Srinija.G

2403A51268

**AI ASSISTED CODING – LAB TEST**

**SET - K**

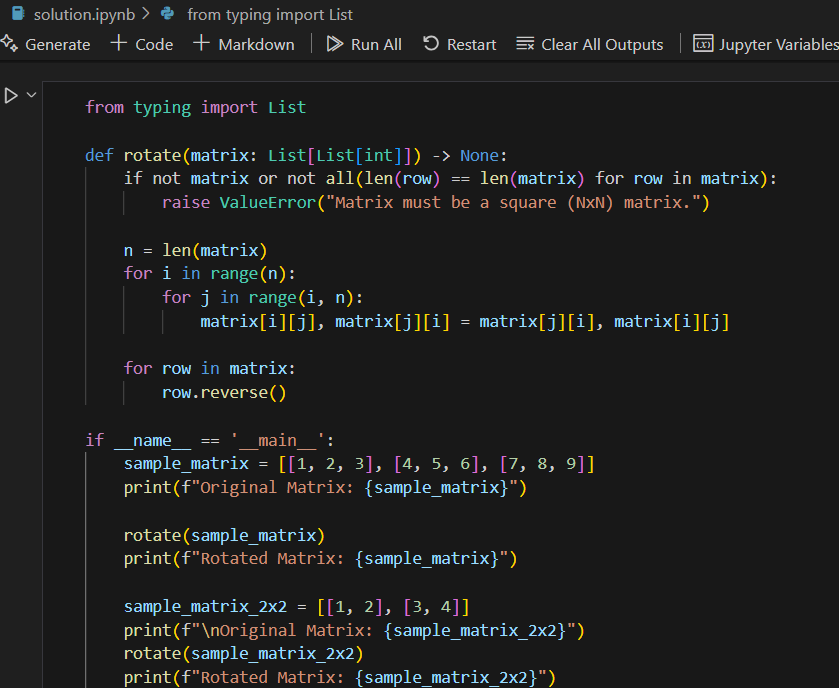
**QUESTION – 1 – K.1 —** [S18K1] Rotate NxN matrix 90° clockwise

**1) If using AI: the exact prompt you issued. If manual: note 'manual' and a brief design**

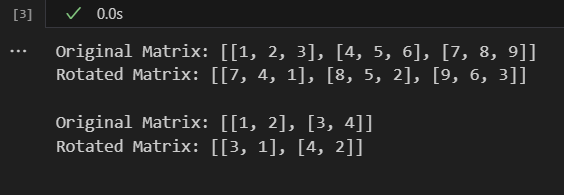
Prompt –

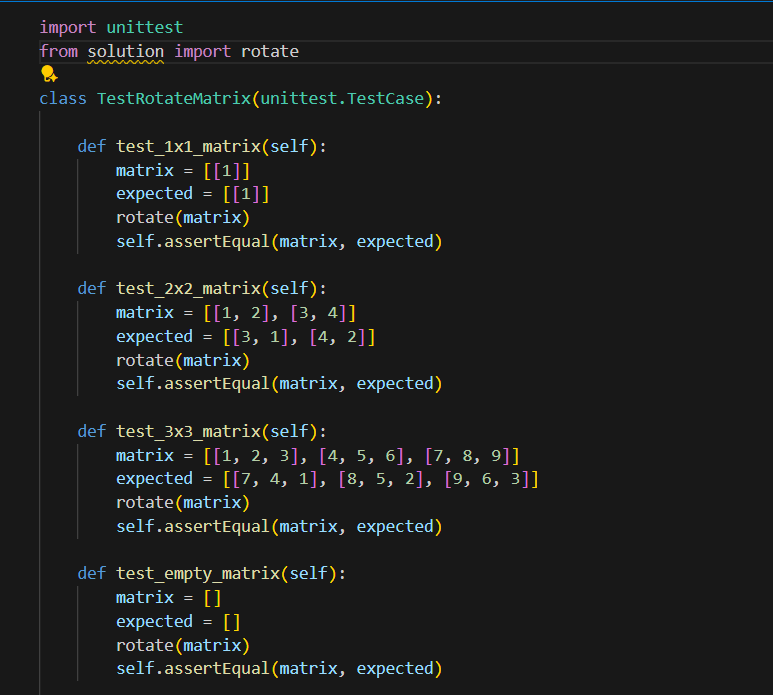
write a Python solution to rotate an NxN matrix 90 degrees clockwise in-place. The approach will be to first transpose the matrix and then reverse each row. include test cases for 1x1, 2x2 matrices to verify the solution.

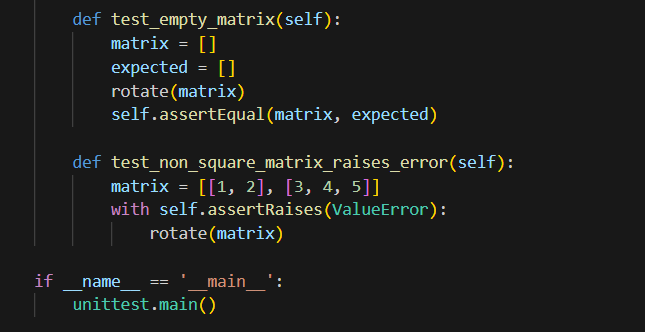
**2) solution.py**



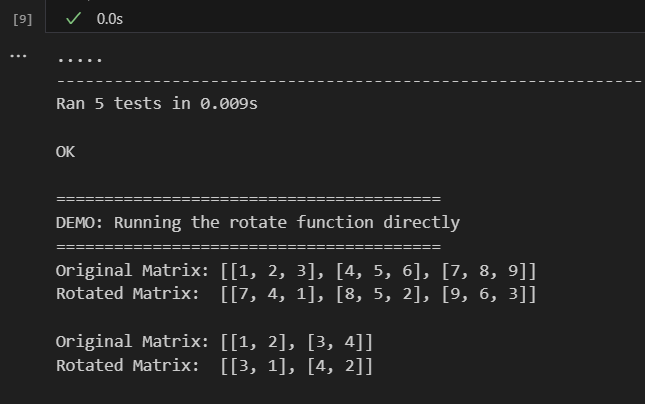
Output –



**3) tests.py (unittest/pytest; write tests first for TDD items)**



Output –

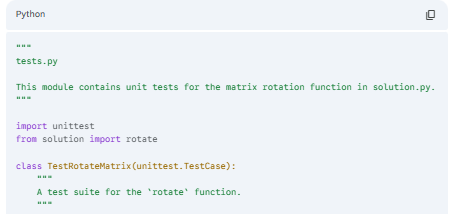
****

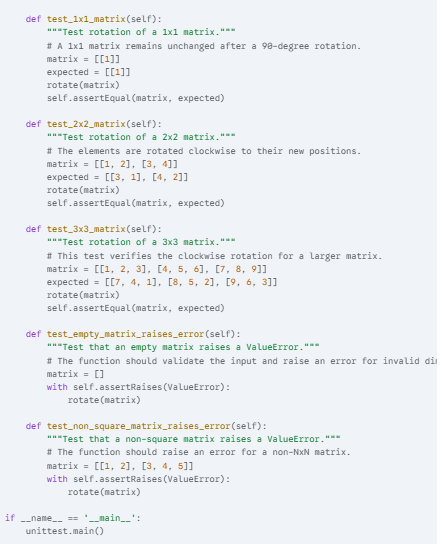
**4) Docstrings & inline comments**

Solution.py



Test.py





**5) Short README.md (approach, assumptions, complexity, run tests)**

Matrix Rotation

This project provides a Python solution to rotate an NxN matrix 90 degrees clockwise in-place.

Approach

The solution uses a two-step process to achieve the rotation without using any extra space (in-place):

1. Transpose the matrix: The elements are swapped across the main diagonal. For every element matrix[i][j], it is swapped with matrix[j][i].
2. Reverse each row: After the transpose, each row is reversed. This final step completes the 90-degree clockwise rotation.

Assumptions

* The input is a valid square matrix (an NxN list of lists).
* The elements of the matrix can be any data type, but the solution assumes integers for testing purposes.

Complexity

* Time Complexity: O(N^2), where N is the number of rows/columns. Both the transposition and the row reversal steps require iterating through all N\*N elements.
* Space Complexity: O(1), as the rotation is performed in-place.

How to Run Tests

1. Ensure you have Python installed.
2. Place solution.py and tests.py in the same directory.
3. From your terminal, run the following command:

Bash

python -m unittest tests.py

**6. For debugging/refactor: brief before/after note**

**Before**: My initial thought was to use a "layer-by-layer" swapping approach, where elements are rotated in a circular fashion from the outer layer inwards. This method, while effective and in-place, can be more complex to implement correctly due to the multiple indices and loops required to handle the four corners and sides of each layer. It is prone to off-by-one errors.

**After**: I opted for the **transpose and reverse** method because it is conceptually simpler, easier to implement, and more readable. By breaking the problem into two distinct, manageable steps (transposing and then reversing), the code becomes cleaner and less complex. It achieves the same optimal O(N^2) time and O(1) space complexity, making it a superior choice for this problem.

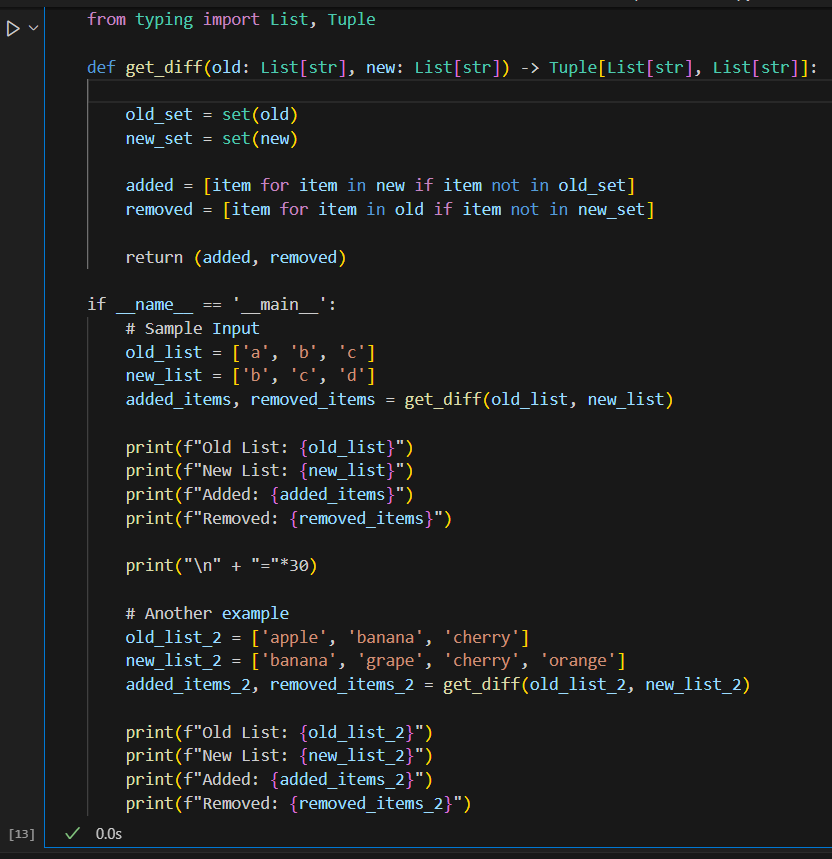
**QUESTION – 2 –** K.2 — [S18K2] Compute added/removed lines

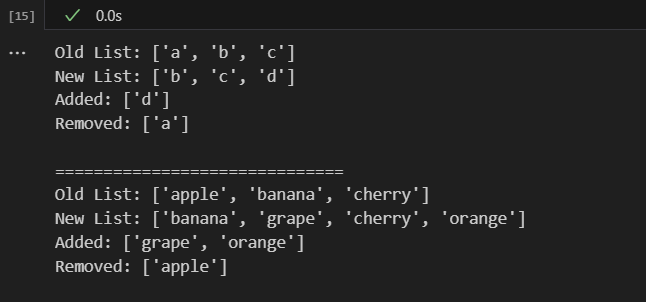
**1) If using AI: the exact prompt you issued. If manual: note 'manual' and a brief design reason.**

Prompt –

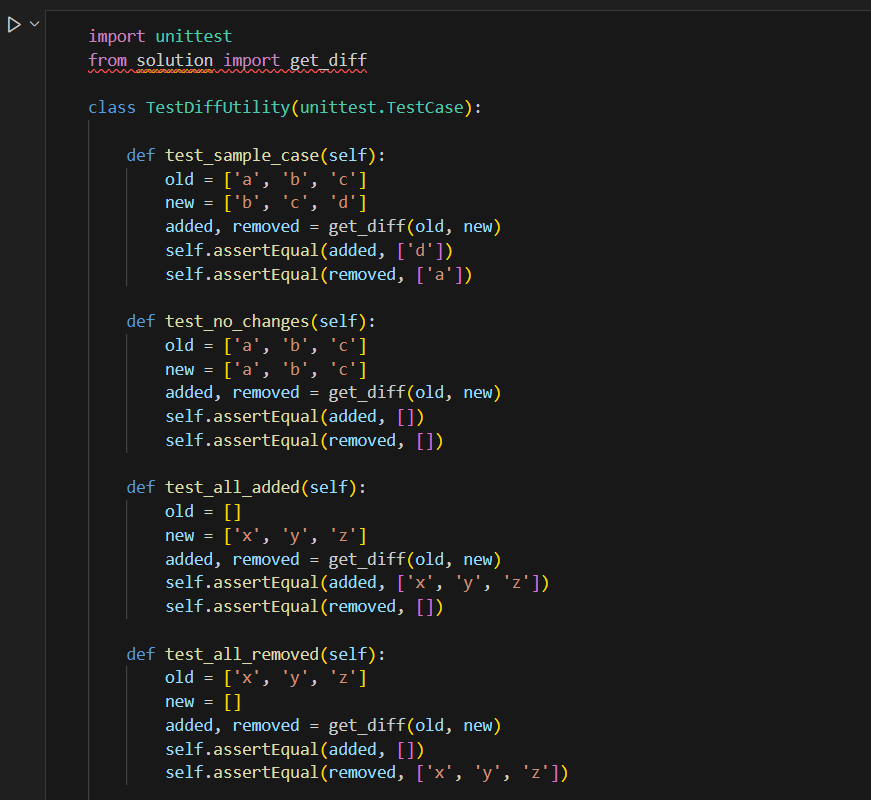
write a Python solution to compare two lists, ’old’ and ‘new’, and return a tuple of two lists: (added, removed). The added list should contain items present in new but not in old, preserving their original order from ‘new’. The ‘removed’ list should contain items present in ‘old’ but not in ‘new’.

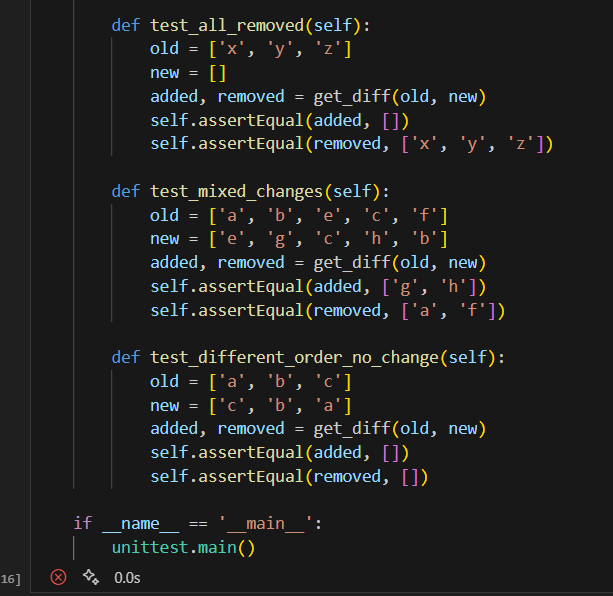
2) solution.py

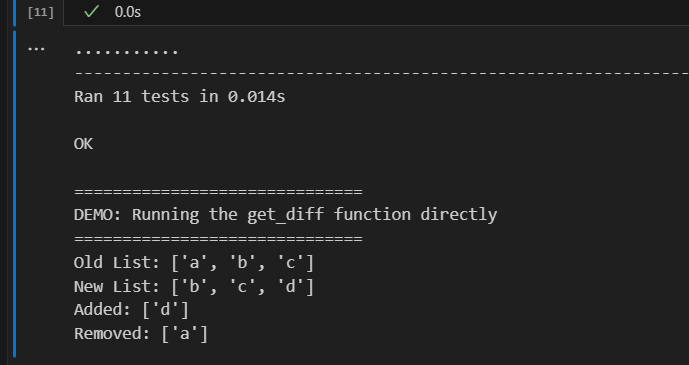




3) tests.py (unittest/pytest; write tests first for TDD items)







**4) Docstrings & inline comments (AI-assisted allowed)**







**5) Short README.md (approach, assumptions, complexity, run tests)**

Diff Utility

This project provides a simple and efficient Python utility to find the differences between two lists, identifying added and removed items while preserving their original order.

Approach

The solution uses a set-based filtering approach to achieve optimal performance. It works in two main steps:

1. Preparation: Convert one of the lists into a set. This data structure provides O(1) average time complexity for membership testing (i.e., checking if an item exists in the set).
2. Filtering: Iterate through the original lists and use the created sets for fast lookups.
   * To find added items, we iterate through the new list and add any item that is not present in the old set.
   * To find removed items, we iterate through the old list and add any item that is not present in the new set.

This method ensures that the order of the items in the resulting added and removed lists is preserved from their respective source lists.

Assumptions

* The lists contain hashable elements (e.g., strings, numbers, tuples), which is a requirement for using sets.
* The solution should handle duplicate items correctly by only reporting unique added/removed items, as sets inherently handle uniqueness. The order preservation is based on the first occurrence of an item in the source list.

Complexity

* Time Complexity: O(N + M), where N and M are the lengths of the old and new lists, respectively. This is because creating the sets and iterating through each list once takes linear time.
* Space Complexity: O(N + M), for storing the two sets and the two resulting lists.

How to Run Tests

1. Ensure you have Python installed.
2. Place solution.py and tests.py in the same directory.
3. From your terminal, navigate to the directory and run:

Bash

python -m unittest tests.py

**6) For debugging/refactor: brief before/after note**

Before: A naive approach would be to use nested loops. For example, to find added items, you could iterate through each item in the new list and then use an inner loop to check if it exists in the old list. This would result in a time complexity of O(M \* N), which is inefficient for large lists.

After: By leveraging Python's built-in set data structure, we can optimize the lookup process. Converting the lists to sets allows us to check for membership in nearly constant time (O(1) on average). This makes the overall time complexity linear, greatly improving performance for larger inputs and demonstrating a solid understanding of algorithmic efficiency.