Permutation

May 18, 2020

1 Permutation

Question - Let's use recursion to help us solve the following permutation problem:

Given a list of items, the goal is to find all of the permutations of that list. For example, Given a list like: `[0, 1, 2]` Permutations: `[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]` Notice that the expected output is a list of permutation with each permuted item being represented by a list. Such an object that contains other object is called "compound" object.

The Idea Build a compoundList incrementally starting with a blank list, and permute (add) each element of original input list at all possible positions.

For example, take `[0, 1, 2]` as the original input list:

1. Start with a blank compoundList `[[]]`. This is actually the last call of recursive function stack. Pick the an element `2` of original input list, making the compoundList as `[[2]]`

2. Pick next element `1` of original input list, and add this element at position 0, and 1 for each list of previous compoundList. **We will require to create copy of all lists of previous compoundList, and add the new element.** Now, the compoundList will become `[[1, 2], [2, 1]]`.

3. Pick next element `0` of original input list, and add this element at position 0, 1, and 2 for each list of previous compoundList. Now, the compoundList will become `[[0, 1, 2], [1, 0, 2], [1, 2, 0], [0, 2, 1], [2, 0, 1], [2, 1, 0]]`.

Additional Resource While dealing with a "compound" object, a simple copy operation might not work as expected. You would need a function that can create a deep copy. For this purpose, you can make use of `deepcopy()` function from the `copy` module in Python. This module provides the function for normal (Shallow) and deep copy operations. Refer here - https://docs.python.org/3/library/copy.html for syntax and detailed information, that says: >**Difference between Deep and Shallow Copy** The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances): - A shallow copy constructs a new compound object and then inserts references into it to the objects found in the original. - A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

Example Illustration of deep copy, shallow copy, and assignment operator

```
In [ ]: import copy                                                   # `copy` module

        list1 = [0, 1, 2]
        list2 = [7, 8, 9]
        compoundList1 = [list1, list2]                        # create a compound object


        '''ASSIGNMENT OPERATION - Points a new reference to the existing object.'''
        compoundList2 = compoundList1

        # id() - returns the identity of the object passed
        print(id(compoundList1) == id(compoundList2))          # True - compoundList2 is the sam
        print(id(compoundList1[0]) == id(compoundList2[0]))    # True - compoundList2[0] is the


        '''SHALLOW COPY'''
        compoundList2 = copy.copy(compoundList1)

        print(id(compoundList1) == id(compoundList2))          # False - compoundList2 is now a
        print(id(compoundList1[0]) == id(compoundList2[0]))    # True - compoundList2[0] is the


        '''DEEP COPY'''
        compoundList2 = copy.deepcopy(compoundList1)

        print(id(compoundList1) == id(compoundList2))          # False - compoundList2 is now a
        print(id(compoundList1[0]) == id(compoundList2[0]))    # False - compoundList2[0] is now
```

1.0.1 Exercise - Write the function definition here

```
In [ ]: # Code

        import copy

        def permute(inputList):
            """
            Args: myList: list of items to be permuted
            Returns: list of permutation with each permuted item being represented by a list
            """
            pass
```

1.0.2 Test - Let's test your function

```
In [ ]: # Test Cases

        # Helper Function
        def check_output(output, expected_output):
            """
            Return True if output and expected_output
```

```
    contains the same lists, False otherwise.

    Note that the ordering of the list is not important.

    Examples:
        check_output([ [0, 1], [1, 0] ] ], [ [1, 0], [0, 1] ]) returns True

    Args:
        output(list): list of list
        expected_output(list): list of list

    Returns:
        bool
    """
    o = copy.deepcopy(output)  # so that we don't mutate input
    e = copy.deepcopy(expected_output)  # so that we don't mutate input

    o.sort()
    e.sort()
    return o == e

print ("Pass" if  (check_output(permute([]), [[]])) else "Fail")
print ("Pass" if  (check_output(permute([0]), [[0]])) else "Fail")
print ("Pass" if  (check_output(permute([0, 1]), [[0, 1], [1, 0]])) else "Fail")
print ("Pass" if  (check_output(permute([0, 1, 2]), [[0, 1, 2], [0, 2, 1], [1, 0, 2], [1
```

Hide Solution

```
In [ ]: # Recursive Solution
        """
        Args: myList: list of items to be permuted
        Returns: compound list: list of permutation with each permuted item being represented by
        """
        import copy                                # We will use `deepcopy()` function from the

        def permute(inputList):

            # a compound list
            finalCompoundList = []                  # compoundList to be returned

            # Terminaiton / Base condition
            if len(inputList) == 0:
                finalCompoundList.append([])

            else:
                first_element = inputList[0]        # Pick one element to be permuted
                after_first = slice(1, None)        # `after_first` is an object of type 'slice'
                rest_list = inputList[after_first]  # convert the `slice` object into a list
        # Recursive function call
        sub_compoundList = permute(rest_list)

        # Iterate through all lists of the compoundList returned from previous call
        for aList in sub_compoundList:

            # Permuted the `first_element` at all positions 0, 1, 2 ... len(aList) in ea

            for j in range(0, len(aList) + 1):

                # A normal copy/assignment will change aList[j] element
                bList = copy.deepcopy(aList)

                # A new list with size +1 as compared to aList
                # is created by inserting the `first_element` at position j in bList
                bList.insert(j, first_element)

                # Append the newly created list to the finalCompoundList
                finalCompoundList.append(bList)

    return finalCompoundList
```