Is Palindrome Permutation

In this lesson, you will learn how to find if a string is a palindrome permutation or not.

We'll cover the following

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

Specifically:

Given a string, write a function to check if it is a permutation of a palindrome.

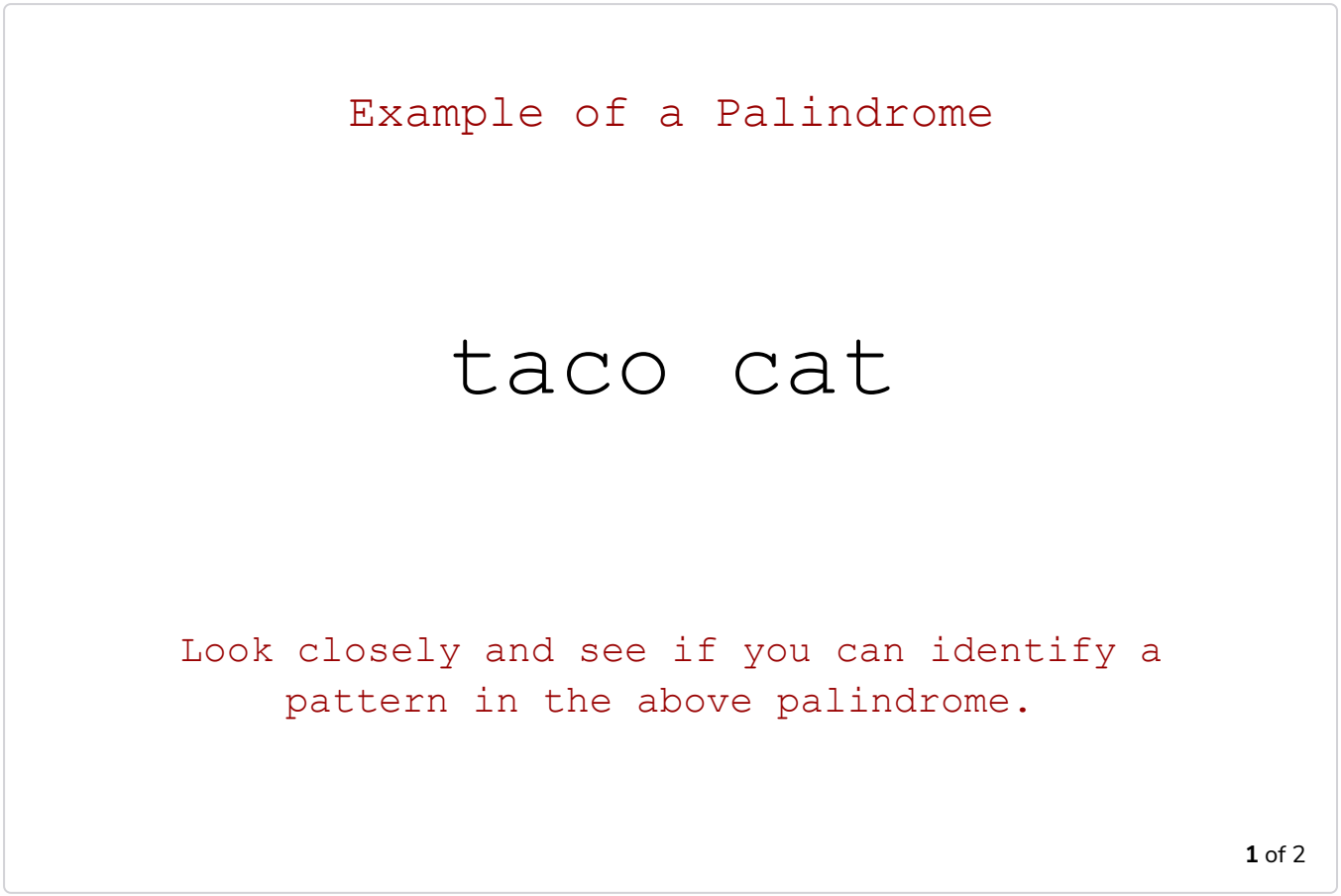
Recall:

> **Palindrome**: A word or phrase that is the same forwards and backward.

> **Permutation**: A rearrangement of letters.

The palindrome does not need to be limited to just dictionary words.

Before jumping straight to the code, let’s think about how we can approach this problem. Here is an example of a palindrome:



From the slides above, the idea we get is that a string that has an even length must have all even counts of characters, while strings that have an odd length must have exactly one character with an odd count. An even-length-ed string can't have an odd number of exactly one character; otherwise, it wouldn't be even. This is true since an odd number plus any set of even numbers will yield an odd number.

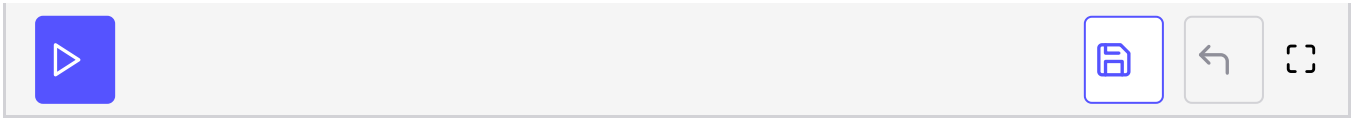
Alternatively, an odd-length-ed string can't have all characters with even counts, the sum of any number of even numbers is even. We can, therefore, say that it's sufficient to be a permutation of a palindrome, that is a string can have no more than one character that is odd.

Implementation

Let's make use of the observation discussed above regarding the palindrome property to implement the code in Python. We use a Python dictionary to solve the problem in the code below:

```
def is_palin_perm(input_str):
```

```
def is_palin_perm(input_str):
    input_str = input_str.replace(" ", "")
    input_str = input_str.lower()

    d = dict()

    for i in input_str:
        if i in d:
            d[i] += 1
        else:
            d[i] = 1

    odd_count = 0
    for k, v in d.items():
        if v % 2 != 0 and odd_count == 0:
            odd_count += 1
        elif v % 2 != 0 and odd_count != 0:
            return False
    return True

palin_perm = "Tact Coa"
not_palin_perm = "This is not a palindrome permutation"

print(is_palin_perm(palin_perm))
print(is_palin_perm(not_palin_perm))
```



is_palin_perm()

Explanation

Lines 2-3 are covering the normalization process which we also discussed in the previous lesson. Once all the spaces are removed from `input_str` and the characters in `input_str` are converted to lowercase, a dictionary named `d` is defined on **line 5**.

On **line** 7, by using a `for` loop, all the characters in `input_str` are stored as keys in `d` with their values equal to their number of occurrences in `input_str`.

`odd_count` is initialized to `0` on **line 13**. Next, we loop over `d` using the `for` loop on **line 14** where `k` is the key of each item, and `v` is the value for that key. On **line 15**, we check if `v` is an odd number and `odd_count` equals `0`. If for some `k`, `v` is `odd` and `odd_count` is `0`, we increment `odd_count` by `1` on **line 6**. In this way, we are recording for an instance if we have encountered the middle element of an odd-length string. However, on **line 17**, if `v` is an odd number, but `odd_count` equals something other than `0`, it implies that there is more than one character which has an odd number of occurrences in

`input_str`. Therefore, `False` is returned on **line 18** to indicate that the condition on **line 17** is `True` and `input_str` is not a permutation of a palindrome. On the other hand, if all goes well and the condition on **line 17** never evaluates to `True`, `True` is returned on **line 19** to indicate a positive response.

I hope this explanation clarifies the implementation. In the next lesson, we will discuss the implementation to solve the “Check Permutation” problem. Happy learning!