SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
ProgramName:B. Tech		**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 - Wednesday	**Time(s)**	
Duration	2 Hours	**Applicableto Batches**	
AssignmentNumber:8.3(Present assignment number)/**24**(Total number of assignments)			
2403A51101 **ANUJA**			

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	Week4 - Wednesday

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#1

Use AI to generate test cases for is_valid_email(email) and then implement the validator function.

Requirements:

- Must contain @ and . characters.
- Must not start or end with special characters.
- Should not allow multiple @.

PROMPT:

Generate test cases (both valid and invalid) for the function is_valid_email(email) based on these rules:

- Must contain @ and . characters.
- Must not start or end with special characters (@, ., _, -).
- Should not allow multiple @.

Implement the function is_valid_email(email) in Python to pass all generated test cases.
Show the output of test execution.
CODE;

```python
def is_valid_email(email):
    """Check if email is valid based on requirements"""

    # Check if email is empty
    if not email:
        return False

    # Check if email contains @ and . characters
    if '@' not in email or '.' not in email:
        return False

    # Check if email has multiple @ symbols
    if email.count('@') > 1:
        return False

    # Check if email starts with special characters
    if email[0] in '@._-':
        return False

    # Check if email ends with special characters
    if email[-1] in '@._-':
        return False

    return True

# Test cases
print("EMAIL VALIDATION TEST RESULTS")
print("=" * 40)

# Valid emails
valid_emails = [
    "user@example.com",
    "test@domain.co.uk",
```

```
    "test@domain.co.uk",
    "name123@company.org"
]

print("\nVALID EMAILS:")
for email in valid_emails:
    result = is_valid_email(email)
    print(f"✓ {email} -> {result}")

# Invalid emails
invalid_emails = [
    "user@@example.com",       # Multiple @
    "@user@example.com",       # Starts with @
    "user@example.com@",       # Ends with @
    "user@example..com",       # Consecutive dots
    "user @example.com",       # Contains space
    "",                        # Empty string
    "user@example",            # Missing dot
    "user.example.com"         # Missing @
]

print("\nINVALID EMAILS:")
for email in invalid_emails:
    result = is_valid_email(email)
    print(f"X {email} -> {result}")

# Test summary
print("\n" + "=" * 40)
print("REQUIREMENTS CHECKED:")
print("✓ Must contain @ and . characters")
print("✓ Must not start or end with special characters")
print("✓ Should not allow multiple @ symbols")
```

Expected Output#1

- Email validation logic passing all test cases

```
EMAIL VALIDATION TEST RESULTS
========================================

VALID EMAILS:
✓ user@example.com -> True
✓ test@domain.co.uk -> True
✓ name123@company.org -> True

INVALID EMAILS:
✗ user@@example.com -> False
✗ @user@example.com -> False
✗ user@example.com@ -> False
✗ user@example..com -> False
✗ user @example.com -> False
✗  -> False
✗ user@example -> False
✗ user.example.com -> False

========================================
REQUIREMENTS CHECKED:
✓ Must contain @ and . characters
✓ Must not start or end with special characters
✓ Must not allow multiple @ symbols
```

Task Description#2 (Loops)

- Ask AI to generate test cases for assign_grade(score) function. Handle boundary and invalid inputs.

  Requirements
- AI should generate test cases for assign_grade(score) where: 90-100: A, 80-89: B, 70-79: C, 60-69: D, <60: F
- Include boundary values and invalid inputs (e.g., -5, 105, "eighty").

PROMPT:

Generate test cases for the function assign_grade(score) with the following grading rules:

- 90–100: A
- 80–89: B
- 70–79: C
- 60–69: D
- <60: F

Include **boundary test cases** (e.g., 59, 60, 69, 70, 79, 80, 89, 90, 100).
Include **invalid inputs** such as negative numbers (e.g., -5), numbers greater than 100 (e.g., 105), and non-numeric inputs (e.g., "eighty", None).
Provide the expected outputs for each test case.
Implement the assign_grade(score) function in Python to handle valid and invalid inputs properly.
Show the test execution results
CODE:

```python
def assign_grade(score):
    """Assign grade based on score with proper error handling"""

    # Check for invalid inputs
    if score is None:
        return "Error: Score cannot be None"

    if not isinstance(score, (int, float)):
        return "Error: Score must be a number"

    if score < 0:
        return "Error: Score cannot be negative"

    if score > 100:
        return "Error: Score cannot exceed 100"

    # Assign grades based on score ranges
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"
```

```python
# Test cases
print("GRADE ASSIGNMENT TEST RESULTS")
print("=" * 50)

# Valid score test cases
print("\nVALID SCORES:")
valid_tests = [
    (100, "A"), (95, "A"), (90, "A"),
    (89, "B"), (85, "B"), (80, "B"),
    (79, "C"), (75, "C"), (70, "C"),
    (69, "D"), (65, "D"), (60, "D"),
    (59, "F"), (50, "F"), (0, "F")
]

for score, expected in valid_tests:
    result = assign_grade(score)
    status = "✓" if result == expected else "✗"
    print(f"{status} Score: {score:3d} -> Grade: {result} (Expected: {expected})")

# Boundary test cases
print("\nBOUNDARY TEST CASES:")
boundary_tests = [
    (59, "F"), (60, "D"), (69, "D"), (70, "C"),
    (79, "C"), (80, "B"), (89, "B"), (90, "A"), (100, "A")
]

for score, expected in boundary_tests:
    result = assign_grade(score)
    status = "✓" if result == expected else "✗"
    print(f"{status} Score: {score:3d} -> Grade: {result} (Expected: {expected})")
```

```python
# Invalid input test cases
print("\nINVALID INPUTS:")
invalid_tests = [
    (-5, "Error: Score cannot be negative"),
    (105, "Error: Score cannot exceed 100"),
    ("eighty", "Error: Score must be a number"),
    ("95", "Error: Score must be a number"),
    (None, "Error: Score cannot be None"),
    ([], "Error: Score must be a number"),
    ({}, "Error: Score must be a number")
]

for score, expected in invalid_tests:
    result = assign_grade(score)
    status = "✓" if result == expected else "✗"
    print(f"{status} Input: {score!r:15} -> {result}")

# Test summary
print("\n" + "=" * 50)
print("GRADING RULES:")
print("90-100: A")
print("80-89:  B")
print("70-79:  C")
print("60-69:  D")
print("<60:    F")
print("\nAll test cases completed!")
```

Expected Output#2

Grade assignment function passing test suite

```
GRADE ASSIGNMENT TEST RESULTS
==================================================

VALID SCORES:
✓ Score: 100 -> Grade: A (Expected: A)
✓ Score:  95 -> Grade: A (Expected: A)
✓ Score:  90 -> Grade: A (Expected: A)
✓ Score:  89 -> Grade: B (Expected: B)
✓ Score:  85 -> Grade: B (Expected: B)
✓ Score:  80 -> Grade: B (Expected: B)
✓ Score:  79 -> Grade: C (Expected: C)
✓ Score:  75 -> Grade: C (Expected: C)
✓ Score:  70 -> Grade: C (Expected: C)
✓ Score:  69 -> Grade: D (Expected: D)
✓ Score:  65 -> Grade: D (Expected: D)
✓ Score:  60 -> Grade: D (Expected: D)
✓ Score:  59 -> Grade: F (Expected: F)
✓ Score:  50 -> Grade: F (Expected: F)
✓ Score:   0 -> Grade: F (Expected: F)
```

```
BOUNDARY TEST CASES:
✓ Score:  59 -> Grade: F (Expected: F)
✓ Score:  60 -> Grade: D (Expected: D)
✓ Score:  69 -> Grade: D (Expected: D)
✓ Score:  70 -> Grade: C (Expected: C)
✓ Score:  79 -> Grade: C (Expected: C)
✓ Score:  80 -> Grade: B (Expected: B)
✓ Score:  89 -> Grade: B (Expected: B)
✓ Score:  90 -> Grade: A (Expected: A)
✓ Score: 100 -> Grade: A (Expected: A)

INVALID INPUTS:
✓ Input: -5              -> Error: Score cannot be negative
✓ Input: 105             -> Error: Score cannot exceed 100
✓ Input: 'eighty'        -> Error: Score must be a number
✓ Input: '95'            -> Error: Score must be a number
✓ Input: None            -> Error: Score cannot be None
✓ Input: []              -> Error: Score must be a number
✓ Input: {}              -> Error: Score must be a number

================================================
GRADING RULES:
90-100: A
80-89:  B
70-79:  C
60-69:  D
<60:    F

All test cases completed!
```

Task Description#3

- Generate test cases using AI for is_sentence_palindrome(sentence). Ignore case, punctuation, and spaces

Requirement

- Ask AI to create test cases for is_sentence_palindrome(sentence)
  (ignores case, spaces, and punctuation).

- Example:
  "A man a plan a canal Panama" → True

PROMT:
Generate test cases for the function is_sentence_palindrome(sentence) that checks if a sentence is a palindrome while **ignoring case, spaces, and punctuation**.
Include:

- Simple palindromes (e.g., "Racecar", "Madam").
- Multi-word palindromes (e.g., "A man a plan a canal Panama").
- Sentences with punctuation and mixed case (e.g., "Was it a car or a cat I saw?").

- Non-palindrome examples.
- Edge cases such as an empty string, single character, and strings with only spaces/punctuation.

Provide the **expected output (True/False)** for each test case.
Implement the is_sentence_palindrome(sentence) function in Python.
Show the test execution results.
CODE:

```
def is_sentence_palindrome(sentence):
    """Check if sentence is palindrome ignoring case, spaces, punctuation"""

    if not sentence:
        return False

    # Remove all non-letters and make lowercase
    clean = ''.join(char.lower() for char in sentence if char.isalpha())

    # Check if palindrome
    return clean == clean[::-1]

# Test cases
print("PALINDROME TEST RESULTS")
print("=" * 40)

# Test palindromes
palindromes = [
    "Racecar",
    "A man a plan a canal Panama",
    "Was it a car or a cat I saw?",
    "Madam",
    "Do geese see God"
]

print("\nPALINDROMES (should be True):")
for text in palindromes:
    result = is_sentence_palindrome(text)
    print(f"✓ '{text}' -> {result}")
```

```
# Test non-palindromes
non_palindromes = [
    "Hello World",
    "Python",
    "This is not a palindrome"
]

print("\nNON-PALINDROMES (should be False):")
for text in non_palindromes:
    result = is_sentence_palindrome(text)
    print(f"✗ '{text}' -> {result}")

# Test edge cases
edge_cases = [
    ("", False),
    ("a", True),
    ("   ", False),
    ("A", True)
]

print("\nEDGE CASES:")
for text, expected in edge_cases:
    result = is_sentence_palindrome(text)
    status = "✓" if result == expected else "✗"
    print(f"{status} {text!r} -> {result}")
```

Expected Output#3

- Function returns True/False for cleaned sentences
- Implement the function to pass AI-generated tests.

```
PALINDROME TEST RESULTS
========================================

PALINDROMES (should be True):
✓ 'Racecar' -> True
✓ 'A man a plan a canal Panama' -> True
✓ 'Was it a car or a cat I saw?' -> True
✓ 'Madam' -> True
✓ 'Do geese see God' -> True

NON-PALINDROMES (should be False):
✗ 'Hello World' -> False
✗ 'Python' -> False
✗ 'This is not a palindrome' -> False

EDGE CASES:
✓ '' -> False
✓ 'a' -> True
✓ '   ' -> False
✓ 'A' -> True
```

Task Description#4

- Let AI fix it Prompt AI to generate test cases for a ShoppingCart class (add_item, remove_item, total_cost).

  Methods:
  Add_item(name,orice)
  Remove_item(name)
  Total_cost()

PROMT:
Generate **test cases** for a ShoppingCart class with the following methods:

- add_item(name, price) → adds an item to the cart.
- remove_item(name) → removes an item from the cart (handle if item doesn't exist).
- total_cost() → returns the total price of all items.

Include **valid and invalid test cases**, such as:

- Adding multiple items.
- Removing items that exist.
- Trying to remove items not in the cart.
- Checking total cost with no items, one item, and multiple items.
- Edge cases like negative price, zero price, or duplicate items.

Provide **expected outputs** for each test case.
Implement the ShoppingCart class in Python.
Run all test cases and display results.
CODE;

```python
class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        if price <= 0:
            return "Invalid price"
        self.items[name] = price
        return f"Added {name}"

    def remove_item(self, name):
        if name in self.items:
            self.items.pop(name)
            return f"Removed {name}"
        return f"{name} not found"

    def total_cost(self):
        return sum(self.items.values())

# Test
print("SHOPPING CART TEST")
print("=" * 30)

cart = ShoppingCart()

print("\n1. Add items:")
print(cart.add_item("Apple", 2.50))
print(cart.add_item("Banana", 1.75))
print(f"Total: ${cart.total_cost()}")
```

```
print("\n2. Remove item:")
print(cart.remove_item("Apple"))
print(f"Total: ${cart.total_cost()}")

print("\n3. Remove non-existent:")
print(cart.remove_item("Grapes"))

print("\n4. Invalid price:")
print(cart.add_item("Invalid", -5))
print(cart.add_item("Free", 0))

print("\n5. Empty cart:")
cart.remove_item("Banana")
print(f"Total: ${cart.total_cost()}")
```

Expected Output#4

- Full class with tested functionalities

```
SHOPPING CART TEST
==============================

1. Add items:
Added Apple
Added Banana
Total: $4.25

2. Remove item:
Removed Apple
Total: $1.75

3. Remove non-existent:
Grapes not found

4. Invalid price:
Invalid price
Invalid price

5. Empty cart:
Total: $0
```

Task Description#5

- Use AI to write test cases for convert_date_format(date_str) to switch from "YYYY-

MM-DD" to "DD-MM-YYYY".
Example: "2023-10-15" → "15-10-2023"

PROMT:
Write test cases for a function convert_date_format(date_str) that changes a date from "YYYY-MM-DD" to "DD-MM-YYYY".

- **Include normal dates, boundary dates, invalid formats, empty string, and wrong values (like "2023-13-01").**
- **Show the expected outputs.**
- **Then write the Python function and run the test cases.**

CODE:

```
from datetime import datetime

def convert_date_format(date_str):
    """Convert date from YYYY-MM-DD to DD-MM-YYYY"""

    if not date_str:
        return "Error: Empty date"

    try:
        # Parse the date
        date_obj = datetime.strptime(date_str, "%Y-%m-%d")
        # Convert to DD-MM-YYYY format
        return date_obj.strftime("%d-%m-%Y")
    except ValueError:
        return "Error: Invalid date format"

# Test cases
print("DATE FORMAT CONVERTER TEST")
print("=" * 40)

# Normal dates
print("\n1. NORMAL DATES:")
normal_dates = [
    ("2023-01-15", "15-01-2023"),
    ("2024-12-31", "31-12-2024"),
    ("2020-02-29", "29-02-2020")  # Leap year
]
```

```python
for date, expected in normal_dates:
    result = convert_date_format(date)
    status = "✓" if result == expected else "✗"
    print(f"{status} {date} -> {result}")

# Boundary dates
print("\n2. BOUNDARY DATES:")
boundary_dates = [
    ("2023-01-01", "01-01-2023"),  # First day of year
    ("2023-12-31", "31-12-2023"),  # Last day of year
    ("2023-02-28", "28-02-2023"),  # Last day of February
    ("2024-02-29", "29-02-2024")   # Leap year February
]

for date, expected in boundary_dates:
    result = convert_date_format(date)
    status = "✓" if result == expected else "✗"
    print(f"{status} {date} -> {result}")

# Invalid formats
print("\n3. INVALID FORMATS:")
invalid_dates = [
    ("2023-13-01", "Error: Invalid date format"),  # Invalid month
    ("2023-02-30", "Error: Invalid date format"),  # Invalid day
    ("2023-04-31", "Error: Invalid date format"),  # Invalid day
    ("2023-00-15", "Error: Invalid date format"),  # Zero month
    ("2023-01-00", "Error: Invalid date format")   # Zero day
]

for date, expected in invalid_dates:
    result = convert_date_format(date)
```

```
for date, expected in invalid_dates:
    result = convert_date_format(date)
    status = "✓" if result == expected else "✗"
    print(f"{status} {date} -> {result}")

# Edge cases
print("\n4. EDGE CASES:")
edge_cases = [
    ("", "Error: Empty date"),
    ("2023-1-15", "Error: Invalid date format"),   # Missing leading zeros
    ("2023-01-5", "Error: Invalid date format"),   # Missing leading zeros
    ("15-01-2023", "Error: Invalid date format"),  # Wrong format
    ("2023/01/15", "Error: Invalid date format"),  # Wrong separator
    ("abc-def-ghi", "Error: Invalid date format")   # Non-numeric
]

for date, expected in edge_cases:
    result = convert_date_format(date)
    status = "✓" if result == expected else "✗"
    print(f"{status} {date!r} -> {result}")

# Test summary
print("\n" + "=" * 40)
print("FUNCTION FEATURES:")
print("✓ Converts YYYY-MM-DD to DD-MM-YYYY")
print("✓ Handles leap years")
print("✓ Validates date existence")
print("✓ Error handling for invalid dates")
print("✓ Handles edge cases")
```

Expected Output#5

- Function converts input format correctly for all test cases

```
DATE FORMAT CONVERTER TEST
========================================

1. NORMAL DATES:
✓ 2023-01-15 -> 15-01-2023
✓ 2024-12-31 -> 31-12-2024
✓ 2020-02-29 -> 29-02-2020

2. BOUNDARY DATES:
✓ 2023-01-01 -> 01-01-2023
✓ 2023-12-31 -> 31-12-2023
✓ 2023-02-28 -> 28-02-2023
✓ 2024-02-29 -> 29-02-2024

3. INVALID FORMATS:
✓ 2023-13-01 -> Error: Invalid date format
✓ 2023-02-30 -> Error: Invalid date format
✓ 2023-04-31 -> Error: Invalid date format
✓ 2023-00-15 -> Error: Invalid date format
✓ 2023-01-00 -> Error: Invalid date format

4. EDGE CASES:
✓ '' -> Error: Empty date
✓ '2023-1-15' -> Error: Invalid date format
✓ '2023-01-5' -> Error: Invalid date format
✓ '15-01-2023' -> Error: Invalid date format
✓ '2023/01/15' -> Error: Invalid date format
```

```
✓ '15-01-2023' -> Error: Invalid date format
✓ '2023/01/15' -> Error: Invalid date format
✓ 'abc-def-ghi' -> Error: Invalid date format

========================================
FUNCTION FEATURES:
✓ Converts YYYY-MM-DD to DD-MM-YYYY
✓ Handles leap years
✓ Validates date existence
✓ Error handling for invalid dates
✓ Handles edge 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

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**