SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
NAME:MANASA ACHINA ENROLL NO.:2403A53043 BATCH NO.:24BTCAICYB02		**Assignment Type: Lab**	**AcademicYear:**2025-2026
CourseCoordinatorName		Venkataramana Veeramsetty	
Instructor(s)Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
CourseCode	24CS002PC215	**CourseTitle**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week4 - Tuesday	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	
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:** • To introduce students to test-driven development (TDD) using AI code generation tools.	Week4 - Wednesday

- 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#1

PROMPT:
Generate test cases to validate a function is_prime(n) which checks if a number is prime. Include edge cases like 0, 1, 2, negative numbers, large primes, and typical small inputs.

QUESTION:
Use AI to generate test cases for a function is_prime(n) and then implement the function.

```
import math
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    for i in range(5, int(math.sqrt(n)) + 1, 6):
        if n % i == 0 or n % (i + 2) == 0:
            return False
    return True
```

Requirements:

- Only integers > 1 can be prime.
- Check edge cases: 0, 1, 2, negative numbers, and large primes.

Expected Output#1
A working prime checker that passes AI-generated tests using edge coverage.

```
test_cases = [
    (0, False), # Edge case: 0 is not prime
    (1, False), # Edge case: 1 is not prime
    (2, True),  # Edge case: 2 is prime
    (3, True),  # Edge case: 3 is prime
    (4, False), # Composite number
    (5, True),  # Prime number
    (10, False), # Composite number
    (17, True), # Prime number
    (25, False), # Composite number
    (-5, False), # Edge case: Negative numbers are not prime
    (999983, True) # Large prime number
]
for number, expected_output in test_cases:
    actual_output = is_prime(number)
    assert actual_output == expected_output, f"Input: {number}, Expected: {expected_output}, Got: {actual_
    print(f"Test case {number}: Passed")
```

```
Test case 0: Passed
Test case 1: Passed
Test case 2: Passed
Test case 3: Passed
Test case 4: Passed
Test case 5: Passed
Test case 10: Passed
Test case 17: Passed
Test case 25: Passed
Test case -5: Passed
Test case 999983: Passed
```

Task Description#2 (Loops)

PROMPT:

Generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f) including known pairs like 0° C = 32° F, decimals, and invalid inputs such as strings or None to check input validation.

QUESTION:

- Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).

```
def celsius_to_fahrenheit(celsius):
    """Converts Celsius to Fahrenheit."""
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    """Converts Fahrenheit to Celsius."""
    return (fahrenheit - 32) * 5/9
```

Requirements

- Validate known pairs: 0°C = 32°F, 100°C = 212°F.
- Include decimals and invalid inputs like strings or None

Expected Output#2

Dual conversion functions with complete test coverage and safe type handling

```
test_cases = [
    (0, 32),      # 0°C = 32°F
    (100, 212),   # 100°C = 212°F
    (25.5, 77.9), # Decimal test case
    (-10, 14),    # Negative temperature
    ('abc', None), # Invalid input: string
    (None, None)  # Invalid input: None
]
for celsius, fahrenheit in test_cases:
    if isinstance(celsius, (int, float)):
        actual_fahrenheit = celsius_to_fahrenheit(celsius)
        assert abs(actual_fahrenheit - fahrenheit) < 1e-9, f"Input: {celsius}°C, Expected: {fahrenheit}°F, Got: {actual_fa
        print(f"Test case {celsius}°C to {fahrenheit}°F: Passed")
    else:
        try:
            celsius_to_fahrenheit(celsius)
            assert False, f"Input: {celsius}, Expected None for invalid input"
        except (TypeError, ValueError):
            print(f"Test case for invalid input {celsius} in celsius_to_fahrenheit: Passed")
for celsius, fahrenheit in test_cases:
    if isinstance(fahrenheit, (int, float)):
        actual_celsius = fahrenheit_to_celsius(fahrenheit)
        assert abs(actual_celsius - celsius) < 1e-9, f"Input: {fahrenheit}°F, Expected: {celsius}°C, Got: {actual_celsius}
        print(f"Test case {fahrenheit}°F to {celsius}°C: Passed")
    else:
        try:
            fahrenheit_to_celsius(fahrenheit)
            assert False, f"Input: {fahrenheit}, Expected None for invalid input"
        except (TypeError, ValueError):
            print(f"Test case for invalid input {fahrenheit} in fahrenheit_to_celsius: Passed")
```

```
Test case 0°C to 32°F: Passed
Test case 100°C to 212°F: Passed
Test case 25.5°C to 77.9°F: Passed
Test case -10°C to 14°F: Passed
Test case for invalid input abc in celsius_to_fahrenheit: Passed
Test case for invalid input None in celsius_to_fahrenheit: Passed
Test case 32°F to 0°C: Passed
Test case 212°F to 100°C: Passed
Test case 77.9°F to 25.5°C: Passed
Test case 14°F to -10°C: Passed
Test case for invalid input None in fahrenheit_to_celsius: Passed
Test case for invalid input None in fahrenheit_to_celsius: Passed
```

Task Description#3

PROMPT:

Generate test cases for a function count_words(text) that returns the number of words in a sentence. Include normal text, cases with multiple spaces, punctuation, and empty strings.

QUESTION:

Use AI to write test cases for a function count_words(text) that returns the number of words in a
sentence.

```
import string

def count_words(text):
  """Counts the number of words in a sentence."""
  if not isinstance(text, str):
    return 0 # Or raise a TypeError, depending on desired behavior for non-string input

  # Remove punctuation
  text_no_punctuation = text.translate(str.maketrans('', '', string.punctuation))

  # Split the text into words using whitespace as a delimiter
  words = text_no_punctuation.split()

  return len(words)
```

Requirement

Handle normal text, multiple spaces, punctuation, and empty strings.

Expected Output#3

Accurate word count with robust test case validation.

```
test_cases = [
    ("This is a sentence.", 4),  # Normal text
    ("This sentence has  multiple   spaces.", 5), # Multiple spaces
    ("This sentence has punctuation!", 4), # Punctuation
-   ("This sentence has, commas; and. other punctuation!", 6), # Various punctuation
+   ("This sentence has, commas; and. other punctuation!", 7), # Various punctuation - Corrected expected output
    ("", 0), # Empty string
    ("   ", 0), # String with only spaces
    ("Word", 1), # Single word
    (" Word with leading space", 4), # Leading space
    ("Word with trailing space ", 4), # Trailing space
    (123, 0), # Invalid input: number
    (None, 0) # Invalid input: None
]

-# Iterate through the test cases and check the output of count_words
for text, expected_output in test_cases:
    actual_output = count_words(text)
    assert actual_output == expected_output, f"Input: '{text}', Expected: {expected_output}, Got: {actual_output}"
    print(f"Test case '{text}': Passed")
```

```
Test case 'This is a sentence.': Passed
Test case 'This sentence has  multiple   spaces.': Passed
Test case 'This sentence has punctuation!': Passed
Test case 'This sentence has, commas; and. other punctuation!': Passed
Test case '': Passed
Test case '   ': Passed
Test case 'Word': Passed
Test case ' Word with leading space': Passed
Test case 'Word with trailing space ': Passed
Test case '123': Passed
Test case 'None': Passed
```

Task Description#4

PROMPT:

Generate test cases for a BankAccount class with methods deposit(amount), withdraw(amount), and check_balance(). Include cases where negative deposits or withdrawals raise errors, and withdrawing more than the current balance is not allowed

QUESTION:

- Generate test cases for a BankAccount class with:
  Methods:
  deposit(amount)
  withdraw(amount)
  check balance()

```
class BankAccount:
    def __init__(self, initial_balance=0):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative")
        self.balance = initial_balance

    def deposit(self, amount):
        if amount < 0:
            raise ValueError("Deposit amount cannot be negative")
        self.balance += amount

    def withdraw(self, amount):
        if amount < 0:
            raise ValueError("Withdrawal amount cannot be negative")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def check_balance(self):
        return self.balance
```

Requirements:

- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance.

Expected Output#4

- AI-generated test suite with a robust class that handles all test cases.

```
import unittest
class TestBankAccount(unittest.TestCase):
    def test_initial_balance(self):
        account = BankAccount(100)
        self.assertEqual(account.check_balance(), 100)
    def test_initial_balance_default(self):
        account = BankAccount()
        self.assertEqual(account.check_balance(), 0)
    def test_initial_balance_negative_error(self):
        with self.assertRaises(ValueError):
            BankAccount(-100)
    def test_deposit(self):
        account = BankAccount(100)
        account.de
        self.asser
    def test_depos
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.deposit(-50)
    def test_withdraw(self):
        account = BankAccount(100)
        account.withdraw(50)
        self.assertEqual(account.check_balance(), 50)
```

```
def __init__(initial_balance=0)
<class '__main__.BankAccount'>
```

```
    def test_withdraw_negative_error(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw(-50)

    def test_withdraw_insufficient_funds_error(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw(150)

    def test_deposit_and_withdraw(self):
        account = BankAccount(100)
        account.deposit(50)
        account.withdraw(30)
        self.assertEqual(account.check_balance(), 120)
```

Task Description#5
PROMPT:

Generate test cases for is_number_palindrome(num) that check if an integer reads the same backward, including examples like 121 (True), 123 (False), zero, and negative numbers handled gracefully.

QUESTION:

Generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

```
def is_number_palindrome(num):
    """Checks if an integer reads the same backward."""
    if num < 0:
        return False  # Negative numbers are not palindromes
    return str(num) == str(num)[::-1]
```

Examples:

121 → True
123 → False
0, negative numbers → handled gracefully

Expected Output#5

Number-based palindrome checker function validated against test cases.

```
test_cases = [
    (121, True),   # Positive palindrome
    (123, False),  # Positive non-palindrome
    (0, True),     # Edge case: 0 is a palindrome
    (-121, False), # Edge case: negative number
    (1221, True),  # Even number of digits
    (12321, True), # Odd number of digits
    (10, False),   # Number ending in 0
    (1, True),     # Single digit number
    (-1, False)    # Negative single digit
]

for num, expected_output in test_cases:
    actual_output = is_number_palindrome(num)
    assert actual_output == expected_output, f"Input: {num}, Expected: {expected_output}, Got: {actual_output}"
    print(f"Test case {num}: Passed")
```

```
Test case 121: Passed
Test case 123: Passed
Test case 0: Passed
Test case -121: Passed
Test case 1221: Passed
Test case 12321: Passed
Test case 10: Passed
Test case 1: Passed
Test case -1: Passed
```

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

Evaluation Criteria:

Criteria	Max Marks
Task #1	0.5
Task #2	0.5
Task #3	0.5
Task #4	0.5
Task #5	0.5
Total	**2.5 Marks**