**Course Name:Competitive Programming**

**Name:Swetha Dixit**

**Batch:33**

**Hall-Ticket:2303A52415**

**Assignment:8.2**

**Task-1:**
**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

```
import unittest

from is_even import is_even


class TestIsEven(unittest.TestCase):
    # Given examples
    def test_examples(self):
        self.assertTrue(is_even(2))
        self.assertFalse(is_even(7))
        self.assertTrue(is_even(0))
        self.assertTrue(is_even(-4))
        self.assertFalse(is_even(9))

    # Zero handling
    def test_zero(self):
        self.assertTrue(is_even(0))

    # Positive integers
    def test_positive_even_and_odd(self):
        self.assertTrue(is_even(10))
        self.assertFalse(is_even(11))

    # Negative integers
    def test_negative_even_and_odd(self):
        self.assertTrue(is_even(-10))
        self.assertFalse(is_even(-11))

    # Very large integers
    def test_large_integers(self):
        self.assertTrue(is_even(10**18))          # large even
        self.assertFalse(is_even(10**18 + 1))     # large odd
        self.assertTrue(is_even(-(10**18)))       # large negative even
        self.assertFalse(is_even(-(10**18 + 1)))  # large negative odd

    # Input should only be integers
```

```
    def test_rejects_non_integers(self):
        for bad in (2.0, "2", None, [2], (2,), {"n": 2}):
            with self.subTest(bad=bad):
                with self.assertRaises(TypeError):
                    is_even(bad)

    # bool is a subclass of int in Python; reject it explicitly
    def test_rejects_bool(self):
        with self.assertRaises(TypeError):
            is_even(True)
        with self.assertRaises(TypeError):
            is_even(False)


if __name__ == "__main__":
    unittest.main()
```

## Task-2:

## 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.

```
import unittest

# --- Implementation ---

def to_uppercase(text):
    """
    Converts a string to uppercase.
    Raises TypeError if input is not a string.
    """
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.upper()

def to_lowercase(text):
    """
    Converts a string to lowercase.
    Raises TypeError if input is not a string.
    """
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.lower()

# --- Test Suite ---

class TestStringConverters(unittest.TestCase):

    # -- Tests for to_uppercase --
    def test_upper_basic(self):
        self.assertEqual(to_uppercase("ai coding"), "AI CODING")

    def test_upper_mixed(self):
        self.assertEqual(to_uppercase("PyThon"), "PYTHON")

    def test_upper_empty(self):
        self.assertEqual(to_uppercase(""), "")
```

```
    def test_upper_invalid_input(self):
        # Should raise TypeError for None or Numbers
        with self.assertRaises(TypeError):
            to_uppercase(None)
        with self.assertRaises(TypeError):
            to_uppercase(123)

    # -- Tests for to_lowercase --
    def test_lower_basic(self):
        self.assertEqual(to_lowercase("TEST"), "test")

    def test_lower_mixed(self):
        self.assertEqual(to_lowercase("HeLLo"), "hello")

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

    def test_lower_invalid_input(self):
        with self.assertRaises(TypeError):
            to_lowercase(None)
        with self.assertRaises(TypeError):
            to_lowercase(45.67)

# --- Run Tests ---
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

## 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

**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

```
import unittest

# --- Implementation ---

def sum_list(numbers):
    """
    Calculates the sum of a list of numbers.
    Ignores non-numeric values (strings, None, etc.) to prevent errors.
    """
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list")

    total = 0
    for item in numbers:
        # Check if the item is an integer or a float
        if isinstance(item, (int, float)):
            total += item
    return total

# --- Test Suite ---

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_types(self):
        # The function should ignore "a" and sum 2 + 3
        self.assertEqual(sum_list([2, "a", 3]), 5)

    def test_all_invalid(self):
        # Should return 0 if no numbers are present
        self.assertEqual(sum_list(["a", "b", None]), 0)

    def test_floats(self):
        self.assertEqual(sum_list([1.5, 2.5]), 4.0)

    def test_not_a_list(self):
        # Should raise error if input is not a list at all
        with self.assertRaises(TypeError):
            sum_list(123)

# --- Run Tests ---
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

## Task-4:Task Description

## 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**

```
import unittest

# --- Implementation ---

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

    def add_mark(self, mark):
        """
        Adds a mark to the student's record.
        Validates that the mark is between 0 and 100.
        """
        if not isinstance(mark, (int, float)):
            raise TypeError("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):
        """Calculates the average of the stored marks."""
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)

    def get_result(self):
        """Returns 'Pass' if average is >= 40, else 'Fail'."""
        avg = self.calculate_average()
```

```
        return "Pass" if avg >= 40 else "Fail"

# --- Test Suite ---

class TestStudentResult(unittest.TestCase):

    def setUp(self):
        # Create a new instance before every test
        self.student = StudentResult()

    def test_pass_scenario(self):
        self.student.add_mark(60)
        self.student.add_mark(70)
        self.student.add_mark(80)
        self.assertEqual(self.student.calculate_average(), 70)
        self.assertEqual(self.student.get_result(), "Pass")

    def test_fail_scenario(self):
        self.student.add_mark(30)
        self.student.add_mark(35)
        self.student.add_mark(40)
        self.assertEqual(self.student.calculate_average(), 35)
        self.assertEqual(self.student.get_result(), "Fail")

    def test_invalid_mark_negative(self):
        # Should raise ValueError for negative marks
        with self.assertRaises(ValueError):
            self.student.add_mark(-10)

    def test_invalid_mark_too_high(self):
        # Should raise ValueError for marks > 100
        with self.assertRaises(ValueError):
            self.student.add_mark(105)

    def test_empty_marks(self):
        # Average of 0 marks should be 0 and result Fail
        self.assertEqual(self.student.calculate_average(), 0)
        self.assertEqual(self.student.get_result(), "Fail")

    def test_boundary_pass(self):
```

```
        # Exactly 40 should be a Pass
        self.student.add_mark(40)
        self.assertEqual(self.student.get_result(), "Pass")

# --- Run Tests ---
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

## Task-5:

## 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.

```
import unittest

# --- Implementation ---

def is_valid_username(username):
    """
    Validates a username based on length and character content.
    Returns True if valid, False otherwise.
    """
    # 1. Check if input is a string
    if not isinstance(username, str):
        return False
```

```python
    # 2. Check minimum length
    if len(username) < 5:
        return False

    # 3. Check for spaces explicitly (though isalnum handles this,
    # it's good to be explicit for the requirements)
    if " " in username:
        return False

    # 4. Check for alphanumeric (letters and numbers only)
    # This rejects special chars like @, #, and spaces
    if not username.isalnum():
        return False

    return True

# --- Test Suite ---

class TestUsernameValidator(unittest.TestCase):

    def test_valid_username(self):
        self.assertTrue(is_valid_username("user01"))
        self.assertTrue(is_valid_username("PythonDev"))

    def test_too_short(self):
        self.assertFalse(is_valid_username("ai"))
        self.assertFalse(is_valid_username("1234"))

    def test_contains_space(self):
        self.assertFalse(is_valid_username("user name"))
        self.assertFalse(is_valid_username(" user1"))

    def test_special_characters(self):
        self.assertFalse(is_valid_username("user@123"))
        self.assertFalse(is_valid_username("admin!"))

    def test_empty_string(self):
        self.assertFalse(is_valid_username(""))
```

```python
    def test_non_string_input(self):
        # Should return False for None or numbers
        self.assertFalse(is_valid_username(None))
        self.assertFalse(is_valid_username(123456))

# --- Run Tests ---
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_1_8.2.py"
.......
----------------------------------------------------------------------
Ran 7 tests in 0.002s

OK
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_2_8.2.py"
........
```

```
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_2_8.2.py"
........
----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_3_8.2.py"
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s
```

powersh...
C Comp...
Python

```
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_2_8.2.py"
OK
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv
intec/Ai coding/Assignment 4.2/Task_3_8.2.py"
.......
----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
(.venv) PS C:\Users\intec\Ai coding> & "C:/Users/intec/Ai coding/.venv/Scripts/python.exe" "c:/Users/
intec/Ai coding/Assignment 4.2/Task_3_8.2.py"
```

powersh...
C Comp...
Python

Navigate to Command
Scroll to Previous Command (Ctrl+UpArrow)
Scroll to Next Command (Ctrl+DownArrow)

Ln 60, Col 61 (1841 selected) Spaces: 4 UTF-8 CRLF { } Python 3.14.0 (.venv