Task

Implement the following Python functions and a class: `is_strong_password(password)` to validate password strength (at least 8 characters, includes uppercase, lowercase, digit, special character, no spaces) with at least 3 assert tests; `classify_number(n)` to categorize numbers as Positive, Negative, or Zero, handling invalid inputs like strings and None, including boundary conditions (-1, 0, 1), with at least 3 assert tests; `is_anagram(str1, str2)` to check if two strings are anagrams, ignoring case, spaces, and punctuation, and handling edge cases (empty strings, identical words), with at least 3 assert tests; and an `Inventory` class with `add_item(name, quantity)` and `remove_item(name, quantity)` methods for stock management, with at least 3 assert tests.

Password Strength Validator

Subtask:

Implement the `is_strong_password(password)` function based on the requirements (at least 8 characters, includes uppercase, lowercase, digit, and special character, no spaces). Generate at least 3 assert test cases for the function, and ensure the logic passes all tests.

Reasoning: First, I will define the `is_strong_password` function based on the given requirements, including checks for length, uppercase, lowercase, digit, special character, and spaces. Then, I will generate at least three assert test cases to validate the function's logic.

```
import re

def is_strong_password(password):
    """
    Checks if a password meets the strength requirements:
    - At least 8 characters long.
    - Includes at least one uppercase letter.
    - Includes at least one lowercase letter.
    - Includes at least one digit.
    - Includes at least one special character.
    - Does not contain any spaces.
    """
    if len(password) < 8:
        return False
    if not re.search(r"[A-Z]", password):
        return False
    if not re.search(r"[a-z]", password):
        return False
    if not re.search(r"\d", password):
        return False
    if not re.search(r"[!@#$%^&*()_+={}\[\]:;\"'<>,.?/~`\\-]", password):
        return False
    if " " in password:
        return False
```

```
    return True

# Test cases
assert is_strong_password("Password123!") == True, "Test Case 1 Failed: Strong password should pass"
assert is_strong_password("weakpw") == False, "Test Case 2 Failed: Too short"
assert is_strong_password("Nospacewith1!") == True, "Test Case 3 Failed: Valid password with no spaces"
assert is_strong_password("Password123") == False, "Test Case 4 Failed: No special character"
assert is_strong_password("password123!") == False, "Test Case 5 Failed: No uppercase letter"
assert is_strong_password("PASSWORD123!") == False, "Test Case 6 Failed: No lowercase letter"
assert is_strong_password("Password!@#") == False, "Test Case 7 Failed: No digit"
assert is_strong_password("Password 123!") == False, "Test Case 8 Failed: Contains space"

print("All assert tests passed!")
```

```
All assert tests passed!
```

Number Classification with Loops

Subtask:

Implement the `classify_number(n)` function using loops. The function should classify numbers as Positive, Negative, or Zero, and handle invalid inputs like strings and None, including boundary conditions (-1, 0, 1). Generate at least 3 assert test cases for the function, and ensure the classification logic passes all tests.

Reasoning: I will define the `classify_number(n)` function according to the requirements, handling numeric and non-numeric inputs, and then provide comprehensive assert test cases to validate its functionality.

```
def classify_number(n):
    """
    Classifies a number as 'Positive', 'Negative', or 'Zero'.
    Handles invalid inputs like strings and None.
    """
    try:
        # Attempt to convert input to a number (integer or float)
        # This handles cases where n might be a string representation of a number
        if isinstance(n, (int, float)):
            num = n
        elif isinstance(n, str):
            # Try to convert string to float for numerical comparison
            num = float(n)
        elif n is None:
            return 'Invalid Input'
        else:
            return 'Invalid Input'

        if num > 0:
            return 'Positive'
        elif num < 0:
            return 'Negative'
        else:
            return 'Zero'
```

```
        except (ValueError, TypeError):
            # Catch errors for non-numeric strings or other unconvertible types
            return 'Invalid Input'

    # Test cases
    assert classify_number(5) == 'Positive', "Test Case 1 Failed: Positive number"
    assert classify_number(-10) == '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 Positive"
    assert classify_number(-1) == 'Negative', "Test Case 5 Failed: Boundary Negative"
    assert classify_number('hello') == 'Invalid Input', "Test Case 6 Failed: String input"
    assert classify_number(None) == 'Invalid Input', "Test Case 7 Failed: None input"
    assert classify_number(3.14) == 'Positive', "Test Case 8 Failed: Float positive"
    assert classify_number('-5.5') == 'Negative', "Test Case 9 Failed: String float negative"
    assert classify_number('0') == 'Zero', "Test Case 10 Failed: String zero"
    assert classify_number('') == 'Invalid Input', "Test Case 11 Failed: Empty string"
    assert classify_number([]) == 'Invalid Input', "Test Case 12 Failed: List input"

    print("All assert tests for classify_number passed!")
```

```
All assert tests for classify_number passed!
```

Anagram Checker

Subtask:

Implement the `is_anagram(str1, str2)` function. The function should ignore case, spaces, and punctuation, and handle edge cases (empty strings, identical words). Generate at least 3 assert test cases for the function, and ensure it correctly identifies anagrams and passes all tests.

Reasoning:

First, I will define the `is_anagram` function, incorporating logic to normalize strings by converting them to lowercase, removing non-alphanumeric characters, and then comparing their sorted versions. Afterward, I will generate comprehensive assert test cases to validate its functionality, including edge cases like empty strings, identical words, and cases with varying spaces and punctuation.

Reasoning: I will implement the `is_anagram` function following the specified requirements, including normalization of strings and comparison of sorted characters. Then, I will add assert test cases covering various scenarios and edge cases to ensure the function's correctness.

```
import re

def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams, ignoring case, spaces, and punctuation.
    """
    def normalize_string(s):
        # Convert to lowercase
        s = s.lower()
        # Remove non-alphanumeric characters (keep only letters and digits)
```

```
        # Remove non-alphanumeric characters (keep only letters and digits)
        s = re.sub(r'[^a-z0-9]', '', s)
        return s

    normalized_str1 = normalize_string(str1)
    normalized_str2 = normalize_string(str2)

    # Anagrams must have the same characters, so their sorted versions must be identical
    return sorted(normalized_str1) == sorted(normalized_str2)

# Test cases
assert is_anagram('Listen', 'Silent') == True, "Test Case 1 Failed: Basic anagrams"
assert is_anagram('Debit card', 'Bad credit') == True, "Test Case 2 Failed: Anagrams with spaces"
assert is_anagram('The eyes', 'They see') == True, "Test Case 3 Failed: Anagrams with spaces and different words"
assert is_anagram('A gentleman', 'Elegant man') == True, "Test Case 4 Failed: Anagrams with spaces and different words"
assert is_anagram('Rail safety', 'Fairly at ease') == False, "Test Case 5 Failed: Anagrams with more complex phrases"
assert is_anagram('Hello', 'World') == False, "Test Case 6 Failed: Not anagrams"
assert is_anagram('', '') == True, "Test Case 7 Failed: Both empty strings"
assert is_anagram('a', 'a') == True, "Test Case 8 Failed: Identical single characters"
assert is_anagram('a', 'b') == False, "Test Case 9 Failed: Different single characters"
assert is_anagram('hello!', 'olleh.') == True, "Test Case 10 Failed: Anagrams with punctuation"
assert is_anagram('Race', 'Care') == True, "Test Case 11 Failed: Identical words with different case"
assert is_anagram('Anagram', 'Nag a ram') == True, "Test Case 12 Failed: Anagram with spaces and different word order"
assert is_anagram('!@#$', '$$#@!') == True, "Test Case 13 Failed: Strings with only non-alphanumeric chars (normalize to empty string)"
assert is_anagram('  ', '') == True, "Test Case 14 Failed: Empty string and string with spaces"


print("All assert tests for is_anagram passed!")
```

```
All assert tests for is_anagram passed!
```

Reasoning: The previous code failed an assertion because 'Rail safety' and 'Fairly at ease' are not anagrams due to different character counts. I will correct the expected outcome of this specific test case to `False` to fix the assertion error.

```
import re

def is_anagram(str1, str2):
    """
    Checks if two strings are anagrams, ignoring case, spaces, and punctuation.
    """
    def normalize_string(s):
        # Convert to lowercase
        s = s.lower()
        # Remove non-alphanumeric characters (keep only letters and digits)
        s = re.sub(r'[^a-z0-9]', '', s)
        return s

    normalized_str1 = normalize_string(str1)
    normalized_str2 = normalize_string(str2)

    # Anagrams must have the same characters, so their sorted versions must be identical
    return sorted(normalized_str1) == sorted(normalized_str2)

# Test cases
assert is_anagram('Listen', 'Silent') == True, "Test Case 1 Failed: Basic anagrams"
```

```
assert is_anagram('Debit card', 'Bad credit') == True, "Test Case 2 Failed: Anagrams with spaces"
assert is_anagram('The eyes', 'They see') == True, "Test Case 3 Failed: Anagrams with spaces and different words"
assert is_anagram('A gentleman', 'Elegant man') == True, "Test Case 4 Failed: Anagrams with spaces and different words"
assert is_anagram('Rail safety', 'Fairly at ease') == False, "Test Case 5 Failed: Anagrams with more complex phrases"
assert is_anagram('Hello', 'World') == False, "Test Case 6 Failed: Not anagrams"
assert is_anagram('', '') == True, "Test Case 7 Failed: Both empty strings"
assert is_anagram('a', 'a') == True, "Test Case 8 Failed: Identical single characters"
assert is_anagram('a', 'b') == False, "Test Case 9 Failed: Different single characters"
assert is_anagram('hello!', 'olleh.') == True, "Test Case 10 Failed: Anagrams with punctuation"
assert is_anagram('Race', 'Care') == True, "Test Case 11 Failed: Identical words with different case"
assert is_anagram('Anagram', 'Nag a ram') == True, "Test Case 12 Failed: Anagram with spaces and different word order"
assert is_anagram('!@#$', '$$#@!') == True, "Test Case 13 Failed: Strings with only non-alphanumeric chars (normalize to empty string)"
assert is_anagram('  ', '') == True, "Test Case 14 Failed: Empty string and string with spaces"


print("All assert tests for is_anagram passed!")
```

```
All assert tests for is_anagram passed!
```

Inventory Class

Subtask:

Implement an `Inventory` class with `add_item(name, quantity)` and `remove_item(name, quantity)` methods to simulate a real-world inventory system with stock management. Generate at least 3 assert-based tests to cover the functionality of the class and its methods.

Summary:

Data Analysis Key Findings

- **Password Strength Validator**: The `is_strong_password` function was successfully implemented, validating passwords against criteria including a minimum length of 8 characters, presence of uppercase, lowercase, digit, and special characters, and absence of spaces. The function was rigorously tested with 8 assert cases, exceeding the minimum requirement, and all tests passed, confirming its accuracy and robustness.
- **Number Classification**: The `classify_number` function correctly categorizes numbers as 'Positive', 'Negative', or 'Zero'. It effectively handles a wide range of inputs, including integers, floats, and string representations of numbers, and robustly manages invalid inputs such as arbitrary strings, `None`, empty strings, and other non-numeric types by returning 'Invalid Input'. This functionality was verified through 12 comprehensive assert tests, all of which passed.
- **Anagram Checker**: The `is_anagram` function was successfully implemented to determine if two strings are anagrams, ignoring case, spaces, and punctuation. It employs a `normalize_string` helper to preprocess strings for accurate comparison. The function was thoroughly tested with 14 assert cases, covering various scenarios including basic anagrams, phrases with spaces, punctuation, non-anagrams, and edge cases like empty strings. A minor correction was made during testing to accurately reflect that `'Rail safety'` and `'Fairly at ease'` are not anagrams, after which all tests passed.

Insights or Next Steps

- All implemented functions demonstrated robust error handling and comprehensive test coverage, indicating a strong understanding of the requirements and potential edge cases.
- Future enhancements could include implementing the `Inventory` class as specified in the overall task, ensuring similar levels of test coverage and error handling for its methods.

Task

Implement an `Inventory` class with `add_item(name, quantity)` and `remove_item(name, quantity)` methods and generate at least 3 assert-based tests to cover the functionality of the class and its methods.

Inventory Class

Subtask:

Implement an `Inventory` class with `add_item(name, quantity)` and `remove_item(name, quantity)` methods to simulate a real-world inventory system with stock management. Generate at least 3 assert-based tests to cover the functionality of the class and its methods.

Summary:

Data Analysis Key Findings

- An `Inventory` class was successfully implemented, designed to manage item stock using an internal dictionary to store item names and their corresponding quantities.
- The `add_item` method correctly handles adding new items to the inventory and updating the quantity of existing items.
- The `remove_item` method includes robust error handling for two scenarios:
  - Attempting to remove an item that does not exist in the inventory.
  - Attempting to remove a quantity greater than the available stock, preventing negative stock levels.
- At least three assert-based tests were generated, covering the core functionalities:
  - Adding new items to an empty inventory.
  - Adding more quantity to an existing item.
  - Removing items, including scenarios where the item is completely removed, and where only a partial quantity is removed.
  - Error cases for `remove_item` where the item is not found or the removal quantity exceeds available stock.

Insights or Next Steps

- The current `Inventory` class provides a foundational stock management system; consider extending its capabilities with methods like `get_item_quantity(name)` for querying stock levels, or `list_all_items()` to retrieve the full inventory.
- To further enhance the system's robustness, implement additional validation for input parameters (e.g., ensuring `quantity` is a non-negative integer) and explore handling edge cases such as concurrent access to the inventory in a multi-threaded environment.