Is Anagram

In this lesson, you will learn how to determine if a string is an anagram of another string or not.

WE'LL COVER THE FOLLOWING

In this lesson, we will determine whether two strings are anagrams of each other.

Simply put, an anagram is when two strings can be written using the same letters.

Examples:

```
"rail safety" = "fairy tales"
"roast beef" = "eat for BSE"
```

It sometimes changes a proper noun or personal name into a sentence:

```
"William Shakespeare" = "I am a weakish speller"
"Madam Curie" = "Radium came"
```

We will write two solutions to this problem. The first one will be concise, while the second one will be more robust and efficient. Both of these approaches involve normalizing the input string, which means converting them into lowercase and removing all the characters which are not alphanumeric.

Let's get started with the first approach.

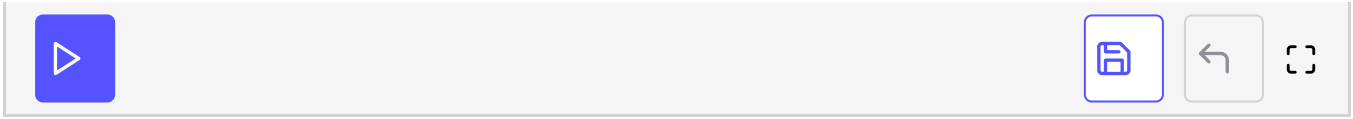
Implementation

Solution 1

```
s1 = "fairy tales"
s2 = "rail safety"

s1 = s1.replace(" ", "").lower()
s2 = s2.replace(" ", "").lower()

# Requires n log n time (since any comparison
# based sorting algorithm requires at least
# n log n time to sort).
print(sorted(s1) == sorted(s2))
```



On **lines 4-5**, `s1` and `s2` are normalized using `replace()` and `lower()` functions. `replace(" ","")` replaces all the spaces with an empty string so that we are only left with alphabets or numbers. `lower()` converts all the alphabets to lowercase. On **line 10**, the two strings are sorted after the normalization and then compared. If `s1` and `s2` are anagrams of each other, both the lists returned from `sorted` will be the same and `True` will be printed onto the console. On the other hand, if `sorted(s1)` does not equal `sorted(s2)`, `False` will be printed onto the console.

You may notice that the solution above is very concise. However, the time complexity for the above algorithm will be $O(nlogn)$ as sorting takes $O(nlogn)$. Therefore, we prefer the second approach, which is more efficient than the sorting approach.

Solution 2: The Preferred Solution

In this solution, we make use of a Python dictionary. First, we store the characters of one string in a dictionary, keeping track of the count of individual characters. Then we compare the count of each character with the characters of the other string. If they counterbalance each other, we'll declare the two strings as an anagram of each other. This solution is of linear time complexity which is an improvement on $O(nlogn)$.

Have a look at the code below to see how we have implemented the approach mentioned above:

```
def is_anagram(s1, s2):
```

```
    ht = dict()

    if len(s1) != len(s2):

        return False

    for i in s1:
        if i in ht:
            ht[i] += 1
        else:
            ht[i] = 1
    for i in s2:
        if i in ht:
            ht[i] -= 1
        else:
            ht[i] = 1
    for i in ht:
        if ht[i] != 0:
            return False
    return True

s1 = "fairy tales"
s2 = "rail safety"
## normalizing the strings
s1 = s1.replace(" ", "").lower()
s2 = s2.replace(" ", "").lower()

print(is_anagram(s1, s2))
```

On **line 2**, `ht` is initialized to a Python dictionary. For two strings to be anagrams of each other, they must be of the same length. Therefore, on **line 4**, we compare the length of the two strings. If they are not equal, `False` is returned on **line 5** before any more computation. If `s1` and `s2` have the same number of characters, the execution jumps to the `for` loop on **line** 7.

This `for` loop iterates over each character of `s1` and checks if it is present in `ht`. If it's not, we store the character `i` as a key in `ht` while setting its value to `1` on **line 11**. If `i` is present in `ht`, then the code on **line 9** executes which increments the value against `i` in `ht` by `1`. Now that we have the count of characters of `s1` stored in `ht`, the code proceeds to the `for` loop on **line 12**. The `for` loop on **line 12** is exactly the same as the previous `for` loop but only with a minor difference. This `for` loop iterates over `s2` and if `i` is present in `ht` as a key, instead of incrementing its value, it decrements its value by `1`.

At this point, if `s1` and `s2` are anagrams of each other, all the keys in `ht` will have `0` as their values. This is because the count of each character in `s2` will cancel out the count of each character in `s1` because they have the same

number of each type of character. So, on **line 17**, as we iterate over every key in `ht`, if `ht[i]` does not equal `0`, `False` is returned from the function to indicate that `s1` and `s2` are not anagrams of each other. If the condition on **line 18** never evaluates to `True`, `True` is returned from the function on **line 20** to declare `s1` and `s2` as anagrams.

Also, note that `s1` and `s2` are normalized on **lines 25-26** to remove the spaces which might result in uneven lengths even if `s1` and `s2` are anagrams of each other.

That is all we have up for this lesson. In the next lesson, we'll study "Is Palindrome Permutation". Stay tuned!