ASSIGNMENT-8.1

Task Description #1 (Password Strength Validator – Apply AI in Security Context)

• Task: Apply AI to generate at least 3 assert test cases for

is_strong_password(password) and implement the validator function.

• Requirements:

o Password must have at least 8 characters.

o Must include uppercase, lowercase, digit, and specialcharacter.

o Must not contain spaces.

Example Assert Test Cases:

assert is_strong_password("Abcd@123") == True

assert is_strong_password("abcd123") == False

assert is_strong_password("ABCD@1234") == True

Expected Output #1:

• Password validation logic passing all AI-generated test cases.

```
import re
def is_strong_password(password):
    if len(password) < 8:
        return False

    if " " in password:
        return False
    has_upper = re.search(r"[A-Z]", password)
    has_lower = re.search(r"[a-z]", password)
    has_digit = re.search(r"[0-9]", password)
    has_special = re.search(r"[!@#$%^&*(),.?\":{}|<>_\-+=/\\[\]]", password)

    if has_upper and has_lower and has_digit and has_special:
        return True

    return False

password = input("Enter your password: ")

if is_strong_password(password):
    print("Strong Password")
else:
    print(" Weak Password")
```

#OUTPUT:

Enter your password: CSE136@22

Strong Password

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:

o Classify numbers as Positive, Negative, or Zero.

o Handle invalid inputs like strings and None.

o 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:

```
def classify_number(n):
    if not isinstance(n, (int, float)):
        return "Invalid Input"
    for _ in range(1):
        if n > 0:
            return "Positive"
        elif n < 0:
            return "Negative"
        else:
            return "Zero"
# User Input
user_input = input("Enter a number: ")
try:
    number = float(user_input)
    result = classify_number(number)
except:
    result = "Invalid Input"
print("Classification:", result)
```

#OUTPUT

Enter a number: 20

Classification: Positive

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:

o Ignore case, spaces, and punctuation.

o 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.

```
def is_anagram(str1, str2):
    if not isinstance(str1, str) or not isinstance(str2, str):
        return False

    cleaned1 = ""
    cleaned2 = ""
    for ch in str1.lower():
        if ch.isalnum():
            cleaned1 += ch

    for ch in str2.lower():
        if ch.isalnum():
            cleaned2 += ch

    return sorted(cleaned1) == sorted(cleaned2)
string1 = input("Enter first string: ")
string2 = input("Enter second string: ")

if is_anagram(string1, string2):
    print("The strings are Anagrams.")
else:
    print("The strings are NOT Anagrams.")
```

#OUTPUT

Enter first string: sai

Enter second string: isa

The strings are Anagrams.

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:

o add_item(name, quantity)

o remove_item(name, quantity)

o 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:

```
class Inventory:
    def __init__(self):
        self.stock = {}

    def add_item(self, item, quantity):
        if item in self.stock:
            self.stock[item] += quantity
        else:
            self.stock[item] = quantity

    def remove_item(self, item, quantity):
        if item in self.stock and self.stock[item] >= quantity:
            self.stock[item] -= quantity
            return True
        return False

    def check_stock(self, item):
        return self.stock.get(item, 0)
# take user inputs to test the Inventory class methods
inventory = Inventory()
# Test adding items
item_to_add = input("Enter the item to add: ")
quantity_to_add = int(input("Enter the quantity to add: "))
```

```
inventory.add_item(item_to_add, quantity_to_add)
# Test checking stock levels
item_to_check = input("Enter the item to check stock levels: ")
stock_level = inventory.check_stock(item_to_check)
print(f"Stock level for {item_to_check}: {stock_level}")
# Test removing items
item_to_remove = input("Enter the item to remove: ")
quantity_to_remove = int(input("Enter the quantity to remove: "))
if inventory.remove_item(item_to_remove, quantity_to_remove):
    print(f"Removed {quantity_to_remove} of {item_to_remove}.")
else:
    print(f"Failed to remove {quantity_to_remove} of {item_to_remove}. Not
enough stock or item does not exist.")
# Assert-based tests
assert inventory.check_stock(item_to_add) == quantity_to_add, "Test failed:
Item not added correctly."
assert inventory.remove_item(item_to_add, quantity_to_add) == True, "Test
failed: Item not removed correctly."
assert inventory.check_stock(item_to_add) == 0, "Test failed: Stock level not
updated after removal."
assert inventory.remove_item(item_to_add, 1) == False, "Test failed: Should
not remove item that is out of stock."
print("All tests passed successfully.")
```

#OUTPUT

Enter the item to add: apple

Enter the quantity to add: 2

Enter the item to check stock levels: 1

Stock level for 1: 0

Enter the item to remove: mango

Enter the quantity to remove: 2

Failed to remove 2 of mango. Not enough stock or item does not exist.

All tests passed successfully.

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:

o Validate "MM/DD/YYYY" format.

o Handle invalid dates.

o 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

```
import re
def validate_date(date_str):
    # Check if the format is correct
    if not re.match(r'^\d{2}/\d{2}/\d{4}$', date_str):
        return "Invalid date format. Please use DD/MM/YYYY."

    day, month, year = map(int, date_str.split('/'))

    # Check for valid month
    if month < 1 or month > 12:
        return "Invalid month. Month must be between 1 and 12."

    # Check for valid day based on the month
    if month in [1, 3, 5, 7, 8, 10, 12]:  # Months with 31 days
        if day < 1 or day > 31:
            return "Invalid day. Day must be between 1 and 31 for the given
month."
    elif month in [4, 6, 9, 11]:  # Months with 30 days
        if day < 1 or day > 30:
            return "Invalid day. Day must be between 1 and 30 for the given
month."
    else:  # February
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):  # Leap
year
            if day < 1 or day > 29:
                return "Invalid day. Day must be between 1 and 29 for February
in a leap year."
        else:
            if day < 1 or day > 28:
                return "Invalid day. Day must be between 1 and 28 for February
in a non-leap year."

    return "Valid date."
# Example usage
```

```
date_input = input("Enter a date (DD/MM/YYYY): ")
result = validate_date(date_input)
print(result)
#ASK UNTIL THE USER ENTERS A VALID DATE
while True:
    date_input = input("Enter a date (DD/MM/YYYY): ")
    result = validate_date(date_input)
    print(result)
    if result == "Valid date.":
        break
```

#OUTPUT

Enter a date (DD/MM/YYYY): 12/03/2025

Valid date.

Enter a date (DD/MM/YYYY): 12/00/20

Invalid date format. Please use DD/MM/YYYY.

Enter a date (DD/MM/YYYY): 10/02/2024

Valid date.