**LAB ASSIGNMENT – 8.1**

**NAME :** KARTHIK RAM .D

**HALL.NO :** 2403A52384

**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.

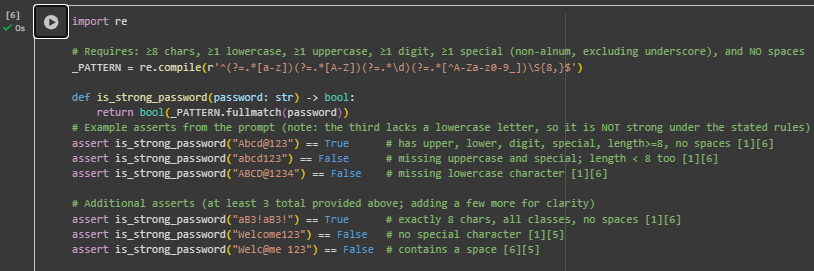
Example Assert Test Cases:

assert is\_strong\_password("Abcd@123") == True

assert is\_strong\_password("abcd123") == False

assert is\_strong\_password("ABCD@1234") == True

**OUTPUT :**



**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 :**



**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).

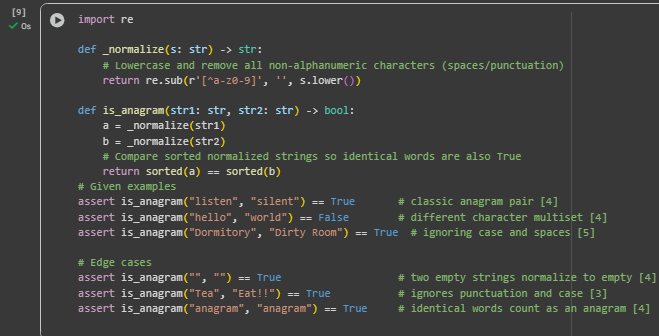
Example Assert Test Cases:

assert is\_anagram("listen", "silent") == True

assert is\_anagram("hello", "world") == False

assert is\_anagram("Dormitory", "Dirty Room") == True

**OUTPUT :**



**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

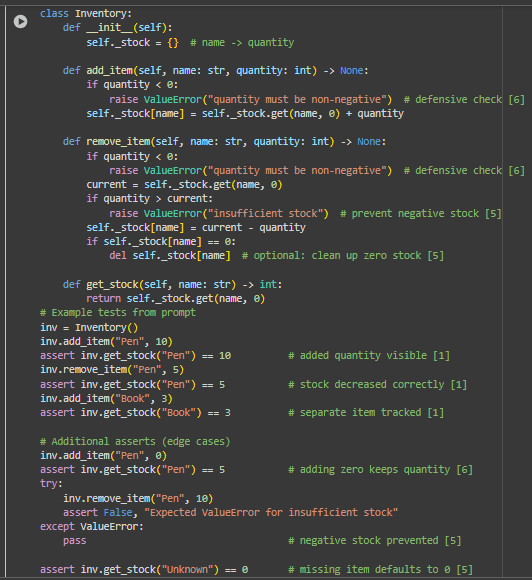
inv.remove\_item("Pen", 5)

assert inv.get\_stock("Pen") == 5

inv.add\_item("Book", 3)

assert inv.get\_stock("Book") == 3

**OUTPUT :**

****

**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".

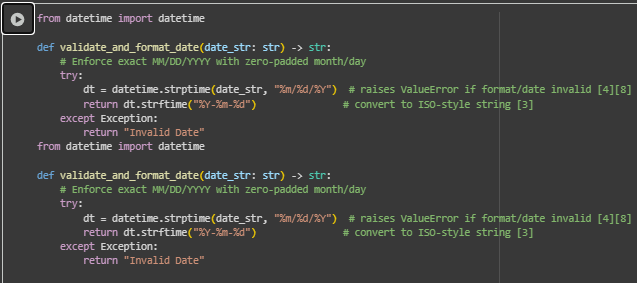
Example Assert Test Cases:

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 :**



**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.