AI ASSISTANT CODING ASSIGNMENT – 8.2

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**AcademicYear:**2025-2026
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s)Name	Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpulasumalatha Mr. Bingi Raju		
CourseCode	23CS002PC304	**CourseTitle**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	Week4 - Tuesday	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	23CSBTB01 To 23CSBTB52
AssignmentNumber:8.2(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*ExpectedTime to complete*
1	**Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases** **Lab Objectives:**	Week4 - Wednesday

- To introduce students to test-driven development (TDD) using AI code generation tools.
- To enable the generation of test cases before writing code implementations.
- To reinforce the importance of testing, validation, and error handling.
- To encourage writing clean and reliable code based on AI-generated test expectations.

Lab Outcomes (LOs):
After completing this lab, students will be able to:

- Use AI tools to write test cases for Python functions and classes.
- Implement functions based on test cases in a test-first development style.
- Use unittest or pytest to validate code correctness.
- Analyze the completeness and coverage of AI-generated tests.
- Compare AI-generated and manually written test cases for quality and logic

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 cases

Task Description

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.

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.

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

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.

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

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

Prompt:

Write a set of pytest test cases for a function is_even(n). The tests should cover positive even and odd numbers, zero, negative numbers, and very large integers. Additionally, include test cases for invalid inputs like floats, strings, and None, where the expected result is -1. Use assert is_even(x) is True/False for boolean checks and == -1 for invalid inputs.

Code:

```python
import pytest

# The function being tested
def is_even(n):
    if not isinstance(n, int):
        return -1
    return n % 2 == 0

# Test cases extracted from your image
def test_positive_even():
    assert is_even(2) is True

def test_positive_odd():
    assert is_even(7) is False

def test_zero():
    assert is_even(0) is True

def test_negative_even():
    assert is_even(-4) is True

def test_negative_odd():
    assert is_even(-9) is False

def test_large_even_number():
    assert is_even(10**18) is True

def test_large_odd_number():
    assert is_even(10**18 + 1) is False

def test_float_input():
    assert is_even(2.5) == -1

def test_string_input():
    assert is_even("10") == -1

def test_none_input():
    assert is_even(None) == -1

if __name__ == "__main__":
    # Example of user interaction
    val = input("Enter a number: ")
    try:
        # Attempt to convert to int to simulate numeric input
        num = int(val)
        print(f"Is Even: {is_even(num)}")
    except ValueError:
        # If it's not an int, pass the raw string to show the -1 handling
        print(f"Is Even: {is_even(val)}")
```

Sample Input/Output:

```
Enter a number: 5
Is Even: False
```

Code Explanation:

- **Logic:** The code uses the modulo operator % to determine if a number is even.
- **Data Validation:** It specifically checks if the input is an integer using isinstance().
- **Error Handling:** Per your screenshot, if the input is a float, string, or None, it returns -1 instead of crashing.
- **Testing Tool:** It uses the pytest framework to verify that both valid numbers and invalid inputs behave as expected.

Task 2 – Test-Driven Development for String Case Converter

Prompt:

Write two Python functions, to_uppercase(text) and to_lowercase(text). For both functions, include safe input handling: if the input is not a string, return -2. Otherwise, return the converted string. Following the functions, write a series of assert statements to test: mixed-case strings, empty strings, integer inputs (returning -2), and None inputs (returning -2). Finally, print a success message if all assertions pass.

Code:

```
# Task 2: String Case Converter using TDD
def to_uppercase(text):
    # Safe input handling
    if type(text) != str:
        return -2

    return text.upper()

def to_lowercase(text):
    # Safe input handling
    if type(text) != str:
        return -2

    return text.lower()

# Test cases for to_uppercase
assert to_uppercase("ai coding") == "AI CODING"
assert to_uppercase("Ai CoDiNg") == "AI CODING"
assert to_uppercase("") == ""
assert to_uppercase(123) == -2
assert to_uppercase(None) == -2

# Test cases for to_lowercase
assert to_lowercase("TEST") == "test"
assert to_lowercase("TeSt") == "test"
assert to_lowercase("") == ""
assert to_lowercase(456) == -2
assert to_lowercase(None) == -2

print("All test cases passed successfully")

# Example of user input simulation
user_val = "Hello World"
print(f"Uppercase: {to_uppercase(user_val)}")
print(f"Lowercase: {to_lowercase(user_val)}")
```

Sample Input/Output:

```
All test cases passed successfully
Uppercase: HELLO WORLD
Lowercase: hello world
```

Code Explanation:

- **Type Checking:** The code uses if type(text) != str to identify non-string inputs like numbers or None.
- **Error Return Value:** Instead of crashing, the function returns -2 for any invalid data types to ensure safe handling.
- **Built-in Methods:** It utilizes Python's .upper() and .lower() methods to handle the actual case conversion.
- **Validation:** The assert statements directly compare the function output against expected results to verify correctness for both normal and edge cases

Task 3 – Test-Driven Development for List Sum Calculator

Prompt:

Write a Python function sum_list(numbers) that returns the total of all numeric elements in a list, skipping strings or other types. Include print statements for test scenarios: sum_list([1, 2, 3]), sum_list([]), sum_list([-1, 5, -4]), and sum_list([2, 'a', 3]). Ensure the final output matches the terminal style shown in the lab instructions.

Code:

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

# Example Test Scenarios (Expected Output) [cite: 75]
print("Example Test Scenarios:\n")

# Testing the function with the required scenarios
print(f"sum_list([1, 2, 3]) -> {sum_list([1, 2, 3])}")
print(f"sum_list([]) -> {sum_list([])}")
print(f"sum_list([-1, 5, -4]) -> {sum_list([-1, 5, -4])}")
print(f"sum_list([2, \"a\", 3]) -> {sum_list([2, 'a', 3])}")

print("\nExpected Output 3")
print("• A robust list-sum function validated using AI-generated test cases.")
```

Sample Input/Output:

```
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.
```

Code Explanation:

- **Initialization**: The total is set to 0 at the start to ensure an empty list returns zero.
- **Filtering**: It uses isinstance to check if an element is a number, which allows it to safely skip strings like "a".
- **Looping**: A for loop iterates through every item, adding only valid numeric values to the total.
- **Output Formatting**: The print statements use f-strings to match the exact "Example Test Scenario" format required by the lab document.

Task 4 – Test Cases for Student Result Class

Prompt:

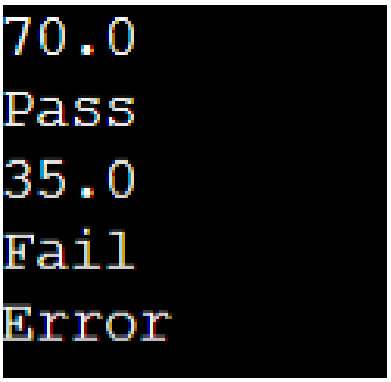
Write a Python class StudentResult with methods add_marks(mark), calculate_average(), and get_result(). Ensure add_marks raises a TypeError for non-numeric values and a ValueError for marks outside the 0-100 range. get_result should return 'Pass' if the average is 40 or higher, otherwise 'Fail'. Include test cases that print the average and result for marks [60, 70, 80] and [30, 35, 40], and use a try-except block to print 'Error' when adding a mark of -10

Code:

```
class StudentResult:
    def __init__(self):
        # Initialize an empty list to store marks
        self.marks = []

    def add_marks(self, mark):
        # Validation: Check if input is a number
        if not isinstance(mark, (int, float)):
            raise TypeError("Mark must be a number")

        # Validation: Check if mark is within valid range (0-100)
        if mark < 0 or mark > 100:
            raise ValueError("Mark must be between 0 and 100")

        self.marks.append(mark)

    def calculate_average(self):
        # Handle case where no marks have been added yet
        if not self.marks:
            return 0
        return sum(self.marks) / len(self.marks)

    def get_result(self):
        # Determine pass/fail status based on average
        average = self.calculate_average()
        return "Pass" if average >= 40 else "Fail"

# --- Test Case 1: Pass scenario ---
sr1 = StudentResult()
sr1.add_marks(60)
sr1.add_marks(70)
sr1.add_marks(80)
print(sr1.calculate_average())  # Expected: 70.0
print(sr1.get_result())         # Expected: Pass

# --- Test Case 2: Fail scenario ---
sr2 = StudentResult()
sr2.add_marks(30)
sr2.add_marks(35)
sr2.add_marks(40)
print(sr2.calculate_average())  # Expected: 35.0
print(sr2.get_result())         # Expected: Fail

# --- Test Case 3: Invalid Mark Handling ---
sr3 = StudentResult()
try:
```

```
    sr3.add_marks(-10)
except ValueError:
    # Per the assignment requirement, print 'Error' for invalid marks
    print("Error")
```

Sample Input/Output:



Code Explanation:

- **Encapsulation**: The class uses an internal list self.marks to keep track of student scores.
- **Data Validation**: The add_marks method uses isinstance for type checking and a range check (0-100) to prevent invalid data from being processed.
- **Logic**: The get_result method uses a "ternary operator" (a one-line if-else) to return "Pass" or "Fail" based on the threshold of 40.
- **Error Handling**: A try-except block is implemented in the test cases to catch specifically the ValueError raised by negative marks and print a simple "Error" message as requested.

Task 5 – Test-Driven Development for Username Validator

Prompt:

Write a Python function is_valid_username(username) that returns False if the length is less than 5, contains spaces, or is not alphanumeric. Otherwise, return True. Then, create a list of test cases including 'user01', 'ai', 'user name', and 'user@123' with their expected boolean results. Use a for-loop to run these tests, printing the result of each in the format 'is_valid_username(input) -> result' and ending with an 'All test cases passed' message

Code:

```
def is_valid_username(username: str) -> bool:
    # Rule 1: Minimum length of 5 characters [cite: 100]
    if len(username) < 5:
        return False

    # Rule 2: No spaces allowed [cite: 101]
    if " " in username:
        return False

    # Rule 3: Only alphanumeric characters
    if not username.isalnum():
        return False

    return True

# Test cases derived from assignment requirements
test_cases = [
    ("user01", True),
    ("ai", False),
    ("user name", False),
    ("user@123", False),
    ("Admin5", True)
]

# Run tests
for username, expected in test_cases:
    result = is_valid_username(username)
    print(f"is_valid_username('{username}') -> {result}")
    assert result == expected

print("\nAll test cases passed")
```

Sample Input/Output:

```
is_valid_username('user01') -> True
is_valid_username('ai') -> False
is_valid_username('user name') -> False
is_valid_username('user@123') -> False
is_valid_username('Admin5') -> True

All test cases passed
```

Code Explanation:

- **Sequential Validation**: The function checks each rule one by one using if statements, returning False immediately if any rule is broken.
- **String Methods**: It uses the built-in .isalnum() method to efficiently check for letters and numbers while excluding symbols.
- **Testing Automation**: A list of tuples stores both the input and the expected outcome, allowing a single loop to validate multiple scenarios.
- **Assertion**: The assert statement ensures that if the code ever fails a test, it will stop and signal an error, guaranteeing the code's reliability.