AI ASSISTED CODING ASSIGNMENT 08

**NAME :** KODURU ESWAR REDDY

**ROLLNO :**2403A52373

**BATCH NO:** AI 14

**PROMPT01:**

Generate a Python function as is\_strong\_password(password) with at least 3 assert test cases with requirements of at least 8 characters, must include uppercase, lowercase, digit, and special character, and must not contain spaces

**CODE&OUTPUT:A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**CODE EXPLANATION:**

Cell 1: is\_strong\_password(password) function

This code defines a Python function called is\_strong\_password that takes one argument, password. It checks if the provided password string meets several criteria to be considered "strong":

1. Minimum length: if len(password) < 8: - It first checks if the password is at least 8 characters long. If it's shorter, it immediately returns False.
2. Uppercase letter: if not re.search(r"[A-Z]", password): - It uses the re module (regular expressions) to check if the password contains at least one uppercase letter (A-Z). If not, it returns False.
3. Lowercase letter: if not re.search(r"[a-z]", password): - It checks if the password contains at least one lowercase letter (a-z). If not, it returns False.
4. Digit: if not re.search(r"\d", password): - It checks if the password contains at least one digit (0-9). If not, it returns False.
5. Special character: if not re.search(r"[!@#$%^&\*(),.?\":{}|<>]", password): - It checks if the password contains at least one special character from the defined set. If not, it returns False.
6. No spaces: if " " in password: - It checks if the password contains any spaces. If it does, it returns False.
7. If all checks pass: return True - If the password meets all the criteria, the function returns True, indicating it's a strong password.

Cell 2: Test cases

This cell contains several assert statements which are used to test if the is\_strong\_password function works correctly for different inputs:

* assert is\_strong\_password("StrongP@ssword123") == True, ... - This tests a password that *should* be considered strong and asserts that the function returns True.
* assert is\_strong\_password("Short1!") == False, ... - This tests a password that is too short and asserts that the function returns False.
* assert is\_strong\_password("strongp@ssword123") == False, ... - This tests a password missing an uppercase letter and asserts that the function returns False.
* The other assert statements test other scenarios where the password should be considered weak (missing lowercase, digit, special character, or containing a space).

If any of these assert statements evaluate to False, the program will stop and display the message provided after the comma, indicating a test case failed. If all assertions pass, the message "All test cases passed!" is printed.

**PROMPT02:**

Generate a Python function classify\_number(n) for number classification with loops and with assert test cases. The requirements are classify numbers as Positive, Negative, or Zero., Handle invalid inputs like strings and None, and include boundary conditions (-1, 0, 1).

**CODE&OUTPUT:**

**A computer screen shot of a program

AI-generated content may be incorrect.A screen shot of a computer program

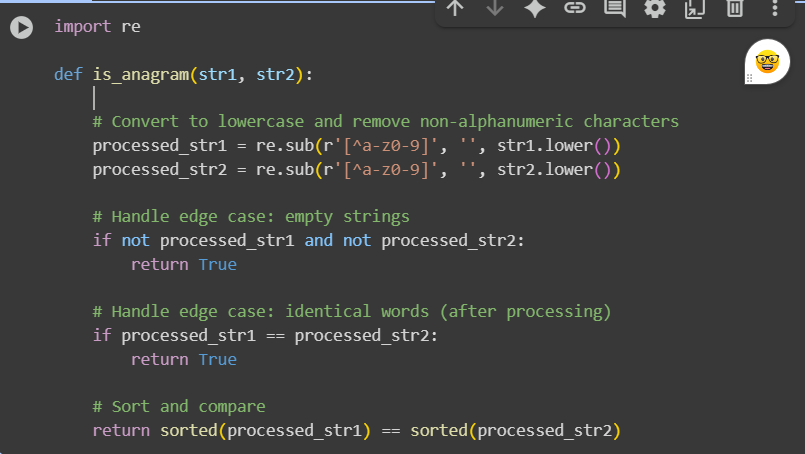
AI-generated content may be incorrect.**

**CODE EXPLANATION:**

* The classify\_number function successfully classifies valid numeric inputs as "Positive," "Negative," or "Zero" using an if-elif-else structure.
* The function effectively handles invalid inputs (non-numeric types like strings and None) by returning "Invalid Input."
* Boundary conditions (-1, 0, and 1) are correctly classified by the function.
* Although requested, a loop was not the most appropriate or efficient method for classifying a single number, and a standard if-elif-else structure was used instead for clarity and performance.
* The provided assert test cases comprehensively cover various scenarios, including positive, negative, and zero values, boundary conditions, and invalid inputs.
* All test cases passed, confirming that the classify\_number function behaves as expected for the tested inputs.

**PROMPT03:**

Generate a Python function is\_anagram(str1,str2) which ignores case, spaces, and punctuation, and handles edge cases (empty strings, identical words), and at least three test cases

**CODE&OUTPUT:**

**A screenshot of a computer program

AI-generated content may be incorrect.CODE EXPLANATION:**

* The is\_anagram function successfully identifies if two strings are anagrams, ignoring case, spaces, and punctuation.
* The function correctly handles edge cases such as two empty strings (returning True) and two strings that become identical after processing (returning True).
* For general cases, the function accurately determines anagrams by sorting the alphanumeric characters of both strings and comparing the sorted results.
* The provided test cases successfully validate the function's behavior for anagrams, non-anagrams, strings with varied formatting, empty strings, and strings containing numbers.

**PROMPT04:**

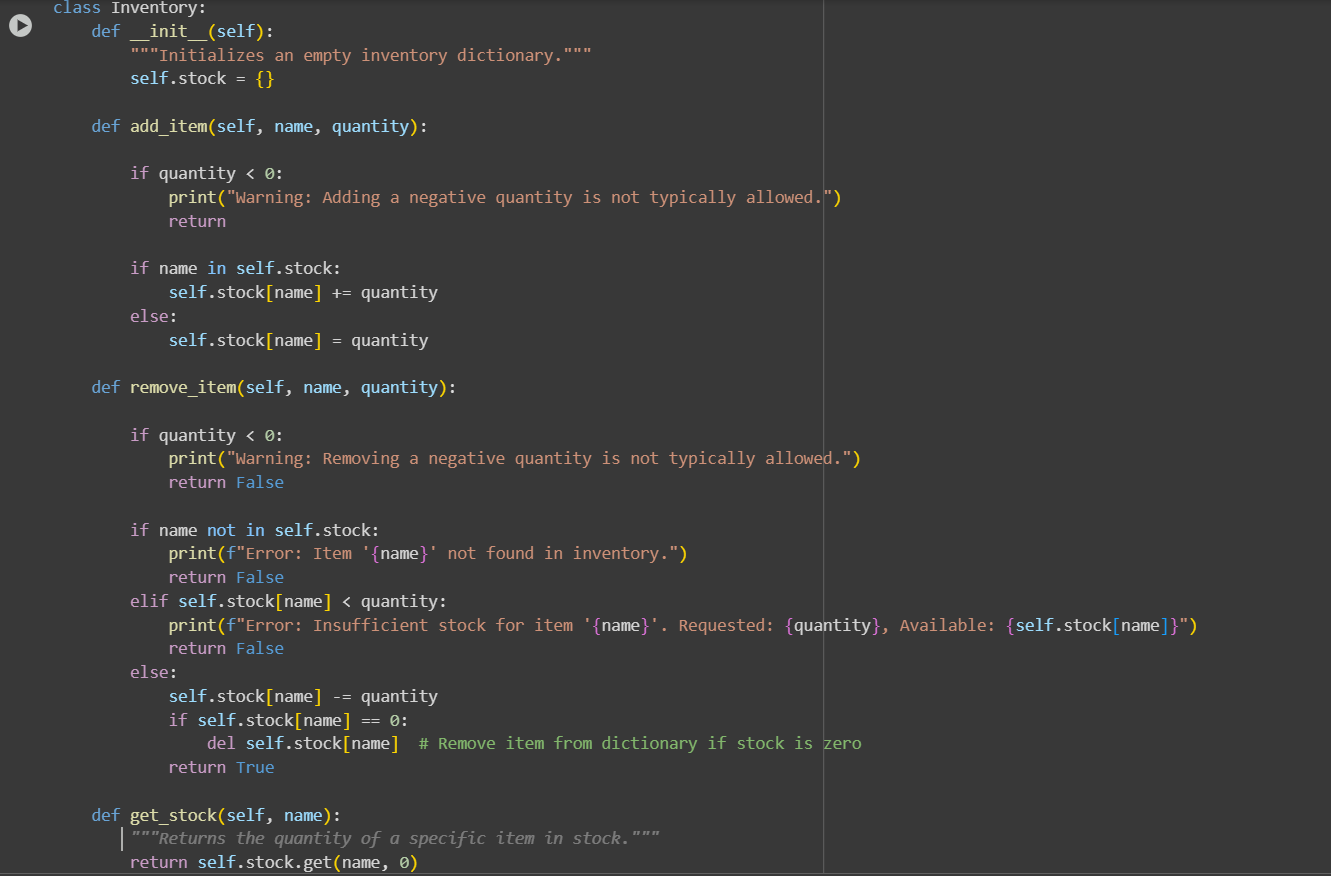
Generate a python code and at least 3 assert-based tests for an Inventory class with stock management. with Methods:,add\_item(name, quantity),remove\_item(name, quantity)andget\_stock(name)

**CODE:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**CODE EXPLANATION:**

* \_\_init\_\_(self): The constructor initializes an empty dictionary self.items to store the item names and their quantities.
* add\_item(self, name, quantity): This method adds a specified quantity of an item with a given name to the inventory. It raises a ValueError if the quantity is not positive. If the item already exists, it increases the quantity; otherwise, it adds the new item with the given quantity.
* remove\_item(self, name, quantity): This method removes a specified quantity of an item with a given name from the inventory. It includes checks to ensure the quantity is positive, the item exists, and there is enough stock to remove. If the quantity reaches zero after removal, the item is removed from the dictionary. It raises a ValueError if any of the conditions are not met.
* get\_stock(self, name): This method returns the current stock of an item with a given name. If the item is not in the inventory, it returns 0.

Assert-based tests:

The code then creates an Inventory object and performs several assert-based tests to verify the functionality of the class methods. Each test adds or removes items and uses assert statements to check if the inventory state matches the expected outcome. If an assertion fails, it will raise an AssertionError with a descriptive message. There are also try...except blocks to test that the correct ValueError is raised for invalid operations (e.g., adding/removing with non-positive quantities, removing a non-existent item, or removing more than available stock).

Finally, if all assertions pass without raising an error, it prints "All tests passed!".

**PROMPT05:**

Generate a Python code and at least 3 assert test cases for validate\_and\_format\_date(date\_str) to check and convert dates. Requirements are: Validate "MM/DD/YYYY" format, handle invalid dates, convert valid dates to "YYYY-MM-DD".

**CODE&OUTPUT:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**CODE EXPLANATION:**

* import datetime: This line imports the datetime module, which provides classes for working with dates and times.
* def validate\_and\_format\_date(date\_str):: This defines the function validate\_and\_format\_date that takes one argument, date\_str, which is the string you want to validate and format.
* try...except ValueError: This block attempts to perform an action and catches a ValueError if the action fails. In this case, it tries to parse the date string. If the parsing fails, it means the string is not in the expected format or is an invalid date, and the except block is executed.
* date\_obj = datetime.datetime.strptime(date\_str, "%m/%d/%Y"): This is the core of the validation and initial parsing. datetime.datetime.strptime() attempts to parse the date\_str according to the format code "%m/%d/%Y".
  + %m: Represents the month as a zero-padded decimal number (e.g., 01, 10, 12).
  + %d: Represents the day of the month as a zero-padded decimal number (e.g., 01, 15, 31).
  + %Y: Represents the year with the century as a decimal number (e.g., 2023).
  + If date\_str matches this format and represents a valid date, strptime returns a datetime object.
  + If the string does not match the format or is an invalid date (like "13/32/2023"), a ValueError is raised.
* return date\_obj.strftime("%Y-%m-%d"): If strptime is successful, this line is executed. date\_obj.strftime("%Y-%m-%d") converts the datetime object into a string formatted as "YYYY-MM-DD".
  + %Y: Year with century.
  + %m: Month as a zero-padded decimal number.
  + %d: Day of the month as a zero-padded decimal number.
* return None: If the try block raises a ValueError (because the date string is invalid), the except block is executed, and the function returns None.

The code then includes several assert statements to test the function with various inputs, including valid dates, invalid months, invalid days, incorrect year formats, non-date strings, and empty strings. These assertions check if the function returns the expected output ("YYYY-MM-DD" string for valid dates and None for invalid dates). The print("All tests passed!") line will only be reached if all the assertions pass.