LAB ASSIGNMENT – 8.1

NAME : M.Shiva

HALL.NO : 2403A52377

BATCH.NO : AI 14

PLATFORM USED : GOOGLE COLLAB AND perplexity.ai

PROMPT 01 :

write a python code that generate at least 3 assert test cases for is_strong_password(password) and implement the validator function.
Requirements: Password must have at least 8 characters. Must include uppercase, lowercase, digit, and special character. Must not contain spaces.

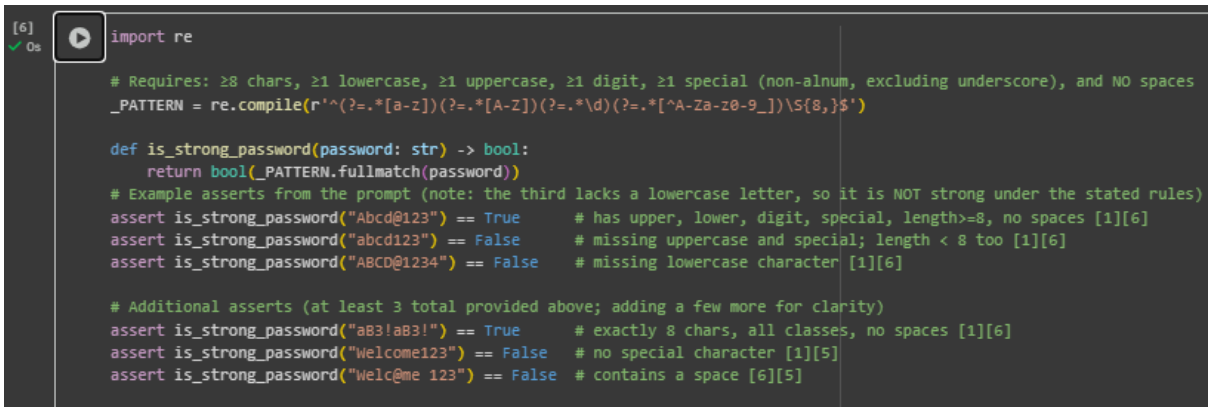
Example Assert Test Cases:

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

assert is_strong_password("abcd123") == False

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

OUTPUT :



```
[6]
✓ 0s
import re

# Requires: ≥8 chars, ≥1 lowercase, ≥1 uppercase, ≥1 digit, ≥1 special (non-alnum, excluding underscore), and NO spaces
_PATTERN = re.compile(r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[^A-Za-z0-9_])\S{8,}$')

def is_strong_password(password: str) -> bool:
    return bool(_PATTERN.fullmatch(password))
# Example asserts from the prompt (note: the third lacks a lowercase letter, so it is NOT strong under the stated rules)
assert is_strong_password("Abcd@123") == True      # has upper, lower, digit, special, length>=8, no spaces [1][6]
assert is_strong_password("abcd123") == False      # missing uppercase and special; length < 8 too [1][6]
assert is_strong_password("ABCD@1234") == False    # missing lowercase character [1][6]

# Additional asserts (at least 3 total provided above; adding a few more for clarity)
assert is_strong_password("aB3!aB3!") == True      # exactly 8 chars, all classes, no spaces [1][6]
assert is_strong_password("Welcome123") == False   # no special character [1][5]
assert is_strong_password("Welc@me 123") == False  # contains a space [6][5]
```

CODE EXPLANATION :

- Pattern logic: use lookaheads to “require” character types: (?=.*[a-z]) lowercase, (?=.*[A-Z]) uppercase, (?=.*\d) digit, (?=.*[^A-Za-z0-9_]) special character (non-alphanumeric excluding underscore).
- Length and spaces: \S{8,} at the end enforces at least 8 characters and forbids any whitespace (spaces, tabs, newlines).
- Full string check: .fullmatch(...) anchors the whole password so partial matches can’t pass.
- Why “ABCD@1234” is False: it has uppercase, digit, special, but no lowercase letter; the rule needs all four categories.
- Test coverage idea: include positive exact-length case (e.g., 8 chars), negatives for missing special/uppercase, and a case with a space to confirm whitespace rejection.

PROMPT 02 :

write a python code that generate at least 3 assert test cases for a classify_number(n) function. Implement using loops. Requirements: Classify numbers as Positive, Negative, or Zero.Handle invalid inputs like strings and None. 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"

OUTPUT :

```python
from numbers import Number

def classify_number(n):
    # Validate type using ABC for numeric types (int, float, Decimal, etc.)
    if not isinstance(n, Number) or isinstance(n, bool):
        return "Invalid"  # reject strings, None, booleans, etc. [4]

    # Use a tiny loop to evaluate conditions in order without nested ifs
    checks = [
        ("Zero", lambda x: x == 0),
        ("Positive", lambda x: x > 0),
        ("Negative", lambda x: x < 0),
    ]
    for label, predicate in checks:
        if predicate(n):
            return label
    return "Invalid"  # fallback; logically unreachable
# Examples from prompt
assert classify_number(10) == "Positive"   # greater than 0 → Positive [1][7]
assert classify_number(-5) == "Negative"   # less than 0 → Negative [1][7]
assert classify_number(0) == "Zero"        # equals 0 → Zero [1][7]

# Boundary conditions
assert classify_number(-1) == "Negative"   # boundary just below zero [7]
assert classify_number(1) == "Positive"    # boundary just above zero [7]

# Invalid inputs
assert classify_number("5") == "Invalid"   # string is not numeric [4]
assert classify_number(None) == "Invalid"  # None is invalid [4]
assert classify_number(True) == "Invalid"  # bool excluded though it is int subclass [4]
```

CODE EXPLANATION :

- Type check first: reject non-numeric inputs using isinstance(n, Number) and also exclude booleans (since True/False are subclasses of int) to return "Invalid".
- Use a small loop of labeled checks instead of nested if-elif: iterate over predicates for Zero (x == 0), Positive (x > 0), and Negative (x < 0), and return the matching label.
- Boundary coverage: explicit asserts for -1, 0, and 1 verify correct behavior around zero.
- Invalid input coverage: asserts for "5", None, and True ensure strings, NoneType, and booleans are handled safely.
- Deterministic order: checking Zero first avoids misclassifying 0 as Positive/Negative, then >0 for Positive, and finally <0 for Negative.

PROMPT 03 :

write a python code that generate at least 3 assert test cases for is_anagram(str1, str2) and implement the function.

Requirements:Ignore case, spaces, and punctuation.

Handle edge cases (empty strings, identical words).

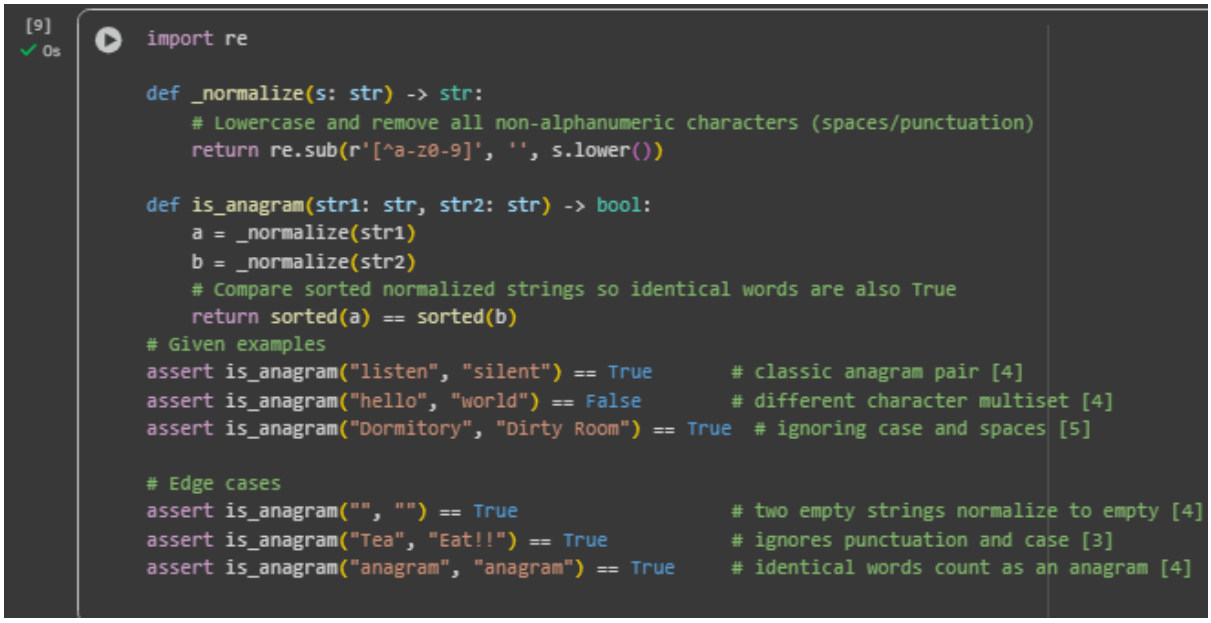
Example Assert Test Cases:

assert is_anagram("listen", "silent") == True

assert is_anagram("hello", "world") == False

assert is_anagram("Dormitory", "Dirty Room") == True

OUTPUT :



```
[9]
✓ 0s
import re

def _normalize(s: str) -> str:
    # Lowercase and remove all non-alphanumeric characters (spaces/punctuation)
    return re.sub(r'[^a-z0-9]', '', s.lower())

def is_anagram(str1: str, str2: str) -> bool:
    a = _normalize(str1)
    b = _normalize(str2)
    # Compare sorted normalized strings so identical words are also True
    return sorted(a) == sorted(b)
# Given examples
assert is_anagram("listen", "silent") == True       # classic anagram pair [4]
assert is_anagram("hello", "world") == False        # different character multiset [4]
assert is_anagram("Dormitory", "Dirty Room") == True  # ignoring case and spaces [5]

# Edge cases
assert is_anagram("", "") == True                   # two empty strings normalize to empty [4]
assert is_anagram("Tea", "Eat!!") == True           # ignores punctuation and case [3]
assert is_anagram("anagram", "anagram") == True     # identical words count as an anagram [4]
```

CODE EXPLANATION :

- Normalize both strings: convert to lowercase and remove everything except letters/digits using a regex like re.sub(r'[^a-z0-9]', '', s.lower()), so case, spaces, and punctuation are ignored.
- Compare character multisets: sort the normalized strings and check equality; if the sorted results match, they are anagrams.
- Empty strings: after normalization, two empty strings both sort to an empty list, so they are considered anagrams by definition.
- Identical words: normalization keeps the same characters, so comparing sorted forms returns True for the same word.
- Robustness: using normalization before comparison ensures inputs like "Dormitory" vs "Dirty Room!!" are handled correctly without special casing.

PROMPT 04 :

Write a python code that generate at least 3 assert-based tests for an Inventory class with stock management.
Methods:add_item(name, quantity)remove_item(name, quantity)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
```

OUTPUT :

```python
class Inventory:
    def __init__(self):
        self._stock = {}  # name -> quantity

    def add_item(self, name: str, quantity: int) -> None:
        if quantity < 0:
            raise ValueError("quantity must be non-negative")  # defensive check [6]
        self._stock[name] = self._stock.get(name, 0) + quantity

    def remove_item(self, name: str, quantity: int) -> None:
        if quantity < 0:
            raise ValueError("quantity must be non-negative")  # defensive check [6]
        current = self._stock.get(name, 0)
        if quantity > current:
            raise ValueError("insufficient stock")  # prevent negative stock [5]
        self._stock[name] = current - quantity
        if self._stock[name] == 0:
            del self._stock[name]  # optional: clean up zero stock [5]

    def get_stock(self, name: str) -> int:
        return self._stock.get(name, 0)
# Example tests from prompt
inv = Inventory()
inv.add_item("Pen", 10)
assert inv.get_stock("Pen") == 10          # added quantity visible [1]
inv.remove_item("Pen", 5)
assert inv.get_stock("Pen") == 5           # stock decreased correctly [1]
inv.add_item("Book", 3)
assert inv.get_stock("Book") == 3          # separate item tracked [1]

# Additional asserts (edge cases)
inv.add_item("Pen", 0)
assert inv.get_stock("Pen") == 5           # adding zero keeps quantity [6]
try:
    inv.remove_item("Pen", 10)
    assert False, "Expected ValueError for insufficient stock"
except ValueError:
    pass                                   # negative stock prevented [5]

assert inv.get_stock("Unknown") == 0       # missing item defaults to 0 [5]
```

CODE EXPLANATION :

- Data structure: a dictionary maps item names to quantities, so add_item increments the count, remove_item decrements it, and get_stock returns 0 for unknown items to keep lookups simple.
- Input validation: add_item and remove_item reject negative quantities; remove_item also raises an error if the requested removal exceeds current stock, preventing negative inventory.
- Zero cleanup: after removing, if an item's quantity reaches 0, it's deleted from the dictionary to keep the inventory compact and accurate.
- Test coverage: asserts check normal flows (add/get, remove/get), boundary behavior (adding zero, querying unknown items), and error handling (insufficient stock raises an exception).
- Why it's reliable: using a single source of truth (the dictionary) and small, focused methods makes reasoning, debugging, and extending the class straightforward for typical stock

PROMPT 05 :

Write a python code that generate at least 3 assert test cases for validate_and_format_date(date_str) to check and convert dates.Requirements:Validate "MM/DD/YYYY" format.Handle invalid dates.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"

OUTPUT :

```python
from datetime import datetime

def validate_and_format_date(date_str: str) -> str:
    # Enforce exact MM/DD/YYYY with zero-padded month/day
    try:
        dt = datetime.strptime(date_str, "%m/%d/%Y")  # raises ValueError if format/date invalid [4][8]
        return dt.strftime("%Y-%m-%d")               # convert to ISO-style string [3]
    except Exception:
        return "Invalid Date"
from datetime import datetime

def validate_and_format_date(date_str: str) -> str:
    # Enforce exact MM/DD/YYYY with zero-padded month/day
    try:
        dt = datetime.strptime(date_str, "%m/%d/%Y")  # raises ValueError if format/date invalid [4][8]
        return dt.strftime("%Y-%m-%d")               # convert to ISO-style string [3]
    except Exception:
        return "Invalid Date"
```

CODE EXPLANATION :

- Parsing vs formatting: strptime parses a string date into a datetime object using a format like "%m/%d/%Y", while strftime formats that datetime back to a string such as "YYYY-MM-DD".
- Strict validation: datetime.strptime enforces both the exact pattern and calendar correctness, so inputs like "02/30/2023" or "13/01/2023" raise errors and can be reported as "Invalid Date".
- Zero-padding required: the "%m/%d/%Y" pattern expects zero-padded month/day (e.g., "02/05/2023"), so variants like "2/5/2023" should be treated as invalid if strict formatting is desired.
- Conversion step: on successful parse, dt.strftime("%Y-%m-%d") reliably converts the valid date to ISO-like "YYYY-MM-DD".
- Edge behavior: the try/except wrapper cleanly returns "Invalid Date" for any mismatches or impossible dates, keeping the function simple and safe.