Lab Assignment 8.1

**Name:** Bhavaneesh.A

**Roll No:**  2403A52350

**Batch:**  13

**TASK-1:** Password Strength Validator

Implement is\_strong\_password(password) function to validate passwords.

Requirements:

- At least 8 characters

- Must include uppercase, lowercase, digit, and special character

- Must not contain spaces

**Code:**

****

**A white background with text

AI-generated content may be incorrect.**

**Output:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Explanation:**

This code contains a Python function called is\_strong\_password and several assert statements to test it.

Here's a breakdown of the code:

1. import re: This line imports the re module, which provides support for regular expressions in Python. Regular expressions are used here to check for the presence of uppercase letters, lowercase letters, digits, and special characters.
2. def is\_strong\_password(password):: This defines the function is\_strong\_password that takes one argument, password, which is the string to be validated.
3. print(f"Checking password: {password}"): This line (and other similar print lines within the function) were added for debugging purposes to show which password is being checked and which validation step might be failing.
4. if len(password) < 8:: This checks if the length of the password is less than 8 characters. If it is, the function prints a failure message and returns False.
5. if ' ' in password:: This checks if the password contains any spaces. If it does, the function prints a failure message and returns False.
6. if not re.search(r'[A-Z]', password):: This uses a regular expression [A-Z] to search for at least one uppercase letter in the password. re.search() returns a match object if found, otherwise None. The not keyword negates the result, so if no uppercase letter is found (re.search returns None), the condition is True, and the function prints a failure message and returns False.
7. if not re.search(r'[a-z]', password):: Similar to the uppercase check, this uses the regex [a-z] to search for at least one lowercase letter.
8. if not re.search(r'\d', password):: This uses the regex \d to search for at least one digit (0-9).
9. special\_char\_match = re.search(r'[!@#$%^&\*()\_+{}\[\]:;<>,.?~\\-]', password): This is the check for special characters. The regular expression [!@#$%^&\*()\_+{}\[\]:;<>,.?~\\-] defines a character set [...] that includes a variety of common special characters. The re.search() function attempts to find any of these characters in the password and stores the result in the special\_char\_match variable.
10. print(f"Special character regex match for {password}: {special\_char\_match}"): This print statement shows the result of the special character regex search (either a match object or None).
11. if not special\_char\_match:: This checks if special\_char\_match is None, meaning no special character from the defined set was found in the password. If no special character is found, the function prints a failure message and returns False.
12. print("Passed all checks"): If the password passes all the previous checks, this line is printed.
13. return True: If the password passes all checks, the function returns True, indicating it is a strong password.
14. assert is\_strong\_password(...) == ...: These lines are assert statements used for testing. They call the is\_strong\_password function with specific inputs and assert that the returned value is equal to the expected value (True or False). If an assertion fails, it raises an AssertionError, indicating a problem with the function's logic.

In summary, the code defines and tests a function that validates password strength based on specific length, content (uppercase, lowercase, digit, special character), and format (no spaces) requirements.

**TASK-2:** Number Classification

Implement classify\_number(n) function using loops.

Requirements:

- Classify numbers as Positive, Negative, or Zero

- Handle invalid inputs like strings and None

- Include boundary conditions (-1, 0, 1)

**Code:**

A screen shot of a computer code

AI-generated content may be incorrect.

**Output:**

**A white background with black text

AI-generated content may be incorrect.**

**Explanation:**

This code contains the Python function classify\_number(n) and a loop to run several test cases against it using assert statements.

Here's a breakdown:

1. def classify\_number(n):: This defines the function classify\_number which takes one argument n.
2. if not isinstance(n, (int, float)):: This checks if the input n is NOT an instance of either an integer (int) or a floating-point number (float). If it's not, it means the input is invalid (like a string or None), and the function returns the string "Invalid Input".
3. if n > 0:: If the input is a number, this checks if it is greater than 0. If true, it returns the string "Positive".
4. elif n < 0:: If the number is not greater than 0, this checks if it is less than 0. If true, it returns the string "Negative".
5. else:: If the number is neither greater than 0 nor less than 0, it must be 0. In this case, the function returns the string "Zero".
6. test\_cases = [...]: This line creates a list of tuples called test\_cases. Each tuple contains an input value for the classify\_number function and the expected output string for that input. This list includes examples of positive, negative, and zero numbers, as well as boundary conditions (-1, 0, 1) and invalid inputs ("abc", None).
7. for input\_value, expected\_output in test\_cases:: This is a for loop that iterates through each tuple in the test\_cases list. In each iteration, it unpacks the tuple into two variables: input\_value and expected\_output.
8. actual\_output = classify\_number(input\_value): Inside the loop, this line calls the classify\_number function with the current input\_value and stores the returned result in the actual\_output variable.
9. assert actual\_output == expected\_output, f"Input: {input\_value}, Expected: {expected\_output}, Got: {actual\_output}": This is an assert statement. It checks if the actual\_output returned by the function is equal to the expected\_output from the test\_cases list. If they are not equal, an AssertionError is raised, and the message provided in the f-string is displayed, showing which input failed and what the expected and actual outputs were. If the assertion is true, nothing happens, and the loop continues.
10. print(f"Test passed for input: {input\_value}"): If an assertion passes without raising an error, this line is printed to confirm that the test for that specific input was successful.
11. print("All test cases passed!"): After the loop finishes without any AssertionError being raised, this line is printed to indicate that all the defined test cases have passed successfully.

In essence, this cell defines the number classification logic and then systematically tests that logic using a predefined set of inputs and their expected results.

**TASK-3:** Anagram Checker

Implement is\_anagram(str1, str2).

Requirements:

- Ignore case, spaces, and punctuation

- Handle edge cases (empty strings, identical words)

**Code:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A close-up of a number

AI-generated content may be incorrect.**

**Output:**

**A computer screen shot of text

AI-generated content may be incorrect.**

**Explanation:**

This code contains the Python function is\_anagram(str1, str2) and a loop to run several test cases against it using assert statements.

Here's a breakdown:

1. import re: This line imports the re module, which provides support for regular expressions in Python. Regular expressions are used here to remove unwanted characters (spaces and punctuation).
2. def is\_anagram(str1, str2):: This defines the function is\_anagram that takes two string arguments, str1 and str2.
3. def clean\_string(s):: This defines a helper function clean\_string that takes a single string s as input.
   * s = s.lower(): Converts the input string to lowercase. This ensures that case is ignored when checking for anagrams.
   * s = re.sub(r'[^a-z0-9]', '', s): This is where regular expressions are used. re.sub() is used to substitute (replace) patterns in a string.
     + r'[^a-z0-9]': This is the regular expression pattern.
       - [...]: Defines a character set.
       - ^: When at the beginning of a character set, it negates the set, meaning it matches any character *not* in the set.
       - a-z: Matches any lowercase letter from 'a' to 'z'.
       - 0-9: Matches any digit from '0' to '9'.
     + So, [^a-z0-9] matches any character that is *not* a lowercase letter or a digit.
     + '': This is the replacement string. It's an empty string, meaning any matched characters are removed.
   * return s: The function returns the cleaned string, which now only contains lowercase letters and digits.
4. cleaned\_str1 = clean\_string(str1): Calls the clean\_string helper function on the first input string str1 and stores the result.
5. cleaned\_str2 = clean\_string(str2): Calls the clean\_string helper function on the second input string str2 and stores the result.
6. if len(cleaned\_str1) != len(cleaned\_str2):: This checks if the lengths of the two cleaned strings are different. If they are, the strings cannot be anagrams, so the function returns False.
7. return sorted(cleaned\_str1) == sorted(cleaned\_str2): This is the core logic for checking anagrams.
   * sorted(cleaned\_str1): Sorts the characters in the cleaned\_str1 alphabetically and returns a list of characters.
   * sorted(cleaned\_str2): Sorts the characters in the cleaned\_str2 alphabetically and returns a list of characters.
   * ==: Compares the two sorted lists of characters. If two strings are anagrams (ignoring case, spaces, and punctuation), their cleaned versions will have the same characters, and thus, the sorted lists of their characters will be identical. The function returns True if the sorted lists are equal (they are anagrams) and False otherwise.
8. test\_cases = [...]: This line creates a list of tuples called test\_cases. Each tuple contains two input strings for the is\_anagram function and the expected boolean output (True if they are anagrams, False otherwise). This list includes various scenarios like basic anagrams, non-anagrams, strings with different cases, spaces, punctuation, numbers, empty strings, and identical words, covering the requirements.
9. for str1, str2, expected\_output in test\_cases:: This is a for loop that iterates through each tuple in the test\_cases list. In each iteration, it unpacks the tuple into str1, str2, and expected\_output.
10. actual\_output = is\_anagram(str1, str2): Inside the loop, this line calls the is\_anagram function with the current input strings str1 and str2 and stores the returned boolean result in the actual\_output variable.
11. assert actual\_output == expected\_output, f"Input: ('{str1}', '{str2}'), Expected: {expected\_output}, Got: {actual\_output}": This is an assert statement. It checks if the actual\_output returned by the function is equal to the expected\_output from the test\_cases list. If they are not equal, an AssertionError is raised with a descriptive message showing the input strings, expected output, and actual output.
12. print(f"Test passed for input: ('{str1}', '{str2}')"): If an assertion passes, this line is printed to confirm the test for that specific input pair was successful.
13. print("All test cases passed!"): After the loop completes without any assertions failing, this line is printed to indicate that all test cases have passed.

In summary, this cell defines a function to check for anagrams while ignoring certain characters and case, and then uses a comprehensive set of test cases to verify the function's correctness.

**TASK-4:** Inventory Class

Implement Inventory class with stock management.

Methods:

- add\_item(name, quantity)

- remove\_item(name, quantity)

- get\_stock(name)

**Code:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Explanation:**

This code contains the Python Inventory class definition and a series of assert statements to test its methods.

Here's a breakdown:

1. class Inventory:: This line defines a new class named Inventory. Classes are blueprints for creating objects (instances) that have properties (data) and methods (functions) associated with them.
2. def \_\_init\_\_(self):: This is the constructor method. It's automatically called when you create a new Inventory object.
   * self.stock = {}: This initializes an empty dictionary called stock as an instance variable (self.stock). This dictionary will store the inventory items, where the keys will be the item names (strings) and the values will be the quantities (integers).
3. def add\_item(self, name, quantity):: This method is used to add a specified quantity of an item with the given name to the inventory.
   * if quantity <= 0:: It first checks if the quantity to add is positive. If not, it prints a message and returns without adding the item.
   * if name in self.stock:: Checks if the item name already exists as a key in the stock dictionary.
     + self.stock[name] += quantity: If the item exists, the quantity is added to the current stock of that item.
     + else: self.stock[name] = quantity: If the item does not exist, it's added to the stock dictionary with the specified quantity.
   * print(f"Added {quantity} of {name}. Current stock: {self.stock[name]}"): Prints a confirmation message showing the quantity added and the new total stock for the item.
4. def remove\_item(self, name, quantity):: This method is used to remove a specified quantity of an item with the given name from the inventory.
   * if quantity <= 0:: It first checks if the quantity to remove is positive. If not, it prints a message and returns without removing the item.
   * if name in self.stock:: Checks if the item name exists in the stock dictionary.
     + if self.stock[name] >= quantity:: If the item exists, this checks if there is enough stock to remove the specified quantity.
       - self.stock[name] -= quantity: If there is enough stock, the quantity is subtracted from the current stock.
       - print(f"Removed {quantity} of {name}. Remaining stock: {self.stock[name]}"): Prints a confirmation message showing the quantity removed and the remaining stock.
       - if self.stock[name] == 0: del self.stock[name]: If the stock of the item becomes 0 after removal, the item is removed from the stock dictionary using del. A message is printed indicating the item is out of stock.
     + else: print(f"Not enough {name} in stock. Available: {self.stock[name]}"): If there is not enough stock, a message is printed indicating the available stock.
   * else: print(f"{name} is not in the inventory."): If the item does not exist in the inventory, a message is printed.
5. def get\_stock(self, name):: This method returns the current stock quantity of an item with the given name.
   * return self.stock.get(name, 0): This uses the dictionary's get() method. It tries to retrieve the value associated with the key name from the self.stock dictionary. If the name is not found in the dictionary, it returns the default value specified, which is 0 in this case.
6. inv = Inventory(): This line creates an instance (an object) of the Inventory class and assigns it to the variable inv. The \_\_init\_\_ method is called automatically at this point, initializing inv.stock as an empty dictionary.
7. assert inv.get\_stock("Pen") == 10 and subsequent assert lines: These are assert statements used to test the functionality of the Inventory class methods.
   * They call the methods (add\_item, remove\_item, get\_stock) on the inv object with specific inputs.
   * They then assert that the result of get\_stock() after these operations is equal to the expected value.
   * For example, after adding 10 "Pen" items, assert inv.get\_stock("Pen") == 10 checks if the stock of "Pen" is indeed 10.
   * These tests cover adding new items, adding to existing items, removing items, removing all of an item, attempting to remove more than available, attempting to remove an item not in inventory, and attempting to add/remove with zero or negative quantities.
8. print("All assert tests passed!"): If all the assert statements execute without raising an AssertionError, this line is printed, indicating that all the tests for the Inventory class have passed successfully.

In summary, this code defines a class to manage inventory with methods for adding, removing, and checking stock, and then includes a set of assertions to automatically test if these methods behave correctly under different scenarios.

**TASK-5:** Date Validation & Formatting

Implement validate\_and\_format\_date(date\_str).

Requirements:

- Validate 'MM/DD/YYYY' format

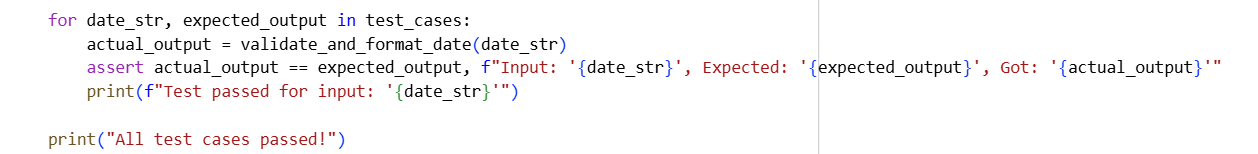
- Handle invalid dates

- Convert valid dates to 'YYYY-MM-DD'

**Code:**

**A screenshot of a computer code

AI-generated content may be incorrect.**

****

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Explanation:**

This code contains the Python function validate\_and\_format\_date(date\_str) and a loop to run several test cases against it using assert statements.

Here's a breakdown:

1. from datetime import datetime: This line imports the datetime class from the datetime module. This class is used for working with dates and times, specifically for parsing and formatting date strings.
2. def validate\_and\_format\_date(date\_str):: This defines the function validate\_and\_format\_date which takes one argument date\_str, the string representing the date to be validated and formatted.
3. if not isinstance(date\_str, str):: This checks if the input date\_str is not a string. If it's not a string (e.g., None), it's considered an invalid input, and the function returns the string "Invalid Date".
4. try:: This starts a try block, which is used to handle potential errors (exceptions) that might occur during the execution of the code within this block.
5. date\_obj = datetime.strptime(date\_str, "%m/%d/%Y"): This is the core of the validation and parsing.
   * datetime.strptime(): This class method attempts to parse a string (date\_str) according to a specified format code ("%m/%d/%Y").
     + "%m": Represents the month as a zero-padded decimal number (e.g., 01, 10).
     + "%d": Represents the day of the month as a zero-padded decimal number (e.g., 01, 15).
     + "%Y": Represents the year with century as a decimal number (e.g., 2023).
     + The slashes / in the format string match the literal slashes in the input date string.
   * If the date\_str successfully matches this format *and* represents a valid date (e.g., no 31st of February), strptime() returns a datetime object, which is stored in date\_obj.
   * If the date\_str does *not* match the format or represents an invalid date, strptime() raises a ValueError.
6. return date\_obj.strftime("%Y-%m-%d"): If strptime() is successful (i.e., no ValueError is raised), this line is executed.
   * date\_obj.strftime(): This method formats a datetime object into a string according to a specified format code ("%Y-%m-%d").
     + "%Y": Year with century.
     + "%m": Month as a zero-padded decimal number.
     + "%d": Day of the month as a zero-padded decimal number.
     + The hyphens - are included as literal characters in the output string.
   * The function returns the formatted date string in "YYYY-MM-DD" format.
7. except ValueError:: This block is executed if a ValueError is raised within the try block (meaning strptime() failed).
   * return "Invalid Date": If a ValueError occurs, the function returns the string "Invalid Date", indicating that the input string was not a valid date in the expected format.
8. test\_cases = [...]: This line creates a list of tuples called test\_cases. Each tuple contains an input date string for the validate\_and\_format\_date function and the expected output string (either a formatted date or "Invalid Date"). This list includes various scenarios like valid dates, invalid dates (including leap year checks), incorrect formats, non-date strings, and None input, covering the requirements.
9. for date\_str, expected\_output in test\_cases:: This is a for loop that iterates through each tuple in the test\_cases list, unpacking each tuple into date\_str and expected\_output.
10. actual\_output = validate\_and\_format\_date(date\_str): Inside the loop, this line calls the validate\_and\_format\_date function with the current date\_str and stores the returned result in the actual\_output variable.
11. assert actual\_output == expected\_output, f"Input: '{date\_str}', Expected: '{expected\_output}', Got: '{actual\_output}'": This is an assert statement. It checks if the actual\_output matches the expected\_output. If they don't match, an AssertionError is raised with a message detailing the input and the expected vs. actual outputs.
12. print(f"Test passed for input: '{date\_str}'"): If an assertion passes, this line is printed to confirm that the test for that specific input was successful.
13. print("All test cases passed!"): After the loop finishes without any AssertionError being raised, this line is printed to indicate that all the defined test cases have passed successfully.

In essence, this cell defines a function that validates and formats date strings using the datetime module's parsing capabilities and includes a set of test cases to ensure it handles both valid and invalid inputs correctly.