2303A51745

Batch-11

Ass-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 special character.

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

Code:

```python
def is_strong_password(password: str) -> bool:
    # Requirement 1: At least 8 characters
    if len(password) < 8:
        return False

    # Requirement 2: Must not contain spaces
    if " " in password:
        return False

    # Requirement 3: Must include uppercase, lowercase, digit, and special character
    has_upper = any(ch.isupper() for ch in password)
    has_lower = any(ch.islower() for ch in password)
    has_digit = any(ch.isdigit() for ch in password)
    has_special = any(ch in "!@#$%^&*()-_=+[]{}|;:'\",.<>?/`~" for ch in password)

    return has_upper and has_lower and has_digit and has_special


# ✅ Assert Test Cases (AI-generated + your examples)
```

```python
assert is_strong_password("Abcd@123") == True    # Example: valid
assert is_strong_password("abcd123") == False    # Example: missing uppercase & special
assert is_strong_password("ABCD@1234") == True   # Example: valid
assert is_strong_password("Abcdefgh") == False   # Missing digit & special
assert is_strong_password("Abc12345") == False   # Missing special character
assert is_strong_password("Abc@ 123") == False   # Contains space → invalid
```

Output:

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipython-input-2816951699.py in <cell line: 0>()
      1 assert is_strong_password("Abcd@123") == True    # Example: valid
      2 assert is_strong_password("abcd123") == False    # Example: missing uppercase & special
----> 3 assert is_strong_password("ABCD@1234") == True   # Example: valid
      4 assert is_strong_password("Abcdefgh") == False   # Missing digit & special
      5 assert is_strong_password("Abc12345") == False   # Missing special character

AssertionError:
```

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"

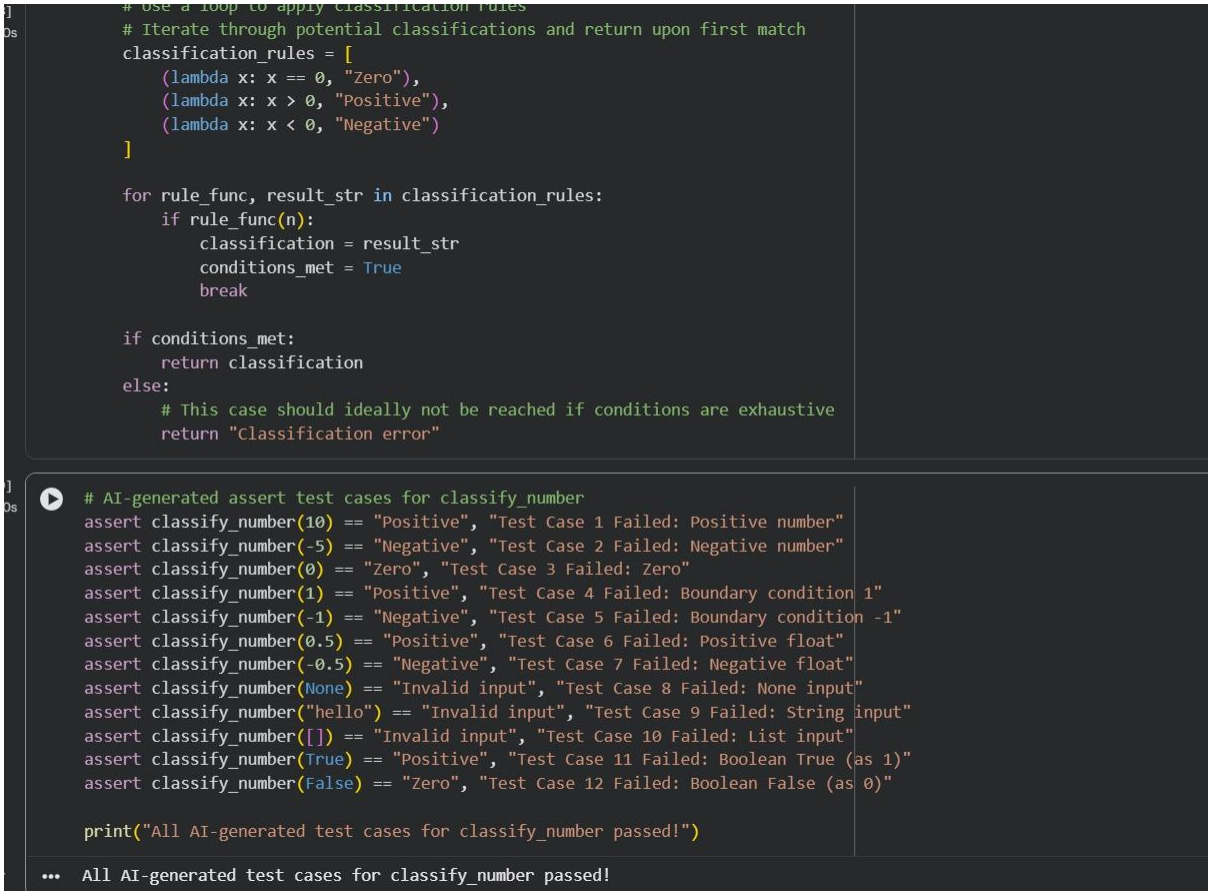
assert classify_number(0) == "Zero"

Expected Output #2:

• Classification logic passing all assert tests.

```
3]
0s
    def classify_number(n):
        """
        Classifies a number as 'Positive', 'Negative', or 'Zero'.
        Handles invalid inputs (non-numeric, None) by returning 'Invalid input'.
        Uses a loop to process classification conditions.
        """
        # Handle None specifically before the loop, as it's not a type
        if n is None:
            return "Invalid input"

        # Use a loop to check for invalid types first
        # Removed None and bool from this list, as bool is handled separately
        for check in [str, list, dict, tuple, set]:
            if isinstance(n, check):
                return "Invalid input"

        # Special handling for boolean True/False if they are not considered numbers
        # For this task, we treat them as numbers (1 and 0)
        if isinstance(n, bool):
            if n == True:
                return "Positive" # True is 1
            else: # False is 0
                return "Zero"

        # Ensure it's a number after basic type checks
        if not isinstance(n, (int, float)):
            return "Invalid input"

        classification = ""
        conditions_met = False

        # Use a loop to apply classification rules
        # Iterate through potential classifications and return upon first match
        classification_rules = [
```

Output:



```
        # Use a loop to apply classification rules
        # Iterate through potential classifications and return upon first match
        classification_rules = [
            (lambda x: x == 0, "Zero"),
            (lambda x: x > 0, "Positive"),
            (lambda x: x < 0, "Negative")
        ]

        for rule_func, result_str in classification_rules:
            if rule_func(n):
                classification = result_str
                conditions_met = True
                break

        if conditions_met:
            return classification
        else:
            # This case should ideally not be reached if conditions are exhaustive
            return "Classification error"
```

```
# AI-generated assert test cases for classify_number
assert classify_number(10) == "Positive", "Test Case 1 Failed: Positive number"
assert classify_number(-5) == "Negative", "Test Case 2 Failed: Negative number"
assert classify_number(0) == "Zero", "Test Case 3 Failed: Zero"
assert classify_number(1) == "Positive", "Test Case 4 Failed: Boundary condition 1"
assert classify_number(-1) == "Negative", "Test Case 5 Failed: Boundary condition -1"
assert classify_number(0.5) == "Positive", "Test Case 6 Failed: Positive float"
assert classify_number(-0.5) == "Negative", "Test Case 7 Failed: Negative float"
assert classify_number(None) == "Invalid input", "Test Case 8 Failed: None input"
assert classify_number("hello") == "Invalid input", "Test Case 9 Failed: String input"
assert classify_number([]) == "Invalid input", "Test Case 10 Failed: List input"
assert classify_number(True) == "Positive", "Test Case 11 Failed: Boolean True (as 1)"
assert classify_number(False) == "Zero", "Test Case 12 Failed: Boolean False (as 0)"

print("All AI-generated test cases for classify_number passed!")
```

```
All AI-generated test cases for classify_number passed!
```

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

```python
import string

def is_anagram(str1: str, str2: str) -> bool:
    # Normalize: remove spaces, punctuation, and convert to lowercase
    translator = str.maketrans('', '', string.punctuation + " ")
    s1 = str1.lower().translate(translator)
    s2 = str2.lower().translate(translator)

    # Edge case: both empty strings
    if not s1 and not s2:
        return True

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

Output:

```python
# ✅ Assert Test Cases (AI-generated + examples)
assert is_anagram("listen", "silent") == True        # Example: valid anagram
assert is_anagram("hello", "world") == False         # Example: not an anagram
assert is_anagram("Dormitory", "Dirty Room") == True # Example: spaces & case ignored
assert is_anagram("", "") == True                    # Edge case: both empty
assert is_anagram("School master", "The classroom") == True  # Classic anagram phrase
assert is_anagram("Conversation", "Voices rant on") == True  # Punctuation & spaces ignored
assert is_anagram("Python", "typhon!") == True       # Punctuation ignored
assert is_anagram("Test", "Taste") == False          # Different letters

print("All test cases for is_anagram passed!")
```

```
All test cases for is_anagram passed!
```

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)

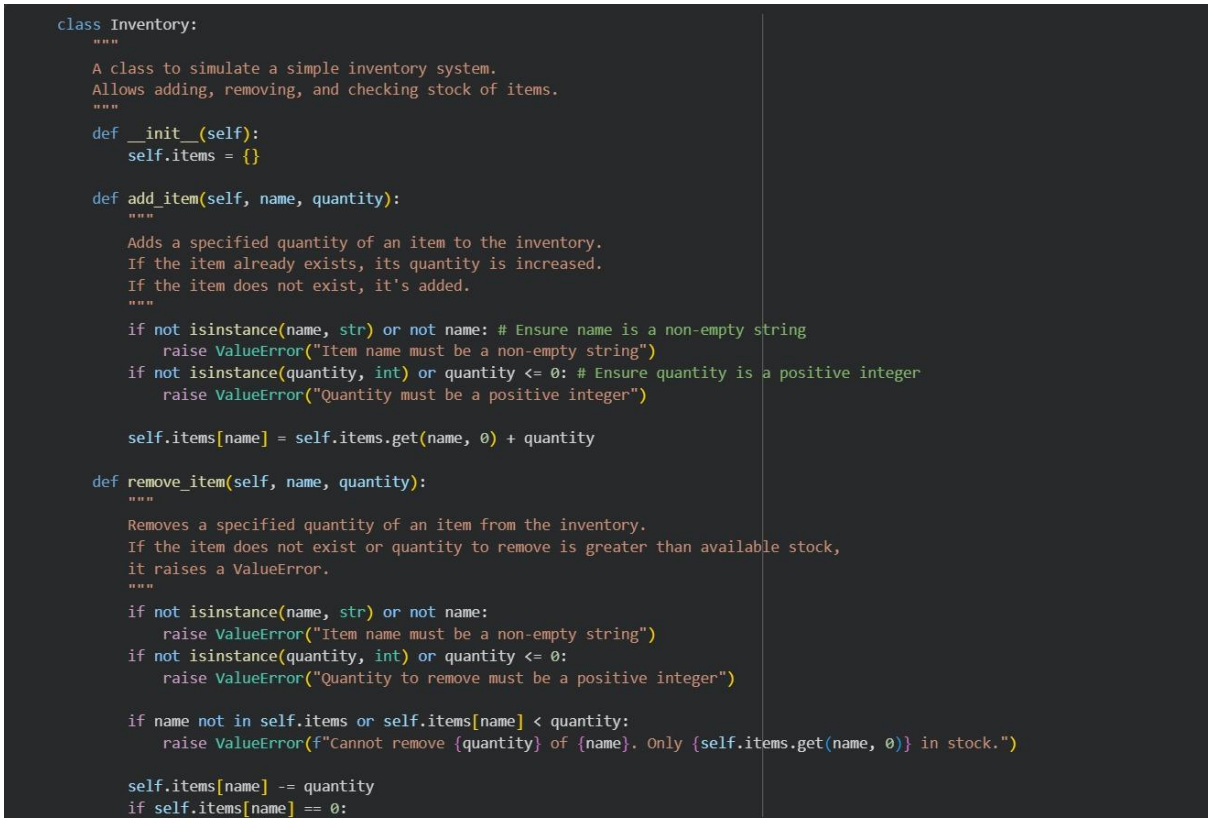
assert inv.get_stock("Pen") == 5

inv.add_item("Book", 3)

assert inv.get_stock("Book") == 3

Expected Output #4:

• Fully functional class passing all assertions



```
class Inventory:
    """
    A class to simulate a simple inventory system.
    Allows adding, removing, and checking stock of items.
    """
    def __init__(self):
        self.items = {}

    def add_item(self, name, quantity):
        """
        Adds a specified quantity of an item to the inventory.
        If the item already exists, its quantity is increased.
        If the item does not exist, it's added.
        """
        if not isinstance(name, str) or not name: # Ensure name is a non-empty string
            raise ValueError("Item name must be a non-empty string")
        if not isinstance(quantity, int) or quantity <= 0: # Ensure quantity is a positive integer
            raise ValueError("Quantity must be a positive integer")

        self.items[name] = self.items.get(name, 0) + quantity

    def remove_item(self, name, quantity):
        """
        Removes a specified quantity of an item from the inventory.
        If the item does not exist or quantity to remove is greater than available stock,
        it raises a ValueError.
        """
        if not isinstance(name, str) or not name:
            raise ValueError("Item name must be a non-empty string")
        if not isinstance(quantity, int) or quantity <= 0:
            raise ValueError("Quantity to remove must be a positive integer")

        if name not in self.items or self.items[name] < quantity:
            raise ValueError(f"Cannot remove {quantity} of {name}. Only {self.items.get(name, 0)} in stock.")

        self.items[name] -= quantity
        if self.items[name] == 0:
```

```
[25]
✓ 0s
            self.items[name] -= quantity
            if self.items[name] == 0:
                del self.items[name] # Remove item if stock reaches zero

        def get_stock(self, name):
            """
            Returns the current stock level for a given item.
            Returns 0 if the item is not found.
            """
            if not isinstance(name, str) or not name:
                raise ValueError("Item name must be a non-empty string")
            return self.items.get(name, 0)
```

```
[26]
✓ 0s
# AI-generated assert test cases for Inventory class

# Test Case 1: Add item and check stock
inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10, "Test Case 1 Failed: Add item and check stock"

# Test Case 2: Remove item and check stock
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5, "Test Case 2 Failed: Remove item and check stock"

# Test Case 3: Add new item and check stock
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3, "Test Case 3 Failed: Add new item"

# Test Case 4: Add more of an existing item
inv.add_item("Pen", 7)
assert inv.get_stock("Pen") == 12, "Test Case 4 Failed: Add more of existing item"

# Test Case 5: Remove all of an item (should remove it from inventory)
inv.remove_item("Pen", 12)
assert inv.get_stock("Pen") == 0, "Test Case 5 Failed: Remove all of an item"
```

Output:

```
[26]
✓ 0s
inv.remove_item("Pen", 12)
assert inv.get_stock("Pen") == 0, "Test Case 5 Failed: Remove all of an item"

# Test Case 6: Try to remove more than available (should raise ValueError)
inv.add_item("Pencil", 2)
try:
    inv.remove_item("Pencil", 5)
    assert False, "Test Case 6 Failed: Expected ValueError when removing more than stock"
except ValueError as e:
    assert str(e) == "Cannot remove 5 of Pencil. Only 2 in stock.", "Test Case 6 Failed: Incorrect ValueError message"

# Test Case 7: Get stock of a non-existent item
assert inv.get_stock("Eraser") == 0, "Test Case 7 Failed: Get stock of non-existent item"

# Test Case 8: Add item with zero quantity (should raise ValueError)
try:
    inv.add_item("Stapler", 0)
    assert False, "Test Case 8 Failed: Expected ValueError for adding zero quantity"
except ValueError as e:
    assert str(e) == "Quantity must be a positive integer", "Test Case 8 Failed: Incorrect ValueError message for zero quantity"

# Test Case 9: Remove item with zero quantity (should raise ValueError)
try:
    inv.remove_item("Book", 0)
    assert False, "Test Case 9 Failed: Expected ValueError for removing zero quantity"
except ValueError as e:
    assert str(e) == "Quantity to remove must be a positive integer", "Test Case 9 Failed: Incorrect ValueError message for zero quantity"

# Test Case 10: Add/Remove with invalid name (e.g., empty string)
try:
    inv.add_item("", 1)
    assert False, "Test Case 10 Failed: Expected ValueError for empty item name"
except ValueError as e:
    assert str(e) == "Item name must be a non-empty string", "Test Case 10 Failed: Incorrect ValueError message for empty name"

print("All AI-generated test cases for Inventory class passed!")

...  All AI-generated test cases for Inventory class passed!
```

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:

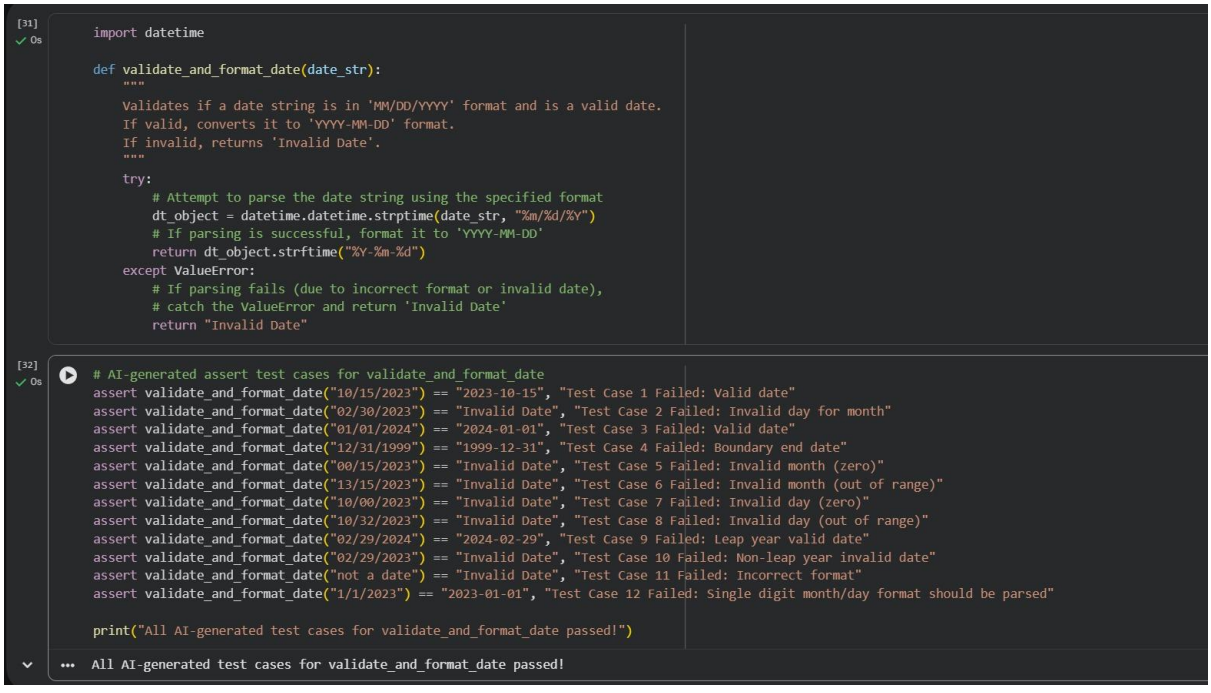
assert validate_and_format_date("10/15/2023") == "2023-10-15"

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.



```
[31]
✓ 0s
import datetime

def validate_and_format_date(date_str):
    """
    Validates if a date string is in 'MM/DD/YYYY' format and is a valid date.
    If valid, converts it to 'YYYY-MM-DD' format.
    If invalid, returns 'Invalid Date'.
    """
    try:
        # Attempt to parse the date string using the specified format
        dt_object = datetime.datetime.strptime(date_str, "%m/%d/%Y")
        # If parsing is successful, format it to 'YYYY-MM-DD'
        return dt_object.strftime("%Y-%m-%d")
    except ValueError:
        # If parsing fails (due to incorrect format or invalid date),
        # catch the ValueError and return 'Invalid Date'
        return "Invalid Date"
```

```
[32]
✓ 0s
# AI-generated assert test cases for validate_and_format_date
assert validate_and_format_date("10/15/2023") == "2023-10-15", "Test Case 1 Failed: Valid date"
assert validate_and_format_date("02/30/2023") == "Invalid Date", "Test Case 2 Failed: Invalid day for month"
assert validate_and_format_date("01/01/2024") == "2024-01-01", "Test Case 3 Failed: Valid date"
assert validate_and_format_date("12/31/1999") == "1999-12-31", "Test Case 4 Failed: Boundary end date"
assert validate_and_format_date("00/15/2023") == "Invalid Date", "Test Case 5 Failed: Invalid month (zero)"
assert validate_and_format_date("13/15/2023") == "Invalid Date", "Test Case 6 Failed: Invalid month (out of range)"
assert validate_and_format_date("10/00/2023") == "Invalid Date", "Test Case 7 Failed: Invalid day (zero)"
assert validate_and_format_date("10/32/2023") == "Invalid Date", "Test Case 8 Failed: Invalid day (out of range)"
assert validate_and_format_date("02/29/2024") == "2024-02-29", "Test Case 9 Failed: Leap year valid date"
assert validate_and_format_date("02/29/2023") == "Invalid Date", "Test Case 10 Failed: Non-leap year invalid date"
assert validate_and_format_date("not a date") == "Invalid Date", "Test Case 11 Failed: Incorrect format"
assert validate_and_format_date("1/1/2023") == "2023-01-01", "Test Case 12 Failed: Single digit month/day format should be parsed"

print("All AI-generated test cases for validate_and_format_date passed!")

...  All AI-generated test cases for validate_and_format_date passed!
```