AI ASSISTED CODING – 8.1

# NAME : N.KEERTHIREDDY

# HALL NO. : 2403A52376

# BATCH : AI 14

TASK - 1

CODE :

import re

def is\_strong\_password(password: str) -> bool:

if len(password) < 8:

return False

if ' ' in password:

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'[^A-Za-z0-9]', password):

return False

return True

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

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

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

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

assert is\_strong\_password("Abc d@123") == False

print("Password validation logic passing all AI-generated test cases.")

## OUTPUT :

A screen shot of a computer code

AI-generated content may be incorrect.

## EXPLANATION :

This function checks if a given password string meets a set of criteria for being considered "strong".

* **if len(password) < 8:**: This checks if the password has at least 8 characters. If it's shorter, it immediately returns False.
* **if ' ' in password:**: This checks if the password contains any spaces. If it does, it returns False.
* **if not any(c.islower() for c in password):**: This uses a generator expression and the any() function to check if there is at least one lowercase character in the password. If there are no lowercase characters, it returns False.
* **if not any(c.isupper() for c in password):**: Similar to the previous check, this verifies if there is at least one uppercase character. If not, it returns False.
* **if not any(c.isdigit() for c in password):**: This checks for the presence of at least one digit in the password. If no digits are found, it returns False.
* **if not any(not c.isalnum() for c in password):**: This is a clever way to check for special characters. c.isalnum() is true for alphanumeric characters (letters and digits). not c.isalnum() is true for characters that are *not* alphanumeric, which are special characters. any() then checks if there's at least one such character. If there are no special characters, it returns False.
* **return True**: If the password passes all the above checks, it means it meets all the criteria for a strong password, and the function returns True.

TASK - 2

## CODE :

def classify\_number(n):

for item in [n]:

if not isinstance(item, (int, float)):

return "Invalid input"

if item > 0:

return "Positive"

elif item < 0:

return "Negative"

else:

return "Zero"

assert classify\_number(10) == "Positive"

assert classify\_number(-5) == "Negative"

assert classify\_number(0) == "Zero"

assert classify\_number("abc") == "Invalid input"

assert classify\_number(None) == "Invalid input"

assert classify\_number(-1) == "Negative"

assert classify\_number(1) == "Positive"

print("# --- Number Classification Function ---")

def classify\_number(n):

for item in [n]:

if not isinstance(item, (int, float)):

return "Invalid input"

if item > 0:

return "Positive"

elif item < 0:

return "Negative"

else:

return "Zero"

assert classify\_number(10) == "Positive"

assert classify\_number(-5) == "Negative"

assert classify\_number(0) == "Zero"

assert classify\_number("abc") == "Invalid input"

assert classify\_number(None) == "Invalid input"

assert classify\_number(-1) == "Negative"

assert classify\_number(1) == "Positive"

print("Classification logic passing all assert tests")

## OUTPUT :

A black screen with white text

AI-generated content may be incorrect.

## EXPLANATION :

The main goal of Task 2 is to create a Python function called classify\_number that can determine if a given input is a positive number, a negative number, or zero.

Here are the specific requirements for this function:

1. Classification: The function needs to correctly identify if a number is positive (greater than 0), negative (less than 0), or exactly zero.

2. Handling Invalid Input: The function must be able to handle inputs that are not numbers, such as text strings or the value None, and return a specific indicator for these invalid inputs.

3. Using Loops: A specific requirement for this task was to implement the classification logic using loops. While a simple if-elif-else structure is more common and efficient for this type of classification in Python, the task specifically requested the use of loops. The provided code fulfills this by using a loop that runs only once within each conditional branch (if n > 0, elif n < 0, else).

4. Boundary Conditions: The test cases should include boundary conditions, which are values at the edges of the classification criteria. For this task, the boundary conditions are -1, 0, and 1. The code includes test cases for these values to ensure the function handles them correctly.

5. AI-Generated Test Cases: The task also required using AI to generate at least three assert test cases. The provided code includes a set of assert statements that serve as these test cases, covering positive, negative, and zero numbers, as well as invalid inputs and boundary conditions.

The expected output for this task is simply that the classification logic, when tested with the assert statements, passes all the tests. This is confirmed by the output "All test cases passed!".

In summary, Task 2 was about creating a function that classifies numbers and handles non-numeric inputs, with the specific constraint of using loops for the classification part, and verifying its correctness with AI-generated test cases that cover various scenarios, including boundaries and invalid inputs.

TASK – 3

## CODE :

import re

def is\_anagram(str1, str2):

def clean\_string(s):

return sorted(re.sub(r'[^a-zA-Z0-9]', '', s).lower())

return clean\_string(str1) == clean\_string(str2)

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

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

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

assert is\_anagram("The eyes!", "They see.") == True

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

print("import re")

def is\_anagram(str1, str2):

def clean\_string(s):

return sorted(re.sub(r'[^a-zA-Z0-9]', '', s).lower())

return clean\_string(str1) == clean\_string(str2)

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

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

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

assert is\_anagram("The eyes!", "They see.") == True

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

print("Function correctly identifying anagrams and passing all AI-generated tests")

## OUTPUT :

A black screen with white text

AI-generated content may be incorrect.

## EXPLANATION :

The provided code attempts to meet these requirements by:

• Defining a helper function clean\_string that removes spaces and punctuation and converts the string to lowercase. This addresses the requirement to ignore case, spaces, and punctuation.

• Comparing the lengths of the cleaned strings. If the lengths are different, they cannot be anagrams.

• Sorting the characters of the cleaned strings and comparing the sorted results. If the sorted strings are identical, the original strings are anagrams.

• Including several assert statements as test cases, covering the requirements for ignoring case/spaces/punctuation and handling edge cases.

Currently, there is an issue with the code where the last test case (assert is\_anagram("The eyes", "They see") == False) is failing. This indicates that the function is incorrectly identifying "The eyes" and "They see" as anagrams when they are not. I am looking into the code to figure out why this is happening.

TASK – 4

## CODE :

class Inventory:

def \_init\_(self):

self.stock = {}

def add\_item(self, name, quantity):

if quantity <= 0:

return

if name in self.stock:

self.stock[name] += quantity

else:

self.stock[name] = quantity

def remove\_item(self, name, quantity):

if quantity <= 0:

return

if name in self.stock:

self.stock[name] = max(self.stock[name] - quantity, 0)

def get\_stock(self, name):

return self.stock.get(name, 0)

# --- 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

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

assert inv.get\_stock("Book") == 0

assert inv.get\_stock("Notebook") == 0

inv.add\_item("Pen", 0)

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

print("Fully functional class passing all assertions.")

## OUTPUT :

A black screen with white text

AI-generated content may be incorrect.

## EXPLANATION :

The main goal of Task 4 was to create a Python class called Inventory that simulates a simple inventory management system. This class should be able to keep track of different items and their quantities in stock.

Here are the specific requirements for the Inventory class:

1. add\_item(name, quantity) method: This method should allow you to add a certain quantity of an item specified by its name to the inventory. If the item already exists, its stock should be increased. If it's a new item, it should be added to the inventory with the given quantity. The code includes a check to ensure that the quantity to add is positive.

2. remove\_item(name, quantity) method: This method should allow you to remove a certain quantity of an item specified by its name from the inventory. It needs to handle cases where the item doesn't exist, where the quantity to remove is more than what's in stock, and where the quantity to remove is zero or negative. If removing the quantity results in the stock becoming zero, the item should be removed from the inventory. The code includes checks for these scenarios and prints informative messages.

3. get\_stock(name) method: This method should return the current stock level of an item specified by its name. If the item is not found in the inventory, it should return 0. The code uses the dict.get() method with a default value of 0 to handle items not in stock efficiently.

4. AI-Generated Test Cases: The task required using AI to generate at least three assert-based tests to verify the functionality of the Inventory class and its methods. The provided code includes a comprehensive set of assert statements that test adding items, removing items (including removing all of an item and attempting to remove more than available), checking stock levels, handling non-existent items, and handling zero or negative quantities for adding/removing.

5. The expected output for this task is that the Inventory class is fully functional and all the AI-generated assert tests pass, confirming that the methods work as intended. The output "All assert tests passed!" indicates that the code successfully meets these requirements.

TASK – 5

## CODE :

from datetime import datetime

def validate\_and\_format\_date(date\_str):

"""

Validates a date string in "MM/DD/YYYY" format and converts it to "YYYY-MM-DD".

Args:

date\_str (str): The date string to validate and format.

Returns:

str: The date in "YYYY-MM-DD" format if valid, otherwise "Invalid Date".

"""

try:

# Attempt to parse the date string in MM/DD/YYYY format

date\_obj = datetime.strptime(date\_str, "%m/%d/%Y")

# If parsing is successful, format it to YYYY-MM-DD

return date\_obj.strftime("%Y-%m-%d")

except ValueError:

# If parsing fails (invalid format or invalid date), return "Invalid Date"

return "Invalid Date"

# AI-generated test cases

assert validate\_and\_format\_date("10/15/2023") == "2023-10-15"

assert validate\_and\_format\_date("02/30/2023") == "Invalid Date" # Invalid day

assert validate\_and\_format\_date("01/01/2024") == "2024-01-01"

assert validate\_and\_format\_date("12/31/2023") == "2023-12-31" # End of year

assert validate\_and\_format\_date("01/00/2023") == "Invalid Date" # Invalid day (zero)

assert validate\_and\_format\_date("13/01/2023") == "Invalid Date" # Invalid month

assert validate\_and\_format\_date("10-15-2023") == "Invalid Date" # Invalid format

assert validate\_and\_format\_date("10/15/23") == "Invalid Date" # Invalid year format

assert validate\_and\_format\_date("abc") == "Invalid Date" # Non-date string

assert validate\_and\_format\_date("") == "Invalid Date" # Empty string

print("• Function passes all AI-generated assertions and handles edge cases.")

## OUTPUT :

A black background with white text

AI-generated content may be incorrect.

## EXPLANATION :

**from datetime import datetime**

* This imports the datetime class from Python's built-in datetime module.
* Used for **parsing** and **formatting** dates.

**🔹 try: ... except ValueError:**

* Wraps the date parsing logic in a try block.
* If the date string is invalid (wrong format or non-existent date), it raises a ValueError.

**🔹 datetime.strptime(date\_str, "%m/%d/%Y")**

* This checks two things:
  1. **Is the format correct?** It must be in MM/DD/YYYY format.
  2. **Is the actual date valid?** (e.g., Feb 30 is invalid)

If both conditions are met, it returns a datetime object.

**🔹 parsed\_date.strftime("%Y-%m-%d")**

* Converts the valid datetime object into a string in YYYY-MM-DD format as required.

**🔹 return "Invalid date"**

* If the parsing fails (format is wrong or the date is impossible), this message is returned