Lab assignment 8.1

# Prompt: generate a password validate code ( is\_strong\_password)that checks whether the passsword is valid or not it contains o Password must have at least 8 characters. o Must include uppercase, lowercase, digit, and special character. o Must not contain spaces

# Code:

# -\*- coding: utf-8 -\*-

"""Password validation helper.

Provides is\_strong\_password(password) -> (bool, list\_of\_reasons).

"""

import re

from typing import List, Tuple

def is\_strong\_password(password: str) -> Tuple[bool, List[str]]:

    """Check whether `password` is strong.

    Rules:

    - At least 8 characters.

    - Includes uppercase, lowercase, digit, and special character.

    - Must not contain any spaces.

    Returns:

        (is\_valid, reasons) where reasons is empty on success.

    """

    reasons: List[str] = []

    if not isinstance(password, str):

        reasons.append("Password must be a string.")

        return False, reasons

    if len(password) < 8:

        reasons.append("Password must have at least 8 characters.")

    if re.search(r"\s", password):

        reasons.append("Password must not contain spaces.")

    if not re.search(r"[A-Z]", password):

        reasons.append("Must include at least one uppercase letter.")

    if not re.search(r"[a-z]", password):

        reasons.append("Must include at least one lowercase letter.")

    if not re.search(r"[0-9]", password):

        reasons.append("Must include at least one digit.")

    # Special character: any character that is not a letter, digit or whitespace

    if not re.search(r"[^A-Za-z0-9\s]", password):

        reasons.append("Must include at least one special character.")

    return (len(reasons) == 0), reasons

if \_\_name\_\_ == "\_\_main\_\_":

    samples = [

        ("Abcdef1!", True),            # valid

        ("short1!", False),           # too short

        ("NoDigit!A", False),         # missing digit

        ("noupper1!", False),         # missing uppercase

        ("NOLOWER1!", False),         # missing lowercase

        ("NoSpecial1A", False),       # missing special

        ("Has Space1!", False),       # contains space

    ]

    for pw, expected in samples:

        ok, reasons = is\_strong\_password(pw)

        status = "PASS" if ok == expected else "FAIL"

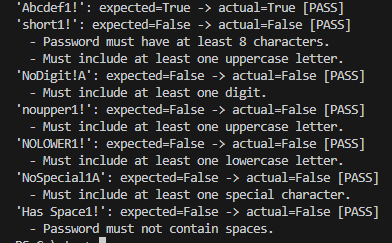
        print(f"{pw!r}: expected={expected} -> actual={ok} [{status}]")

        if reasons:

            for r in reasons:

                print(f"  - {r}")

# output:



# Prompt:

* 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)

# Code:

# -\*- coding: utf-8 -\*-

"""Number classification and simple looped assert tests.

Provides classify\_number(n) which returns one of:

 - 'Positive'

 - 'Negative'

 - 'Zero'

 - 'Invalid input' (for non-number or None)

Includes looped assert tests covering -1, 0, 1 and invalid inputs.

"""

from typing import Any

def classify\_number(n: Any) -> str:

    """Classify n as 'Positive', 'Negative', 'Zero', or 'Invalid input'.

    Rules:

    - Positive: numeric and > 0

    - Negative: numeric and < 0

    - Zero: numeric and == 0

    - Invalid input: None, strings, booleans, or other non-numeric types

    Note: Booleans are rejected as invalid input (since bool is a subclass of int).

    """

    if n is None:

        return "Invalid input"

    # Reject booleans explicitly

    if isinstance(n, bool):

        return "Invalid input"

    # Accept ints and floats

    if isinstance(n, (int, float)):

        if n > 0:

            return "Positive"

        if n < 0:

            return "Negative"

        return "Zero"

    return "Invalid input"

def \_run\_tests() -> None:

    tests = [

        (-1, "Negative"),

        (0, "Zero"),

        (1, "Positive"),

        ("abc", "Invalid input"),

        (None, "Invalid input"),

        (True, "Invalid input"),

    ]

    failures = []

    # Run tests using a loop and assert for each case

    for idx, (inp, expected) in enumerate(tests, start=1):

        result = classify\_number(inp)

        try:

            assert result == expected, f"case {idx}: {inp!r} -> {result} != {expected}"

        except AssertionError as e:

            failures.append(str(e))

    if not failures:

        print(f"All {len(tests)} tests passed.")

    else:

        print(f"{len(failures)} test(s) failed:")

        for f in failures:

            print(" -", f)

if \_\_name\_\_ == "\_\_main\_\_":

    \_run\_tests()

# Output:

All 6 tests passed.

# Prompt:

* 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).

# Code:

def is\_anagram(s1: str, s2: str) -> bool:

    """Return True if s1 and s2 are anagrams.

    Rules:

    - Ignore case, spaces, and punctuation.

    - Empty strings are considered anagrams of each other.

    - Identical words (after normalization) are anagrams.

    """

    def normalize(s: str) -> str:

        # remove non-alphanumeric characters and lower-case

        if s is None:

            return ""

        cleaned = re.sub(r"[^A-Za-z0-9]", "", s)

        return cleaned.lower()

    a = normalize(s1)

    b = normalize(s2)

    # Quick length check

    if len(a) != len(b):

        return False

    # Compare sorted characters (works for unicode alnum too)

    return sorted(a) == sorted(b)

def \_run\_anagram\_tests() -> None:

    tests: List[tuple] = [

        ("listen", "silent", True),                       # basic anagram

        ("Dormitory", "dirty room!", True),              # ignore case/space/punct

        ("hello", "billion", False),                     # not anagram

        ("", "", True),                                  # empty strings

        ("same", "same", True),                          # identical words

    ]

    failures: List[str] = []

    for idx, (s1, s2, expected) in enumerate(tests, start=1):

        res = is\_anagram(s1, s2)

        try:

            assert res == expected, f"anagram case {idx}: {s1!r}, {s2!r} -> {res} != {expected}"

        except AssertionError as e:

            failures.append(str(e))

    if not failures:

        print(f"All {len(tests)} anagram tests passed.")

    else:

        print(f"{len(failures)} anagram test(s) failed:")

        for f in failures:

            print(" -", f)

if \_\_name\_\_ == "\_\_main\_\_":

    \_run\_tests()

    \_run\_anagram\_tests()

# Output:

All 5 anagram tests passed.

# Prompt:

* 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)

# Code:

class Inventory:

    """Simple inventory with stock management.

    Methods:

    - add\_item(name, quantity): add quantity (int>0) to item stock

    - remove\_item(name, quantity): remove quantity; raises ValueError if insufficient stock or invalid input

    - get\_stock(name): return current stock (int, 0 if missing)

    """

    def \_\_init\_\_(self) -> None:

        self.\_stocks = {}

    def add\_item(self, name: str, quantity: int) -> None:

        if not isinstance(name, str) or not name:

            raise ValueError("Invalid item name")

        if not isinstance(quantity, int) or quantity <= 0:

            raise ValueError("Quantity must be a positive integer")

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

    def remove\_item(self, name: str, quantity: int) -> None:

        if not isinstance(name, str) or not name:

            raise ValueError("Invalid item name")

        if not isinstance(quantity, int) or quantity <= 0:

            raise ValueError("Quantity must be a positive integer")

        current = self.\_stocks.get(name, 0)

        if quantity > current:

            raise ValueError("Insufficient stock")

        self.\_stocks[name] = current - quantity

        if self.\_stocks[name] == 0:

            del self.\_stocks[name]

    def get\_stock(self, name: str) -> int:

        if not isinstance(name, str) or not name:

            return 0

        return self.\_stocks.get(name, 0)

def \_run\_inventory\_tests() -> None:

    inv = Inventory()

    tests = []

    # Test 1: add and check stock

    def t1() -> bool:

        inv.add\_item("apple", 10)

        return inv.get\_stock("apple") == 10

    tests.append((t1, True, "add\_item then get\_stock"))

    # Test 2: remove and check stock

    def t2() -> bool:

        inv.remove\_item("apple", 4)

        return inv.get\_stock("apple") == 6

    tests.append((t2, True, "remove\_item reduces stock"))

    # Test 3: removing more than stock raises ValueError

    def t3() -> bool:

        try:

            inv.remove\_item("apple", 100)

        except ValueError:

            return True

        return False

    tests.append((t3, True, "remove more than stock raises"))

    # Test 4: get\_stock for unknown item returns 0

    def t4() -> bool:

        return inv.get\_stock("banana") == 0

    tests.append((t4, True, "get\_stock unknown item -> 0"))

    failures: List[str] = []

    for idx, (fn, expected, desc) in enumerate(tests, start=1):

        try:

            res = fn()

            assert res == expected, f"inventory case {idx} ({desc}) -> {res} != {expected}"

        except AssertionError as e:

            failures.append(str(e))

    if not failures:

        print(f"All {len(tests)} inventory tests passed.")

    else:

        print(f"{len(failures)} inventory test(s) failed:")

        for f in failures:

            print(" -", f)

if \_\_name\_\_ == "\_\_main\_\_":

    \_run\_tests()

    \_run\_anagram\_tests()

    \_run\_inventory\_tests()

# Output:

All 4 inventory tests passed.

# Prompt:

* to 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".

# Code:

def validate\_and\_format\_date(date\_str: Optional[str]) -> Optional[str]:

    """Validate a date string in MM/DD/YYYY and return YYYY-MM-DD or None if invalid.

    Rules:

    - Accepts exactly MM/DD/YYYY where MM is 01-12, DD is valid for the month/year, YYYY is four digits

    - Returns ISO-style 'YYYY-MM-DD' on success, or None on failure

    """

    if not isinstance(date\_str, str):

        return None

    # Basic format check

    m = re.fullmatch(r"(0[1-9]|1[0-2])/(0[1-9]|[12][0-9]|3[01])/([0-9]{4})", date\_str)

    if not m:

        return None

    mm, dd, yyyy = m.group(1), m.group(2), m.group(3)

    try:

        dt = datetime.date(int(yyyy), int(mm), int(dd))

    except ValueError:

        return None

    return dt.isoformat()

def \_run\_date\_tests() -> None:

    tests = [

        ("02/28/2020", "2020-02-28"),  # valid leap-year-adjacent

        ("02/29/2020", "2020-02-29"),  # valid leap day

        ("02/29/2019", None),           # invalid leap day

        ("13/01/2020", None),           # invalid month

        ("04/31/2021", None),           # invalid day for April

        ("1/1/2020", None),             # invalid format (missing leading zeros)

        (None, None),                    # invalid type

    ]

    failures: List[str] = []

    for idx, (inp, expected) in enumerate(tests, start=1):

        res = validate\_and\_format\_date(inp)

        try:

            assert res == expected, f"date case {idx}: {inp!r} -> {res} != {expected}"

        except AssertionError as e:

            failures.append(str(e))

    if not failures:

        print(f"All {len(tests)} date tests passed.")

    else:

        print(f"{len(failures)} date test(s) failed:")

        for f in failures:

            print(" -", f)