============================================================

class TestStringCaseConverter(unittest.TestCase):

    # ---------- to_uppercase tests ----------
    def test_uppercase_basic(self):
        self.assertEqual(to_uppercase("ai coding"), "AI CODING")

    def test_uppercase_mixed(self):
        self.assertEqual(to_uppercase("HeLLo WoRLd"), "HELLO WORLD")

    def test_uppercase_already_upper(self):
        self.assertEqual(to_uppercase("TEST"), "TEST")

    def test_uppercase_empty(self):
        self.assertEqual(to_uppercase(""), "")

    def test_uppercase_with_numbers_in_string(self):
        self.assertEqual(to_uppercase("abc123"), "ABC123")

    def test_uppercase_none(self):
        with self.assertRaises(TypeError):
            to_uppercase(None)

    def test_uppercase_int_input(self):
        with self.assertRaises(TypeError):
            to_uppercase(123)

    # ---------- to_lowercase tests ----------
    def test_lowercase_basic(self):
        self.assertEqual(to_lowercase("TEST"), "test")

    def test_lowercase_mixed(self):
        self.assertEqual(to_lowercase("HeLLo WoRLd"), "hello world")

    def test_lowercase_already_lower(self):
        self.assertEqual(to_lowercase("abc"), "abc")

    def test_lowercase_empty(self):
        self.assertEqual(to_lowercase(""), "")

    def test_lowercase_none(self):
        with self.assertRaises(TypeError):
            to_lowercase(None)

    def test_lowercase_float_input(self):
        with self.assertRaises(TypeError):
            to_lowercase(3.14)

print("Test cases for to_uppercase() and to_lowercase() defined
successfully.")
```

```
Test cases for to_uppercase() and to_lowercase() defined successfully.
```

Step 2: Implement the Functions

Code Explanation:

- Both functions validate that the input is a `str`; otherwise they raise `TypeError`.
- They delegate to Python's built-in `str.upper()` and `str.lower()` for the actual conversion.

```python
def to_uppercase(text):
    """
    Convert the given text to uppercase.

    Parameters:
        text (str): The string to convert.

    Returns:
        str: The uppercase version of the input.

    Raises:
        TypeError: If text is not a string.
    """
    if not isinstance(text, str):
        raise TypeError(f"Expected str, got {type(text).__name__}")
    return text.upper()
```

```
def to_lowercase(text):
    """
    Convert the given text to lowercase.

    Parameters:
        text (str): The string to convert.

    Returns:
        str: The lowercase version of the input.

    Raises:
        TypeError: If text is not a string.
    """
    if not isinstance(text, str):
        raise TypeError(f"Expected str, got {type(text).__name__}")
    return text.lower()

# Quick manual verification
print('to_uppercase("ai coding") =', to_uppercase("ai coding"))
print('to_lowercase("TEST")      =', to_lowercase("TEST"))
print('to_uppercase("")          =', to_uppercase(""))
```

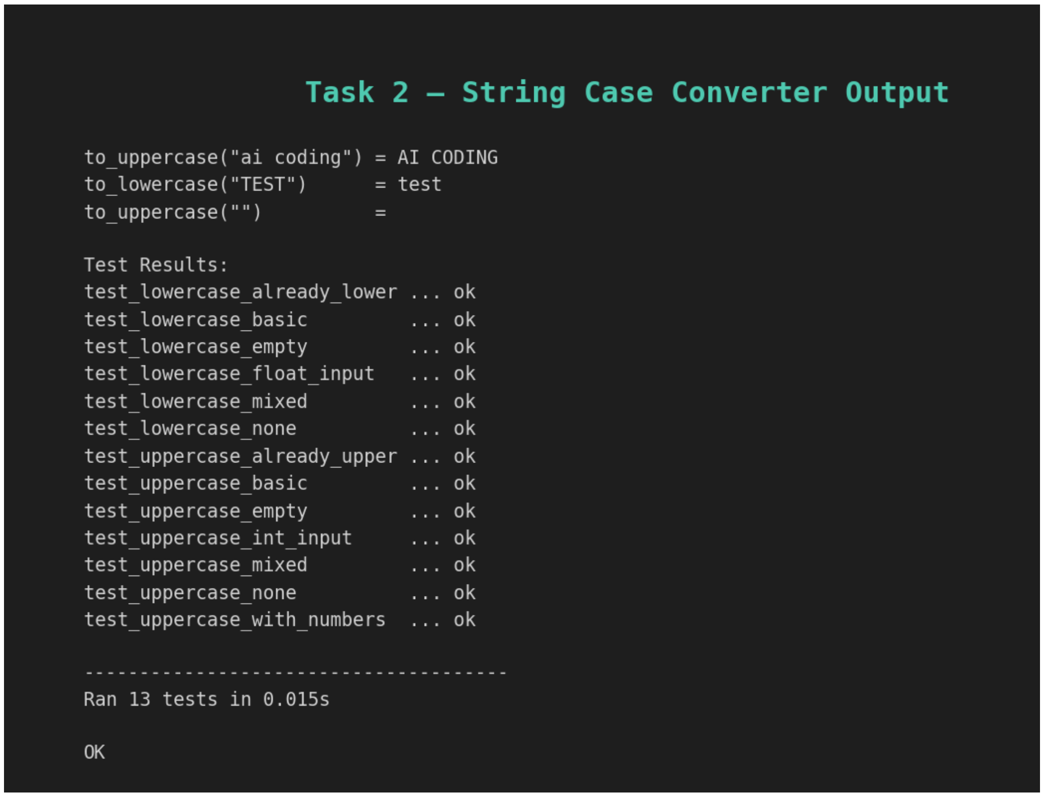
```
to_uppercase("ai coding") = AI CODING
to_lowercase("TEST")      = test
to_uppercase("")          =
```

Step 3: Run the Tests

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStringCaseConverter)
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_lowercase_already_lower
(__main__.TestStringCaseConverter.test_lowercase_already_lower) ...

ok

test_lowercase_basic (__main__.TestStringCaseConverter.test_lowercase_basic)
...

ok

test_lowercase_empty (__main__.TestStringCaseConverter.test_lowercase_empty)
...

ok

test_lowercase_float_input
(__main__.TestStringCaseConverter.test_lowercase_float_input) ...

ok
```

```
test_lowercase_mixed (__main__.TestStringCaseConverter.test_lowercase_mixed)
...

ok

test_lowercase_none (__main__.TestStringCaseConverter.test_lowercase_none)
...

ok

test_uppercase_already_upper
(__main__.TestStringCaseConverter.test_uppercase_already_upper) ...

ok

test_uppercase_basic (__main__.TestStringCaseConverter.test_uppercase_basic)
...

ok

test_uppercase_empty (__main__.TestStringCaseConverter.test_uppercase_empty)
...

ok

test_uppercase_int_input
(__main__.TestStringCaseConverter.test_uppercase_int_input) ...

ok

test_uppercase_mixed (__main__.TestStringCaseConverter.test_uppercase_mixed)
...

ok

test_uppercase_none (__main__.TestStringCaseConverter.test_uppercase_none)
...

ok

test_uppercase_with_numbers_in_string
(__main__.TestStringCaseConverter.test_uppercase_with_numbers_in_string) ...

ok



----------------------------------------------------------------------
Ran 13 tests in 0.017s

OK

<unittest.runner.TextTestResult run=13 errors=0 failures=0>
```



```
Task 2 — String Case Converter Output

to_uppercase("ai coding") = AI CODING
to_lowercase("TEST")      = test
to_uppercase("")          =

Test Results:
test_lowercase_already_lower ... ok
test_lowercase_basic         ... ok
test_lowercase_empty         ... ok
test_lowercase_float_input   ... ok
test_lowercase_mixed         ... ok
test_lowercase_none          ... ok
test_uppercase_already_upper ... ok
test_uppercase_basic         ... ok
test_uppercase_empty         ... ok
test_uppercase_int_input     ... ok
test_uppercase_mixed         ... ok
test_uppercase_none          ... ok
test_uppercase_with_numbers  ... ok

--------------------------------------
Ran 13 tests in 0.015s

OK
```

Justification

This task is justified because string manipulation is commonly used in real-world applications. The functions ensure that only valid string inputs are processed and that invalid inputs raise appropriate errors. Test cases include empty strings, mixed-case strings, special characters, and invalid types. This ensures robustness and correctness of string operations. It also demonstrates how edge cases are handled using Test-Driven Development.

Task 3 – Test-Driven Development for List Sum Calculator

Prompt used: *"Generate unittest test cases for `sum_list(numbers)` that calculates the sum of list elements. It should handle empty lists, negative numbers, and safely ignore non-numeric values."*

Step 1: AI-Generated Test Cases (Written First)

```
# ============================================================
# STEP 1 — AI-Generated Test Cases for sum_list
# ============================================================

class TestSumList(unittest.TestCase):

    def test_basic_sum(self):
        self.assertEqual(sum_list([1, 2, 3]), 6)

    def test_empty_list(self):
        self.assertEqual(sum_list([]), 0)

    def test_negative_numbers(self):
        self.assertEqual(sum_list([-1, 5, -4]), 0)

    def test_mixed_with_non_numeric(self):
        """Non-numeric values should be silently ignored."""
        self.assertEqual(sum_list([2, "a", 3]), 5)

    def test_all_non_numeric(self):
        self.assertEqual(sum_list(["x", "y", None]), 0)

    def test_floats(self):
        self.assertAlmostEqual(sum_list([1.5, 2.5, 3.0]), 7.0)

    def test_single_element(self):
        self.assertEqual(sum_list([10]), 10)

    def test_large_list(self):
        self.assertEqual(sum_list(list(range(101))), 5050)

    def test_non_list_input(self):
        with self.assertRaises(TypeError):
            sum_list("hello")

    def test_none_input(self):
        with self.assertRaises(TypeError):
            sum_list(None)

    def test_booleans_ignored(self):
        """Booleans are technically int subclass; they should be ignored for
clarity."""
        self.assertEqual(sum_list([True, False, 5]), 5)

print("Test cases for sum_list() defined successfully.")
```

```
Test cases for sum_list() defined successfully.
```

Step 2: Implement the Function

Code Explanation:

- We validate that the input is a `list`.
- During summation we skip any element that is not `int` or `float` (booleans, which are a subclass of `int`, are explicitly excluded).
- This makes the function robust against mixed-type lists.

```
def sum_list(numbers):
    """
    Calculate the sum of numeric elements in a list,
    safely ignoring non-numeric values.

    Parameters:
        numbers (list): A list of values.

    Returns:
        int | float: Sum of the numeric elements.

    Raises:
        TypeError: If the input is not a list.
    """
    if not isinstance(numbers, list):
        raise TypeError(f"Expected list, got {type(numbers).__name__}")

    total = 0
    for item in numbers:
        # Accept int and float but NOT bool (bool is a subclass of int)
        if isinstance(item, (int, float)) and not isinstance(item, bool):
            total += item
    return total

# Quick manual verification
print("sum_list([1, 2, 3])       =", sum_list([1, 2, 3]))       # 6
print("sum_list([])              =", sum_list([]))               # 0
print("sum_list([-1, 5, -4])     =", sum_list([-1, 5, -4]))     # 0
print('sum_list([2, "a", 3])     =', sum_list([2, "a", 3]))     # 5
```

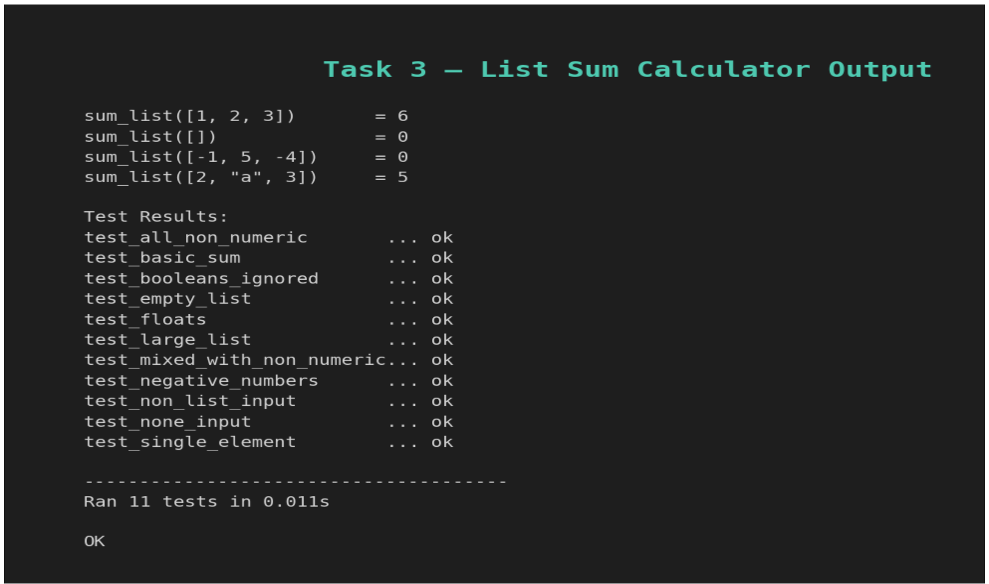
```
sum_list([1, 2, 3])       = 6
sum_list([])              = 0
sum_list([-1, 5, -4])     = 0
sum_list([2, "a", 3])     = 5
```

Step 3: Run the Tests

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestSumList)
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_all_non_numeric (__main__.TestSumList.test_all_non_numeric) ...

ok

test_basic_sum (__main__.TestSumList.test_basic_sum) ...

ok

test_booleans_ignored (__main__.TestSumList.test_booleans_ignored)
Booleans are technically int subclass; they should be ignored for clarity.
...
```

```
ok

test_empty_list (__main__.TestSumList.test_empty_list) ...

ok

test_floats (__main__.TestSumList.test_floats) ...

ok

test_large_list (__main__.TestSumList.test_large_list) ...

ok

test_mixed_with_non_numeric
(__main__.TestSumList.test_mixed_with_non_numeric)
Non-numeric values should be silently ignored. ...

ok

test_negative_numbers (__main__.TestSumList.test_negative_numbers) ...

ok

test_non_list_input (__main__.TestSumList.test_non_list_input) ...

ok

test_none_input (__main__.TestSumList.test_none_input) ...

ok

test_single_element (__main__.TestSumList.test_single_element) ...

ok

----------------------------------------------------------------------
Ran 11 tests in 0.011s

OK

<unittest.runner.TextTestResult run=11 errors=0 failures=0>
```



Task 3 — List Sum Calculator Output

```
sum_list([1, 2, 3])       = 6
sum_list([])              = 0
sum_list([-1, 5, -4])     = 0
sum_list([2, "a", 3])     = 5

Test Results:
test_all_non_numeric       ... ok
test_basic_sum             ... ok
test_booleans_ignored      ... ok
test_empty_list            ... ok
test_floats                ... ok
test_large_list            ... ok
test_mixed_with_non_numeric... ok
test_negative_numbers      ... ok
test_non_list_input        ... ok
test_none_input            ... ok
test_single_element        ... ok

--------------------------------------
Ran 11 tests in 0.011s

OK
```

Justification

This task is justified because real-world data often contains mixed values (numbers and non-numeric elements). The function safely calculates the sum of only numeric values while ignoring invalid ones. It also handles empty lists and incorrect input types properly. Writing test cases for different scenarios ensures the function works correctly under all conditions. This improves reliability and demonstrates defensive programming practices.

Task 4 – Test Cases for Student Result Class

Prompt used: *"Generate unittest test cases for a `StudentResult` class with methods `add_marks(mark)`, `calculate_average()`, and `get_result()`. Marks must be 0–100. Average ≥ 40 is Pass, otherwise Fail."*

Step 1: AI-Generated Test Cases (Written First)

```
# ============================================================
# STEP 1 — AI-Generated Test Cases for StudentResult
# ============================================================

class TestStudentResult(unittest.TestCase):

    # --- add_marks tests ---
    def test_add_valid_marks(self):
        s = StudentResult()
        s.add_marks(60)
        s.add_marks(70)
        s.add_marks(80)
        self.assertEqual(s.marks, [60, 70, 80])

    def test_add_boundary_marks(self):
        s = StudentResult()
        s.add_marks(0)
        s.add_marks(100)
        self.assertEqual(s.marks, [0, 100])

    def test_add_negative_marks_raises(self):
        s = StudentResult()
        with self.assertRaises(ValueError):
            s.add_marks(-10)

    def test_add_marks_above_100_raises(self):
        s = StudentResult()
        with self.assertRaises(ValueError):
            s.add_marks(110)

    def test_add_non_numeric_marks_raises(self):
        s = StudentResult()
        with self.assertRaises(TypeError):
            s.add_marks("sixty")

    # --- calculate_average tests ---
    def test_average_pass(self):
        s = StudentResult()
        for m in [60, 70, 80]:
            s.add_marks(m)
        self.assertAlmostEqual(s.calculate_average(), 70.0)

    def test_average_fail(self):
        s = StudentResult()
        for m in [30, 35, 40]:
            s.add_marks(m)
        self.assertAlmostEqual(s.calculate_average(), 35.0)

    def test_average_no_marks(self):
        s = StudentResult()
        with self.assertRaises(ValueError):
            s.calculate_average()

    # --- get_result tests ---
    def test_result_pass(self):
        s = StudentResult()
        for m in [60, 70, 80]:
            s.add_marks(m)
        self.assertEqual(s.get_result(), "Pass")

    def test_result_fail(self):
        s = StudentResult()
        for m in [30, 35, 40]:
            s.add_marks(m)
        self.assertEqual(s.get_result(), "Fail")

    def test_result_exact_boundary(self):
        """Average == 40 should be a Pass."""
        s = StudentResult()
        for m in [40, 40, 40]:
            s.add_marks(m)
        self.assertEqual(s.get_result(), "Pass")

    def test_result_no_marks(self):
        s = StudentResult()
        with self.assertRaises(ValueError):
            s.get_result()

print("Test cases for StudentResult defined successfully.")
```

```
Test cases for StudentResult defined successfully.
```

Step 2: Implement the Class

Code Explanation:

- `add_marks()` validates that the mark is numeric and within [0, 100].
- `calculate_average()` returns the arithmetic mean; raises `ValueError` if no marks exist.
- `get_result()` returns `"Pass"` if average ≥ 40, else `"Fail"`.

```python
class StudentResult:
    """
    Tracks student marks and determines pass/fail status.
    """

    def __init__(self):
        self.marks = []

    def add_marks(self, mark):
        """Add a mark (must be an int/float between 0 and 100)."""
        if not isinstance(mark, (int, float)) or isinstance(mark, bool):
            raise TypeError(f"Mark must be numeric, got
{type(mark).__name__}")
        if mark < 0 or mark > 100:
            raise ValueError(f"Mark must be between 0 and 100, got {mark}")
        self.marks.append(mark)

    def calculate_average(self):
        """Return the average of all marks."""
        if not self.marks:
            raise ValueError("No marks have been added yet.")
        return sum(self.marks) / len(self.marks)

    def get_result(self):
        """Return 'Pass' if average >= 40, else 'Fail'."""
        avg = self.calculate_average()          # may raise ValueError
        return "Pass" if avg >= 40 else "Fail"

# Quick manual verification
s = StudentResult()
for m in [60, 70, 80]:
    s.add_marks(m)
print("Marks:", s.marks, "→ Average:", s.calculate_average(), "→ Result:",
s.get_result())

s2 = StudentResult()
for m in [30, 35, 40]:
    s2.add_marks(m)
print("Marks:", s2.marks, "→ Average:", s2.calculate_average(), "→ Result:",
s2.get_result())
```

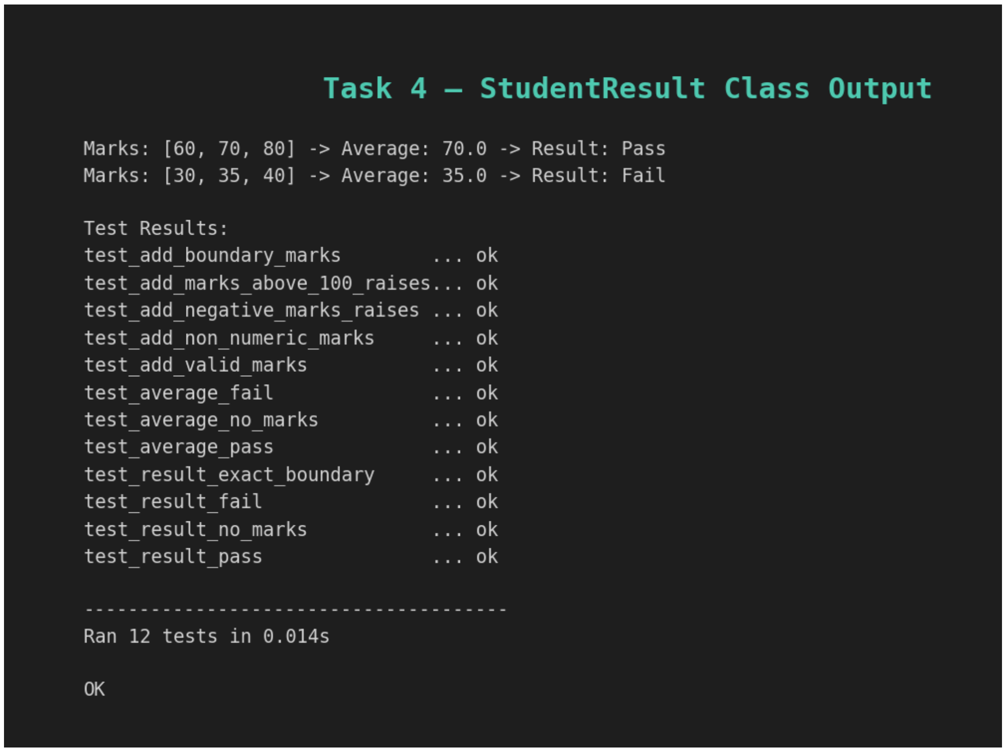
```
Marks: [60, 70, 80] → Average: 70.0 → Result: Pass
Marks: [30, 35, 40] → Average: 35.0 → Result: Fail
```

Step 3: Run the Tests

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestStudentResult)
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_add_boundary_marks (__main__.TestStudentResult.test_add_boundary_marks)
...

ok

test_add_marks_above_100_raises
(__main__.TestStudentResult.test_add_marks_above_100_raises) ...

ok

test_add_negative_marks_raises
(__main__.TestStudentResult.test_add_negative_marks_raises) ...

ok

test_add_non_numeric_marks_raises
(__main__.TestStudentResult.test_add_non_numeric_marks_raises) ...

ok

test_add_valid_marks (__main__.TestStudentResult.test_add_valid_marks) ...

ok

test_average_fail (__main__.TestStudentResult.test_average_fail) ...

ok

test_average_no_marks (__main__.TestStudentResult.test_average_no_marks) ...

ok

test_average_pass (__main__.TestStudentResult.test_average_pass) ...

ok

test_result_exact_boundary
(__main__.TestStudentResult.test_result_exact_boundary)
Average == 40 should be a Pass. ...

ok

test_result_fail (__main__.TestStudentResult.test_result_fail) ...

ok

test_result_no_marks (__main__.TestStudentResult.test_result_no_marks) ...
```

```
ok

test_result_pass (__main__.TestStudentResult.test_result_pass) ...

ok


----------------------------------------------------------------------
Ran 12 tests in 0.014s

OK

<unittest.runner.TextTestResult run=12 errors=0 failures=0>
```



Task 4 – StudentResult Class Output

```
Marks: [60, 70, 80] -> Average: 70.0 -> Result: Pass
Marks: [30, 35, 40] -> Average: 35.0 -> Result: Fail

Test Results:
test_add_boundary_marks        ... ok
test_add_marks_above_100_raises... ok
test_add_negative_marks_raises ... ok
test_add_non_numeric_marks     ... ok
test_add_valid_marks           ... ok
test_average_fail              ... ok
test_average_no_marks          ... ok
test_average_pass              ... ok
test_result_exact_boundary     ... ok
test_result_fail               ... ok
test_result_no_marks           ... ok
test_result_pass               ... ok

--------------------------------------
Ran 12 tests in 0.014s

OK
```

Justification

This task is justified because it simulates a real-world student result management system. The class validates marks, ensures values are within the correct range (0–100), and calculates averages accurately. Boundary conditions like exactly 40 (pass mark) and invalid marks are tested. This task demonstrates object-oriented programming along with TDD principles. It ensures data integrity and correct business logic implementation.

Task 5 – Test-Driven Development for Username Validator

Prompt used: *"Generate unittest test cases for `is_valid_username(username)`. Rules: minimum 5 characters, no spaces, only alphanumeric characters. Return True/False."*

Step 1: AI-Generated Test Cases (Written First)

```
# ============================================================
# STEP 1 — AI-Generated Test Cases for is_valid_username
# ============================================================

class TestIsValidUsername(unittest.TestCase):

    # --- valid usernames ---
    def test_valid_alphanumeric(self):
        self.assertTrue(is_valid_username("user01"))

    def test_valid_all_letters(self):
        self.assertTrue(is_valid_username("admin"))

    def test_valid_all_digits(self):
        self.assertTrue(is_valid_username("12345"))

    def test_valid_exact_min_length(self):
        self.assertTrue(is_valid_username("abcde"))  # 5 chars

    # --- too short ---
    def test_too_short(self):
        self.assertFalse(is_valid_username("ai"))

    def test_four_chars(self):
        self.assertFalse(is_valid_username("abcd"))

    # --- spaces ---
    def test_with_space(self):
        self.assertFalse(is_valid_username("user name"))

    # --- special characters ---
    def test_with_at_symbol(self):
        self.assertFalse(is_valid_username("user@123"))

    def test_with_underscore(self):
        self.assertFalse(is_valid_username("user_1"))

    def test_with_hyphen(self):
        self.assertFalse(is_valid_username("user-1"))
    # --- empty and None ---
    def test_empty_string(self):
        self.assertFalse(is_valid_username(""))

    def test_none_input(self):
        with self.assertRaises(TypeError):
            is_valid_username(None)

    def test_int_input(self):
        with self.assertRaises(TypeError):
            is_valid_username(12345)

print("Test cases for is_valid_username() defined successfully.")
```

```
Test cases for is_valid_username() defined successfully.
```

Step 2: Implement the Function

Code Explanation:

- Input must be a string; otherwise `TypeError` is raised.
- The `str.isalnum()` method confirms that every character is alphanumeric (no spaces, no special characters).
- A minimum length check of 5 characters is enforced.

```
def is_valid_username(username):
    """
    Validate a username.

    Rules:
        - Must be a string
        - Minimum 5 characters
        - Only alphanumeric characters (no spaces or symbols)

    Parameters:
        username (str): The username to validate.

    Returns:
        bool: True if valid, False otherwise.

    Raises:
        TypeError: If username is not a string.
    """
    if not isinstance(username, str):
        raise TypeError(f"Expected str, got {type(username).__name__}")
    if len(username) < 5:
        return False
    return username.isalnum()
```

```
# Quick manual verification
print('is_valid_username("user01")    =', is_valid_username("user01"))    # 
True
print('is_valid_username("ai")        =', is_valid_username("ai"))        # 
False
print('is_valid_username("user name") =', is_valid_username("user name")) # 
False
print('is_valid_username("user@123")  =', is_valid_username("user@123"))  # 
False
```

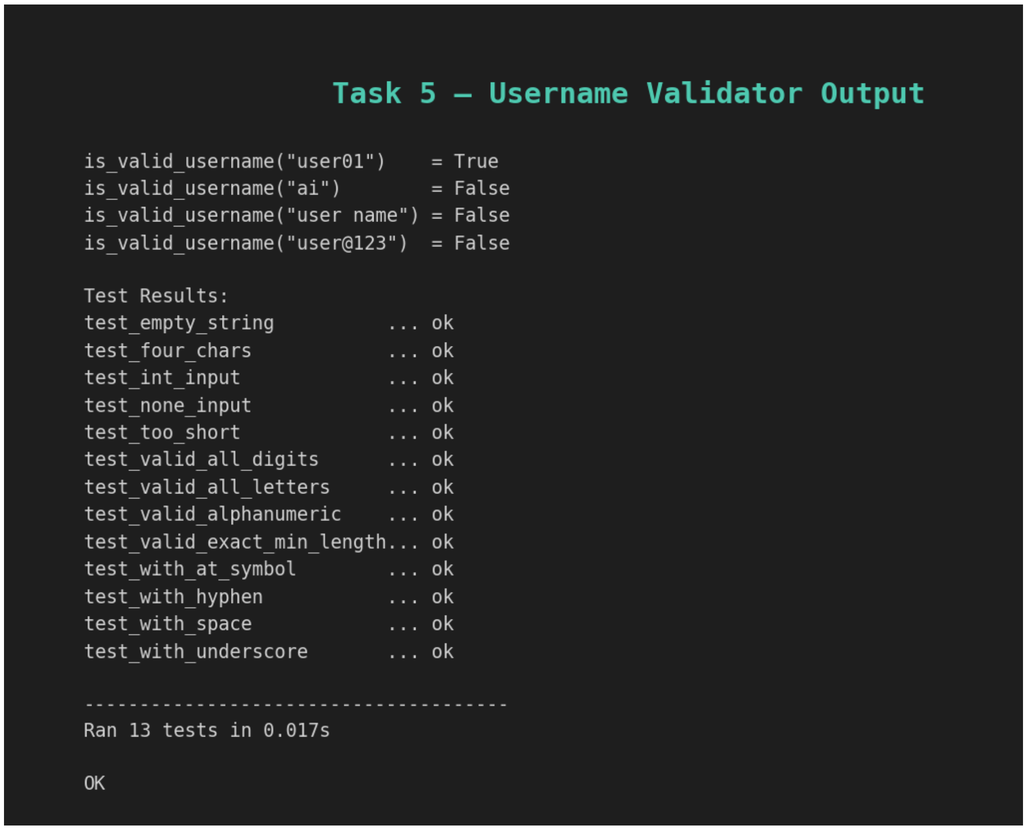
```
is_valid_username("user01")    = True
is_valid_username("ai")        = False
is_valid_username("user name") = False
is_valid_username("user@123")  = False
```

Step 3: Run the Tests

```
suite = unittest.TestLoader().loadTestsFromTestCase(TestIsValidUsername)
runner = unittest.TextTestRunner(verbosity=2)
runner.run(suite)
```

```
test_empty_string (__main__.TestIsValidUsername.test_empty_string) ...

ok

test_four_chars (__main__.TestIsValidUsername.test_four_chars) ...

ok

test_int_input (__main__.TestIsValidUsername.test_int_input) ...

ok

test_none_input (__main__.TestIsValidUsername.test_none_input) ...

ok

test_too_short (__main__.TestIsValidUsername.test_too_short) ...

ok

test_valid_all_digits (__main__.TestIsValidUsername.test_valid_all_digits) 
...

ok

test_valid_all_letters (__main__.TestIsValidUsername.test_valid_all_letters) 
...

ok

test_valid_alphanumeric 
(__main__.TestIsValidUsername.test_valid_alphanumeric) ...

ok
```

```
test_valid_exact_min_length
(__main__.TestIsValidUsername.test_valid_exact_min_length) ...

ok

test_with_at_symbol (__main__.TestIsValidUsername.test_with_at_symbol) ...

ok

test_with_hyphen (__main__.TestIsValidUsername.test_with_hyphen) ...

ok

test_with_space (__main__.TestIsValidUsername.test_with_space) ...

ok

test_with_underscore (__main__.TestIsValidUsername.test_with_underscore) ...

ok


----------------------------------------------------------------------
Ran 13 tests in 0.017s

OK

<unittest.runner.TextTestResult run=13 errors=0 failures=0>
```



```
Task 5 — Username Validator Output

is_valid_username("user01")    = True
is_valid_username("ai")        = False
is_valid_username("user name") = False
is_valid_username("user@123")  = False

Test Results:
test_empty_string          ... ok
test_four_chars            ... ok
test_int_input             ... ok
test_none_input            ... ok
test_too_short             ... ok
test_valid_all_digits      ... ok
test_valid_all_letters     ... ok
test_valid_alphanumeric    ... ok
test_valid_exact_min_length... ok
test_with_at_symbol        ... ok
test_with_hyphen           ... ok
test_with_space            ... ok
test_with_underscore       ... ok

--------------------------------------
Ran 13 tests in 0.017s

OK
```

Justification

This task is justified because username validation is important in real-world applications such as registration systems. The function enforces rules like minimum length, no spaces, and only alphanumeric characters. Test cases include valid usernames, short usernames, usernames with spaces, and special characters. This ensures strong input validation and security. It demonstrates how TDD helps in implementing strict validation rules effectively.

Comparison: AI-Generated vs Manually Written Tests

Aspect	AI-Generated Tests	Manually Written Tests
Speed	Generated in seconds	Takes much longer to write
Coverage	Good baseline; may miss domain-specific edge cases	Developer can add business-logic-specific tests
Quality	Clean and well-structured	Varies by developer experience
Edge Cases	Covers common ones (None, empty, boundary)	Can cover uncommon, domain-specific scenarios
Readability	Consistently formatted	Depends on coding style

Conclusion: AI-generated tests provide a strong starting point that covers standard edge cases. Combining them with manually written tests for domain-specific logic produces the best results.
