Name: T.Snuhith

Roll Number: 2303A510H9

Batch – 03

AI Assisted Coding

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

Input:-

```python
def is_even(n):
    """
    Determines if a number is even.

    Args:
        n: An integer value

    Returns:
        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("Input must be an integer")
    return n % 2 == 0


class TestIsEven(unittest.TestCase):
    """Test cases for is_even() function"""

    def test_even_positive_number(self):
        """Test even positive numbers"""
        self.assertTrue(is_even(2))
        self.assertTrue(is_even(4))
        self.assertTrue(is_even(100))

    def test_odd_positive_number(self):
        """Test odd positive numbers"""
        self.assertFalse(is_even(7))
        self.assertFalse(is_even(1))
        self.assertFalse(is_even(99))

    def test_zero(self):
        """Test zero is even"""
        self.assertTrue(is_even(0))

    def test_even_negative_number(self):
        """Test even negative numbers"""
        self.assertTrue(is_even(-4))
        self.assertTrue(is_even(-2))
        self.assertTrue(is_even(-100))

    def test_odd_negative_number(self):
```

Output:-

```
--------------------------------------------------------------------------------
TASK 1: Even/Odd Number Validator
--------------------------------------------------------------------------------

Function: is_even(n)
Purpose: Determines if a number is even
Requirements: Input must be an integer

Test Scenarios:
  is_even(       2) =>  True
  is_even(       7) => False
  is_even(       0) =>  True
  is_even(      -4) =>  True
  is_even(       9) => False
  is_even( 1000000) =>  True
```

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

• 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.

Input:-

```python
def to_uppercase(text):
    """
    Converts text to uppercase.

    Args:
        text: A string value

    Returns:
        Uppercase version of the string

    Raises:
        TypeError: If text is not a string
    """
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.upper()


def to_lowercase(text):
    """
    Converts text to lowercase.

    Args:
        text: A string value

    Returns:
        Lowercase version of the string

    Raises:
        TypeError: If text is not a string or None
    """
    if text is None:
        raise TypeError("Input cannot be None")
    if not isinstance(text, str):
        raise TypeError("Input must be a string")
    return text.lower()


class TestStringConverters(unittest.TestCase):
    """Test cases for string conversion functions"""

    def test_to_uppercase_basic(self):
        """Test basic uppercase conversion"""
        self.assertEqual(to_uppercase("ai coding"), "AI CODING")
        self.assertEqual(to_uppercase("hello"), "HELLO")
```

n File (ai) Indexing completed. OVR Ln 716, Col 2 Spaces: 4 UTF-8 CRLF { } Python

Output:-

```
--------------------------------------------------------------------------------
TASK 2: String Case Converter
--------------------------------------------------------------------------------

Function 1: to_uppercase(text)
Purpose: Converts text to uppercase
Requirements: Handle empty strings and mixed-case input

Test Scenarios:
  to_uppercase("ai coding") => "AI CODING"
  to_uppercase("") => ""
  to_uppercase("HeLLo WoRLd") => "HELLO WORLD"
  to_uppercase("test123") => "TEST123"


Function 2: to_lowercase(text)
Purpose: Converts text to lowercase
Requirements: Handle empty strings and invalid inputs

Test Scenarios:
  to_lowercase("TEST") => "test"
  to_lowercase("") => ""
  to_lowercase("HeLLo WoRLd") => "hello world"
  to_lowercase("PYTHON123") => "python123"
```

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

• 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.

Input:-

```python
def sum_list(numbers):
    """
    Calculates the sum of numeric values in a list.

    Args:
        numbers: A list that may contain numbers and non-numeric values

    Returns:
        Sum of all numeric values in the list

    Raises:
        TypeError: If input is not a list
    """
    if not isinstance(numbers, list):
        raise TypeError("Input must be a list")

    total = 0
    for item in numbers:
        if isinstance(item, (int, float)) and not isinstance(item, bool):
            total += item
    return total


class TestSumList(unittest.TestCase):
    """Test cases for sum_list() function"""

    def test_sum_positive_numbers(self):
        """Test sum of positive numbers"""
        self.assertEqual(sum_list([1, 2, 3]), 6)
        self.assertEqual(sum_list([10, 20, 30]), 60)

    def test_sum_empty_list(self):
        """Test sum of empty list"""
        self.assertEqual(sum_list([]), 0)

    def test_sum_negative_numbers(self):
        """Test sum with negative numbers"""
        self.assertEqual(sum_list([-1, 5, -4]), 0)
        self.assertEqual(sum_list([-10, -20]), -30)

    def test_sum_mixed_positive_negative(self):
        """Test sum with mixed positive and negative"""
        self.assertEqual(sum_list([1, -1, 5, -3]), 2)

    def test_sum_with_non_numeric_values(self):
```

Output:-

```
--------------------------------------------------------------------------------
TASK 3: List Sum Calculator
--------------------------------------------------------------------------------

Function: sum_list(numbers)
Purpose: Calculates sum of numeric values in a list
Requirements: Handle empty lists, ignore non-numeric values

Test Scenarios:
  sum_list([1, 2, 3]                    ) =>      6  # Positive numbers
  sum_list([]                           ) =>      0  # Empty list
  sum_list([-1, 5, -4]                  ) =>      0  # Mixed positive and negative
  sum_list([2, 'a', 3]                  ) =>      5  # Mixed with non-numeric
  sum_list([1.5, 2.5, 1.0]              ) =>    5.0  # Float values
```

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

• 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

Input:-

```python
class StudentResult:
    """
    Class to manage student marks and calculate results.
    """

    def __init__(self):
        """Initialize with empty marks list"""
        self.marks = []

    def add_marks(self, mark):
        """
        Add a mark to the student's marks list.

        Args:
            mark: A numeric value between 0 and 100

        Raises:
            ValueError: If mark is not between 0 and 100
            TypeError: If mark is not numeric
        """
        if not isinstance(mark, (int, float)) or isinstance(mark, bool):
            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):
        """
        Calculate the average of all marks.

        Returns:
            Average of marks, or 0 if no marks added
        """
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)

    def get_result(self):
        """
        Get the result based on average marks.

        Returns:
            "Pass" if average >= 40, "Fail" otherwise
        """
        average = self.calculate_average()
        return "Pass" if average >= 40 else "Fail"


class TestStudentResult(unittest.TestCase):
    """Test cases for StudentResult class"""

    def setUp(self):
        """Create a new StudentResult instance for each test"""
        self.student = StudentResult()

    def test_add_valid_marks(self):
        """Test adding valid marks"""
        self.student.add_marks(60)
        self.student.add_marks(70)
        self.student.add_marks(80)
        self.assertEqual(self.student.marks, [60, 70, 80])

    def test_add_marks_zero(self):
        """Test adding zero mark"""
        self.student.add_marks(0)
        self.assertEqual(self.student.marks, [0])

    def test_add_marks_hundred(self):
        """Test adding 100 mark"""
        self.student.add_marks(100)
        self.assertEqual(self.student.marks, [100])

    def test_add_marks_float(self):
        """Test adding float marks"""
        self.student.add_marks(75.5)
        self.assertEqual(self.student.marks, [75.5])
```

Output:-

```
--------------------------------------------------------------------------------
TASK 4: Student Result Class
--------------------------------------------------------------------------------

Class: StudentResult
Methods: add_marks(mark), calculate_average(), get_result()
Requirements: Marks 0-100, Pass if average >= 40

--- Scenario 1: Pass (Average: 70) ---
  Marks: [60, 70, 80]
  Average: 70.0
  Result: Pass

--- Scenario 2: Fail (Average: 35) ---
  Marks: [30, 35, 40]
  Average: 35.0
  Result: Fail

--- Scenario 3: Boundary Test (Average: 40 = Pass) ---
  Marks: [40]
  Average: 40.0
  Result: Pass
```

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

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.

Input:-

```python
def is_valid_username(username):
    """
    Validates a username based on specific criteria.

    Requirements:
    - Minimum length: 5 characters
    - No spaces allowed
    - Only alphanumeric characters allowed

    Args:
        username: A string to validate

    Returns:
        True if username is valid, False otherwise

    Raises:
        TypeError: If username is not a string
    """
    if not isinstance(username, str):
        raise TypeError("Username must be a string")

    # Check minimun length
    if len(username) < 5:
        return False

    # Check for spaces
    if " " in username:
        return False

    # Check if only alphanumeric
    if not username.isalnum():
        return False

    return True


class TestUsernameValidator(unittest.TestCase):
    """Test cases for username validation"""

    def test_valid_username_basic(self):
        """Test basic valid username"""
        self.assertTrue(is_valid_username("user01"))
        self.assertTrue(is_valid_username("john12"))

    def test_valid_username_all_letters(self):
        """Test valid username with all letters"""
        self.assertTrue(is_valid_username("username"))
        self.assertTrue(is_valid_username("alice"))

    def test_valid_username_all_numbers(self):
        """Test valid username with all numbers"""
        self.assertTrue(is_valid_username("12345"))
        self.assertTrue(is_valid_username("00000"))

    def test_valid_username_mixed(self):
        """Test valid username with mixed alphanumeric"""
        self.assertTrue(is_valid_username("user123abc"))
        self.assertTrue(is_valid_username("test5"))

    def test_valid_username_exactly_5_chars(self):
        """Test valid username with exactly 5 characters"""
        self.assertTrue(is_valid_username("user1"))
        self.assertTrue(is_valid_username("ab123"))

    def test_valid_username_long(self):
        """Test valid username that is long"""
        self.assertTrue(is_valid_username("verylongusername123"))

    def test_invalid_username_too_short(self):
        """Test invalid username - too short"""
        self.assertFalse(is_valid_username("ai"))
        self.assertFalse(is_valid_username("user"))
        self.assertFalse(is_valid_username(""))
        self.assertFalse(is_valid_username("u123"))

    def test_invalid_username_with_spaces(self):
        """Test invalid username - contains spaces"""
        self.assertFalse(is_valid_username("user name"))
        self.assertFalse(is_valid_username("user 123"))
```

Output :-

```
--------------------------------------------------------------------------------
TASK 5: Username Validator
--------------------------------------------------------------------------------

Function: is_valid_username(username)
Requirements:
  - Minimum length: 5 characters
  - No spaces allowed
  - Only alphanumeric characters

Test Scenarios:
  is_valid_username("user01         ") => VALID
  is_valid_username("ai             ") => INVALID
  is_valid_username("user name      ") => INVALID
  is_valid_username("user@123       ") => INVALID
  is_valid_username("alice          ") => VALID
  is_valid_username("john12         ") => VALID
  is_valid_username("test           ") => INVALID
  is_valid_username("username123    ") => VALID

================================================================================
END OF DEMONSTRATION OUTPUT
================================================================================
```

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.