LAB 8.1

HallTkt: 2403A51313

Batch: 13

Task Description #1 (Password StrengthValidator – Apply AI inSecurity Context)

AI GENERATED CODE:

```python
import re

def is_strong_password(password: str) -> bool:
    if len(password) < 8:
        return False
    if " " in password:
        return False

    has_upper = re.search(r"[A-Z]", password) is not None
    has_lower = re.search(r"[a-z]", password) is not None
    has_digit = re.search(r"\d", password) is not None
    has_special = re.search(r"[^A-Za-z0-9]", password) is not None

    return has_upper and has_lower and has_digit and has_special


# ✅ Corrected Test cases
assert is_strong_password("Abcd@123") == True    # Valid
assert is_strong_password("abcd123") == False    # Missing uppercase + special
assert is_strong_password("ABCD@1234") == False  # Missing lowercase
assert is_strong_password("Abcdefgh") == False   # Missing digit + special
assert is_strong_password("Abc 123@") == False   # Contains space
assert is_strong_password("aB1$5678") == True    # Valid

print("🎉 AI Test Cases passed successfully!")
```

```
🎉 AI Test Cases passed successfully!
```

Observation:

In this experiment, the is_strong_password(password) function was implemented to validate passwords based on specific security rules: a minimum of 8 characters, inclusion of uppercase and lowercase letters, at least one digit, one special character, and no spaces. Multiple test cases were used to verify its functionality, including "Abcd@123" and "aB1$5678" which passed as valid strong passwords, and "abcd123", "ABCD@1234", "Abcdefgh", and "Abc 123@" which failed due to missing required character types or containing spaces. All assertion test cases ran successfully, confirming that the function correctly distinguishes between strong and weak passwords. This demonstrates that the password validator effectively enforces the desired security rules and ensures robust validation through automated testing.

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

AI GENERATED CODE:

```python
def classify_number(n):
    # Handle invalid inputs
    if not isinstance(n, (int, float)):
        return "Invalid input"

    # List of categories to check using loop
    categories = [("Positive", lambda x: x > 0),
                  ("Negative", lambda x: x < 0),
                  ("Zero", lambda x: x == 0)]

    for label, condition in categories:
        if condition(n):
            return label

# ✅ AI-generated test cases
assert classify_number(10) == "Positive"      # Positive number
assert classify_number(-5) == "Negative"      # Negative number
assert classify_number(0) == "Zero"           # Zero
assert classify_number(1) == "Positive"       # Boundary positive
assert classify_number(-1) == "Negative"      # Boundary negative
assert classify_number("abc") == "Invalid input" # Invalid string
assert classify_number(None) == "Invalid input"  # Invalid None

print("🎉 All assertion test cases passed successfully!")
```

```
🎉 All assertion test cases passed successfully!
```

Observation:

In this experiment, the classify_number(n) function was implemented to categorize numbers as Positive, Negative, or Zero, while also handling invalid inputs such as strings and None. The function uses a loop to iterate through defined conditions for classification. Several test cases, including boundary values -1, 0, and 1, were used to validate the logic. Numbers like 10 and 1 were correctly classified as Positive, -5 and -1 as Negative, and 0 as Zero. Invalid inputs such as "abc" and None returned "Invalid input", demonstrating proper error handling. All assertion test cases passed successfully, confirming that the function reliably classifies numbers across typical, boundary, and invalid input scenarios.

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

AI GENERATED CODE:

```python
import re

def is_anagram(str1, str2):
    # Helper: clean strings by removing non-alphabetic characters and converting to lowercase
    def clean(s):
        return re.sub(r'[^a-zA-Z0-9]', '', s).lower()

    s1 = clean(str1)
    s2 = clean(str2)

    # Edge case: both empty strings
    if s1 == "" and s2 == "":
        return True

    # Compare sorted characters
    return sorted(s1) == sorted(s2)

# ✅ AI-generated test cases
assert is_anagram("listen", "silent") == True              # Basic anagram
assert is_anagram("hello", "world") == False              # Not an anagram
assert is_anagram("Dormitory", "Dirty Room") == True      # Ignore case, spaces
assert is_anagram("Astronomer", "Moon starer") == True    # Ignore spaces and case
assert is_anagram("", "") == True                         # Edge case: both empty
assert is_anagram("a", "a") == True                       # Edge case: identical single char
assert is_anagram("a!b@c", "c b a") == True              # Ignore punctuation

print("🎉 All assertion test cases passed successfully!")
```

```
🎉 All assertion test cases passed successfully!
```

Observation:

In this experiment, the is_anagram(str1, str2) function was implemented to determine whether two strings are anagrams of each other. The function first cleans the input strings by removing spaces, punctuation, and converting all characters to lowercase, ensuring that comparisons are case-insensitive and ignore non-alphanumeric characters. Several test cases were used to validate the logic, including "listen" and "silent" as a basic anagram, "Dormitory" and "Dirty Room" to verify handling of spaces and capitalization, and empty strings as an edge case. The function correctly identified all anagrams and non-anagrams, demonstrating robust handling of typical, edge, and punctuation-inclusive inputs. All assertion test cases passed successfully, confirming the reliability of the implemented logic.

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

AI GENERATED CODE:

```python
class Inventory:
    def __init__(self):
        self.stock = {}  # Dictionary to keep track of items and quantities
    def add_item(self, name, quantity):
        if quantity <= 0:
            return  # Ignore invalid quantities
        if name in self.stock:
            self.stock[name] += quantity
        else:
            self.stock[name] = quantity
    def remove_item(self, name, quantity):
        if name not in self.stock or quantity <= 0:
            return  # Nothing to remove
        self.stock[name] -= quantity
        if self.stock[name] <= 0:
            del self.stock[name]  # Remove item if quantity drops to 0 or below
    def get_stock(self, name):
        return self.stock.get(name, 0)  # Return 0 if item doesn't exist
# ✅ AI-generated assert test cases
inv = Inventory()
# Add and check stock
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10
# Remove some quantity
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5
# Add new item
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3
# Edge case: remove more than available
inv.remove_item("Book", 5)
assert inv.get_stock("Book") == 0
# Edge case: adding zero or negative quantity (should not change stock)
inv.add_item("Pen", -2)
assert inv.get_stock("Pen") == 5
# Edge case: removing zero or negative quantity (should not change stock)
inv.remove_item("Pen", 0)
assert inv.get_stock("Pen") == 5
print("🎉 All Inventory class assertion tests passed successfully!")
```

```
🎉 All Inventory class assertion tests passed successfully!
```

Observation:

In this experiment, an Inventory class was implemented to manage stock levels of items using methods add_item, remove_item, and get_stock. The class uses a dictionary to store item names and their quantities. Multiple test cases were used to verify its functionality, including adding new items, removing partial quantities, and checking stock levels. Edge cases such as removing more than the available quantity, adding or removing zero or negative quantities, and querying non-existent items were also tested. The class correctly updated and tracked stock in all scenarios, and all assertion-based test cases passed successfully, demonstrating reliable inventory management and robust handling of typical and boundary conditions.

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

AI GENERATED CODE:

```
from datetime import datetime

def validate_and_format_date(date_str):
    try:
        # Parse the date in MM/DD/YYYY format
        dt = datetime.strptime(date_str, "%m/%d/%Y")
        # Return in YYYY-MM-DD format
        return dt.strftime("%Y-%m-%d")
    except ValueError:
        return "Invalid Date"

# ✅ AI-generated assert test cases
assert validate_and_format_date("10/15/2023") == "2023-10-15"   # Valid date
assert validate_and_format_date("02/30/2023") == "Invalid Date" # Invalid date
assert validate_and_format_date("01/01/2024") == "2024-01-01"   # Valid date
assert validate_and_format_date("13/01/2023") == "Invalid Date" # Invalid month
assert validate_and_format_date("02/29/2024") == "2024-02-29"   # Leap year valid
assert validate_and_format_date("02/29/2023") == "Invalid Date" # Non-leap year

print("🎉 All date validation test cases passed successfully!")
```

```
🎉 All date validation test cases passed successfully!
```

Observation:

In this experiment, the validate_and_format_date(date_str) function was implemented to validate and convert dates from the "MM/DD/YYYY" format to "YYYY-MM-DD". The function uses Python's datetime module to check for correct formatting and valid calendar dates, including leap year considerations. Multiple test cases were used, including valid dates like "10/15/2023" and "01/01/2024", invalid dates such as "02/30/2023" and "13/01/2023", and edge cases like "02/29/2024" (leap year) and "02/29/2023" (non-leap year). The function correctly returned the formatted date for valid inputs and "Invalid Date" for invalid inputs. All assertion test cases passed successfully, demonstrating that the function reliably validates, converts, and handles typical and boundary date scenarios.