Check Permutation

In this lesson, you will learn how to check if a string is a permutation of another string in Python.

WE'LL COVER THE FOLLOWING

- Solution 1
  - Implementation
  - Explanation
- Solution 2
  - Implementation
  - Explanation

In this lesson, we will consider how to determine if a given string is a permutation of another string.

Specifically, we want to solve the following problem:

Given two strings, write a function to determine if one is a permutation of the other.

Here is an example of strings that are permutations of each other:

```
is_permutation_1 = "google"
is_permutation_2 = "ooggle"
```

The strings below are not permutations of each other.

```
not_permutation_1 = "not"
not_permutation_2 = "top"
```

We will solve this problem in Python and analyze the time and space complexity of our approach.

Let's begin!

Solution 1

Implementation

A permutation of a string will have the same number of each type of character as that string as it is just a rearrangement of the letters.

Let's have a look at the code below where we make use of this property:

```
# Approach 1: Sorting
# Time Complexity: O(n log n)
# Space Complexity: O(1)
def is_perm_1(str_1, str_2):
    str_1 = str_1.lower()
    str_2 = str_2.lower()

    if len(str_1) != len(str_2):
        return False

    str_1 = ''.join(sorted(str_1))
    str_2 = ''.join(sorted(str_2))

    n = len(str_1)

    for i in range(n):
        if str_1[i] != str_2[i]:
            return False
    return True
```

is_perm_1()

Explanation

First of all, both the strings `str_1` and `str_2` are normalized on **lines 5-6** as all the characters are converted to lowercase. On **line 8**, we check for a basic permutation property, i.e., the two strings that are permutations of each other have to be of the same length. Therefore, `False` is returned on **line 9** if the length of `str_1` does not equal length of `str_2`.

On **lines 11-12**, both the strings are sorted using the `sorted` function which returns a list. That list is again turned into a string using the `join` function. At this point, `str_1` and `str_2` will be two strings with all the characters sorted in them.

Now if `str_1` is a permutation of `str_2`, all the characters will be the same in the sorted version of both the strings. We'll check this using the `for` loop on **line 16** which runs from `i` equal `0` to `i` equal `n-1`. To traverse the strings,

we make use of `n` which is set to `len(str_1)`. So, on **line 17**, we check for the equality of the characters on the same index. If, in any iteration, `str_1[i]` is not equal to `str_2[i]`, `False` is returned on **line 18**. Otherwise, if `False` is never returned from the function, `True` is returned on **line 19**.

As we are using sorting to check for permutations, the time complexity for this solution is $O(nlogn)$ while space complexity is $O(1)$ as there is no extra use of space.

Solution 2

As Solution 1 has $O(nlogn)$ complexity, we need something better. Solution 2 makes use of a hash table to reach a time complexity of $O(n)$.

Implementation

Let's find out how by having a look at the code below:

```
# Approach 2: Hash Table
# Time Complexity: O(n)
# Space Complexity: O(n)
def is_perm_2(str_1, str_2):
    str_1 = str_1.lower()
    str_2 = str_2.lower()

    if len(str_1) != len(str_2):
        return False

    d = dict()

    for i in str_1:
        if i in d:
            d[i] += 1
        else:
            d[i] = 1
    for i in str_2:
        if i in d:
            d[i] -= 1
        else:
            d[i] = 1

    return all(value == 0 for value in d.values())
```

is_perm_2()

Explanation

The code from **line 5** to **line 9** is the same as in Solution 1. On **line 11**, `d` is initialized to a Python dictionary. Using a `for` loop on **line 13** where `i` equals

a character in `str_1`, `i` is stored as a key in `d` with `1` as a value if it's not present in `d`. If it's present, its value is incremented by `1`. After this `for` loop,

`d` will have the entire count of all the characters present in `str_1`. We'll make use of this in the `for` loop on **line 18** where `str_2` is traversed. If any character in `str_2` is present as a key in `d`, its value is decremented by `1`. If it's not already present in `d`, it is inserted into `d` as a key with `value` equal to `1`.

Now, if `str_1` and `str_2` are permutations of each other, the two `for` loops will counterbalance each other and the values of all the keys in `d` should be `0`. We find this out on **line 24** where we evaluate `value == 0` for all the keys in `d`. Using the `all` function, we combine the evaluation results from all the iterations and return it from the function. The `all()` function returns `True` if all items in an iterable are `True`. Otherwise, it returns `False`.

In the code widget below, you can execute both the functions on different test cases.

```
# Approach 1: Sorting
# Time Complexity: O(n log n)
# Space Complexity: O(1)
def is_perm_1(str_1, str_2):
    str_1 = str_1.lower()
    str_2 = str_2.lower()

    if len(str_1) != len(str_2):
        return False

    str_1 = ''.join(sorted(str_1))
    str_2 = ''.join(sorted(str_2))

    n = len(str_1)

    for i in range(n):
        if str_1[i] != str_2[i]:
            return False
    return True


# Approach 2: Hash Table
# Time Complexity: O(n)
# Space Complexity: O(n)
def is_perm_2(str_1, str_2):
    str_1 = str_1.lower()
    str_2 = str_2.lower()

    if len(str_1) != len(str_2):
        return False

    d = dict()
```

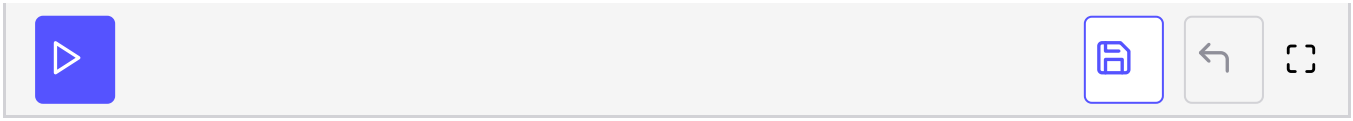
```
    for i in str_1:
        if i in d:
            d[i] += 1
        else:
            d[i] = 1
    for i in str_2:
        if i in d:
            d[i] -= 1
        else:
            d[i] = 1

    return all(value == 0 for value in d.values())

is_permutation_1 = "google"
is_permutation_2 = "ooggle"

not_permutation_1 = "not"
not_permutation_2 = "top"
print(is_perm_1(is_permutation_1, is_permutation_2))
print(is_perm_1(not_permutation_1, not_permutation_2))

print(is_perm_2(is_permutation_1, is_permutation_2))
print(is_perm_2(not_permutation_1, not_permutation_2))
```



Now that you are pretty familiar with the problems regarding string processing, it's test time! Brace yourselves for the upcoming challenge!