

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	
Course Coordinator Name		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)	
Course Code	24CS002PC215	Course Title	AI Assisted Coding
Year/Sem	II/I	Regulation	R24
Date and Day of Assignment	Week4 - Monday	Time(s)	
Duration	2 Hours	Applicable to Batches	
AssignmentNumber: 8.1(Present assignment number)/ 24 (Total number of assignments)			
Q.No.	Question		Expected Time to complete
1	<p>Lab 8: Test-Driven Development with AI – Generating and Working with Test Cases</p> <p>Lab Objectives:</p> <ul style="list-style-type: none"> • To introduce students to test-driven development (TDD) using AI code generation tools. 		Week4 - Monday

	<ul style="list-style-type: none"> • 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. <p>Lab Outcomes (LOs):</p> <p>After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> • 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 <p>Task Description #1 (Password Strength Validator – Apply AI in Security Context)</p> <ul style="list-style-type: none"> • Task: Apply AI to generate at least 3 assert test cases for <code>is_strong_password(password)</code> and implement the validator function. • Requirements: <ul style="list-style-type: none"> ◦ Password must have at least 8 characters. ◦ Must include uppercase, lowercase, digit, and special character. ◦ Must not contain spaces. <p>Example Assert Test Cases:</p> <pre>assert is_strong_password("Abcd@123") == True assert is_strong_password("abcd123") == False assert is_strong_password("ABCD@1234") == True</pre> <p>Expected Output #1:</p> <ul style="list-style-type: none"> • Password validation logic passing all AI-generated test cases. 	
--	---	--

The screenshot shows two instances of Google Colab. In the top instance, a Python script named `AIAC 8.1.ipynb` contains a function `is_strong_password` and several print statements for testing. The bottom instance shows the results of running the code, displaying various password strings and their classification as strong or not.

```

# AIAC 8.1.ipynb
import re

def is_strong_password(password):
    """
    Validates a password based on given security rules.

    Args:
        password: The password string to validate.

    Returns:
        True if the password is strong, False otherwise.
    """
    if len(password) < 8:
        return False
    if not any(c.isupper() for c in password):
        return False
    if not any(c.islower() for c in password):
        return False
    if not any(c.isdigit() for c in password):
        return False
    if not re.search(r'[@#$%^&(),.:;{}<>]', password):
        return False
    if ' ' in password:
        return False
    return True

# AI-generated test cases
print("Password123!" is strong: (is_strong_password('Password123!'))")
print("short1!" is strong: (is_strong_password('short1!'))")
print("NoUppercase1!" is strong: (is_strong_password('NoUppercase1!'))")
print("NoLowercase1!" is strong: (is_strong_password('NoLowercase1!')))
print("NoDigits1!" is strong: (is_strong_password('NoDigits1!'))")
print("NoSpecialChars123" is strong: (is_strong_password('NoSpecialChars123')))
print("Password 123!" is strong: (is_strong_password('Password 123!')))

# What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

```


OUTPUT:

```

# AIAC 8.1.ipynb
if not any(c.isupper() for c in password):
    return False
if not any(c.islower() for c in password):
    return False
if " " in password:
    return False
return True

# AI-generated test cases
print("Password123!" is strong: (is_strong_password('Password123!')))
print("short1!" is strong: (is_strong_password('short1!')))
print("NoUppercase1!" is strong: (is_strong_password('NoUppercase1!')))
print("NoLowercase1!" is strong: (is_strong_password('NoLowercase1!')))
print("NoDigits1!" is strong: (is_strong_password('NoDigits1!')))
print("NoSpecialChars123" is strong: (is_strong_password('NoSpecialChars123')))
print("Password 123!" is strong: (is_strong_password('Password 123!')))

# What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

```

Task Description #2 (Number Classification with Loops – Apply AI for Edge Case Handling)

- Task: Use AI to generate at least 3 assert test cases for a `classify_number(n)` function. Implement using loops.
- Requirements:
 - Classify numbers as Positive, Negative, or Zero.
 - Handle invalid inputs like strings and None.
 - Include boundary conditions (-1, 0, 1).

Example Assert Test Cases:

```

assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"

```

Expected Output #2:

- Classification logic passing all assert tests.

```

def classify_number(n):
    """
    Classifies a number as Positive, Negative, or Zero.

    Args:
        n: The input to classify.

    Returns:
        A string indicating the classification ('Positive', 'Negative', 'Zero', 'Invalid Input').
    """
    if not isinstance(n, (int, float)):
        return "Invalid Input"

    # Classify the number
    if n > 0:
        return "Positive"
    elif n < 0:
        return "Negative"
    else:
        return "Zero"

# AI-generated test cases
assert classify_number(10) == "Positive"
assert classify_number(-5) == "Negative"
assert classify_number(0) == "Zero"
assert classify_number(1) == "Positive" # Boundary case
assert classify_number(-1) == "Negative" # Boundary case
assert classify_number('abc') == "Invalid Input" # Invalid input (string)
assert classify_number(None) == "Invalid Input" # Invalid input (None)
assert classify_number(3.14) == "Positive" # Float input
assert classify_number(-2.71) == "Negative" # Float input

```

OUTPUT:

```

[8]: print("10 is: ", classify_number(10))
print("-5 is: ", classify_number(-5))
print("0 is: ", classify_number(0))
print("1 is: ", classify_number(1))
print("-1 is: ", classify_number(-1))
print("abc is: ", classify_number('abc'))
print("None is: ", classify_number(None))
print("3.14 is: ", classify_number(3.14))
print("-2.71 is: ", classify_number(-2.71))

10 is: Positive
-5 is: Negative
0 is: Zero
1 is: Positive
-1 is: Negative
'abc' is: Invalid Input
None is: Invalid Input
3.14 is: Positive
-2.71 is: Negative

```

Task Description #3 (Anagram Checker – Apply AI for String Analysis)

- Task: Use AI to generate at least 3 assert test cases for `is_anagram(str1, str2)` and implement the function.
- Requirements:
 - Ignore case, spaces, and punctuation.
 - Handle edge cases (empty strings, identical words).

Example Assert Test Cases:

```

assert is_anagram("listen", "silent") == True
assert is_anagram("hello", "world") == False
assert is_anagram("Dormitory", "Dirty Room") == True

```

Expected Output #3:

- Function correctly identifying anagrams and passing all AI-generated tests.

```

import re

def is_anagram(str1, str2):
    """
    Run cell (Ctrl+Enter) to check if the strings are anagrams, ignoring case, spaces, and punctuation.
    cell executed since last change
    executed by Ashar Ganji 10:18 AM (0 minutes ago)
    Returns:
        True if the strings are anagrams, False otherwise.
    """
    # Clean the strings by removing non-alphanumeric characters and converting to lowercase
    cleaned_str1 = re.sub('[^a-zA-Z0-9]', '', str1.lower())
    cleaned_str2 = re.sub('[^a-zA-Z0-9]', '', str2.lower())

    # Check if the sorted strings are equal
    return sorted(cleaned_str1) == sorted(cleaned_str2)

# AI-generated test cases
assert is_anagram("listen", "silent") == True
assert is_anagram("Debit Card", "Bad Credit") == True # Ignore case and spaces
assert is_anagram("a gentleman", "elegant man") == True # Ignore case and spaces
assert is_anagram("The eyes", "They see") == True # Ignore case and spaces
assert is_anagram("Hello", "World") == True # Different strings
assert is_anagram("Hello", "World") == False # Different strings
assert is_anagram("dormitory", "dirty room") == True # Ignore case and spaces
assert is_anagram("identical", "identical") == True # Identical words
assert is_anagram("restful", "fluster") == True # Anagram with different length initially due to spaces
assert is_anagram(" #@%", "%#@!") == True # Only punctuation (should be empty after cleaning)
assert is_anagram("go", "goo") == False # Different length after cleaning

```

OUTPUT:

```

assert is_anagram("a gentleman", "elegant man") == True # Ignore case and spaces
assert is_anagram("") == True # Empty string
assert is_anagram("Hello", "world") == False # Different strings
Run cell (Ctrl+Enter)
cell executed since last change
executed by Ashar Ganji 10:18 AM (0 minutes ago)
Returns:
True
[1]: print('listen' and 'silent' are anagrams: (is_anagram('listen', 'silent')))
print('Debit Card' and 'Bad Credit' are anagrams: (is_anagram('debit card', 'bad credit')))
print('a gentleman' and 'elegant man' are anagrams: (is_anagram('a gentleman', 'elegant man')))
print('The eyes' and 'They see' are anagrams: (is_anagram('the eyes', 'they see')))
print('Empty string' and '' are anagrams: (is_anagram('', '')))
print('Hello' and 'world' are anagrams: (is_anagram('Hello', 'world')))
print('dormitory' and 'dirty room' are anagrams: (is_anagram('dormitory', 'dirty room')))
print('identical' and 'identical' are anagrams: (is_anagram('identical', 'identical')))
print('restful' and 'fluster' are anagrams: (is_anagram('restful', 'fluster')))
print('#@%' and '%#@!' are anagrams: (is_anagram('#@%', '%#@!')))
print('go' and 'goo' are anagrams: (is_anagram('go', 'goo')))

'listen' and 'silent' are anagrams: True
'Debit Card' and 'Bad Credit' are anagrams: True
'a gentleman' and 'elegant man' are anagrams: True
'The eyes' and 'They see' are anagrams: True
'Empty string' and '' are anagrams: True
'Hello' and 'world' are anagrams: False
'dormitory' and 'dirty room' are anagrams: True
'identical' and 'identical' are anagrams: True
'restful' and 'fluster' are anagrams: True
'#@%' and '%#@!' are anagrams: True
'go' and 'goo' are anagrams: False

```

Task Description #4 (Inventory Class – Apply AI to Simulate Real-World Inventory System)

- Task: Ask AI to generate at least 3 assert-based tests for an Inventory class with stock management.
- Methods:
 - add_item(name, quantity)
 - remove_item(name, quantity)
 - get_stock(name)

Example Assert Test Cases:

```

inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3

```

Expected Output #4:

- Fully functional class passing all assertions.

```

class Inventory:
    """Manages inventory stock with methods for adding, removing, and retrieving items.
    """
    def __init__(self):
        self.items = {}

    def add_item(self, name, quantity):
        """Adds the given quantity of an item."""
        if name in self.items:
            self.items[name] += quantity
        else:
            self.items[name] = quantity

    def remove_item(self, name, quantity):
        """Removes the given quantity (if available)."""
        if name in self.items:
            if self.items[name] - quantity >= 0:
                self.items[name] -= quantity
            elif name in self.items and self.items[name] < quantity:
                self.items[name] = 0
        else:
            print(f"Error: Item {name} does not exist in inventory.")

    def get_stock(self, name):
        """Returns the current stock of an item (default 0 if item doesn't exist)."""
        return self.items.get(name, 0)

# All generated test cases
inventory = Inventory()

print("--- Inventory Class Outputs ---")

# Test Case 1: Adding items and checking stock
print("Adding 10 apples and 5 bananas.")
inventory.add_item("apple", 10)
inventory.add_item("banana", 5)
print(f"Stock of apple: {inventory.get_stock('apple')}")
print(f"Stock of banana: {inventory.get_stock('banana')}")
print(f"Stock of orange (non-existent): {inventory.get_stock('orange')}")

assert inventory.get_stock("apple") == 10, "Test Case 1 Failed: Adding Items"
assert inventory.get_stock("banana") == 5, "Test Case 1 Failed: Adding Items"
assert inventory.get_stock("orange") == 0, "Test Case 1 Failed: Non-existent Item"
print("-" * 20)

# Test Case 2: Removing items
print("Removing 5 apples and 10 bananas.")
inventory.remove_item("apple", 5)
inventory.remove_item("banana", 10)
print(f"Stock of apple after removal: {inventory.get_stock('apple')}")
print(f"Stock of banana after removal: {inventory.get_stock('banana')}")
print(f"Stock of orange after removal: {inventory.get_stock('orange')}")

assert inventory.get_stock("apple") == 5, "Test Case 2 Failed: Removing Items"
assert inventory.get_stock("banana") == 0, "Test Case 2 Failed: Removing More Than Available"
assert inventory.get_stock("orange") == 0, "Test Case 2 Failed: Removing Non-existent Item"
print("-" * 20)

# Test Case 3: Adding and then removing, checking final stock
print("Adding 20 grapes, removing 20, then adding 5.")
inventory.add_item("grape", 20)
inventory.remove_item("grape", 20)
print(f"Stock of grape after adding 20: {inventory.get_stock('grape')}")
assert inventory.get_stock("grape") == 0, "Test Case 3 Failed: Adding and then removing all"
inventory.add_item("grape", 5)
print(f"Stock of grape after adding 5: {inventory.get_stock('grape')}")
assert inventory.get_stock("grape") == 5, "Test Case 3 Failed: Add after removing all and adding 5"
print("-" * 20)

--- Inventory Class Outputs ---
Adding 10 apples and 5 bananas.
Stock of apple: 10
Stock of banana: 5
Stock of orange (non-existent): 0
-----
Removing 5 apples and 10 bananas.
Stock of apple after removal: 5
Stock of banana after removal: 0
Stock of orange after removal: 0
-----
Adding 20 grapes, removing 20, then adding 5.
Stock of grape after adding 20: 0
Stock of grape after adding 5: 5

```

OUTPUT:

```

--- Inventory Class Outputs ---
Adding 10 apples and 5 bananas.
Stock of apple: 10
Stock of banana: 5
Stock of orange (non-existent): 0
-----
Removing 5 apples and 10 bananas.
Stock of apple after removal: 5
Stock of banana after removal: 0
Stock of orange after removal: 0
-----
Adding 20 grapes, removing 20, then adding 5.
Stock of grape after adding 20: 0
Stock of grape after adding 5: 5

```

Task Description #5 (Date Validation & Formatting – Apply AI for Data Validation)

- Task: Use AI to generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.
- Requirements:
 - Validate "MM/DD/YYYY" format.
 - Handle invalid dates.
 - Convert valid dates to "YYYY-MM-DD".

Example Assert Test Cases:

```
assert validate_and_format_date("10/15/2023") == "2023-10-15"
assert validate_and_format_date("02/30/2023") == "Invalid Date"
assert validate_and_format_date("01/01/2024") == "2024-01-01"
```

Expected Output #5:

- Function passes all AI-generated assertions and handles edge cases.

OUTPUT:

The screenshot shows a Jupyter Notebook cell with the following code:

```
import datetime

def validate_and_format_date(date_str):
    """
    Validates a date string in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".
    Args:
        date_str: The date string to validate and format.
    Returns:
        The formatted date string in "YYYY-MM-DD" format if valid, otherwise None.
    """
    try:
        # Attempt to parse the date string in MM/DD/YYYY format
        date_obj = datetime.datetime.strptime(date_str, "%m/%d/%Y")
        # If parsing is successful, format it to YYYY-MM-DD
        return date_obj.strftime("%Y-%m-%d")
    except ValueError:
        # If parsing fails, the date is invalid
        return None

# AI generated test cases and printing outputs
print("12/25/2023" formatted: validate_and_format_date('12/25/2023'))
print("02/28/2023" formatted: validate_and_format_date('02/28/2023'))
print("13/01/2023" formatted: validate_and_format_date('13/01/2023'))
print("01/01/2023" formatted: validate_and_format_date('01/01/2023'))
print("not a date" formatted: validate_and_format_date('not a date'))
```

The output of the code is displayed below the cell:

```
12/25/2023 formatted: 2023-12-25
02/28/2023 formatted: None
13/01/2023 formatted: None
01/01/2023 formatted: 2023-01-01
not a date formatted: None
```

A sidebar titled "Gemini X" provides AI-generated feedback and suggestions for improvement.

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

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