June5

04 June 2020 10:36

Python string comparison

To compare two strings, we mean that we want to identify whether the two strings are equivalent to each other or not, or perhaps which string should be greater or smaller than the other.

This is done using the following operators:

- ==: This checks whether two strings are equal
- !=: This checks if two strings are not equal
- <: This checks if the string on its left is smaller than that on its right
- <=: This checks if the string on its left is smaller than or equal to that on its right
- >: This checks if the string on its left is greater than that on its right
- >=: This checks if the string on its left is greater than or equal to that on its right

How to execute the comparison

String comparison in Python takes place character by character. That is, characters in the same positions are compared from both the strings.

If the characters fulfill the given comparison condition, it moves to the characters in the next position. Otherwise, it merely returns False.

Note:

- ✓ The comparisons are case-sensitive, hence same letters in different letter cases(upper/lower) will be treated as separate characters
- ✓ If two characters are different, then their Unicode value is compared; the character with the smaller Unicode value is considered to be lower.

Python String equals case-insensitive check

Sometimes we don’t care about the case while checking if two strings are equal, we can use **casefold()**, **lower()** or **upper()** functions for case-insensitive equality check.

- ❖ **What if characters are same but order is not?**

If the scenario is to check two strings equality even if the order of words or characters is different then you may first use the sort function and then compare both strings.

For example, consider this string:

Str1 = “Hello and Welcome”

Str1 = “Welcome and Hello”

This is called as '**Anagrams**'

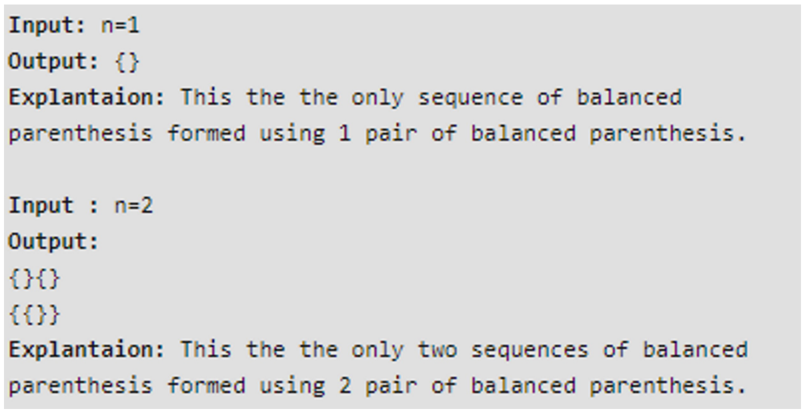
Problem Description
The program takes two strings and checks if the two strings are anagrams.

Algorithm

- Take two strings from the user and store them in separate variables.
- Then use sorted() to sort both the strings into lists.
- Compare the sorted lists and check if they are equal.
- Print the final result.
- Exit.

❖ Print all combinations of balanced parentheses

Write a function to generate all possible n pairs of balanced parentheses.

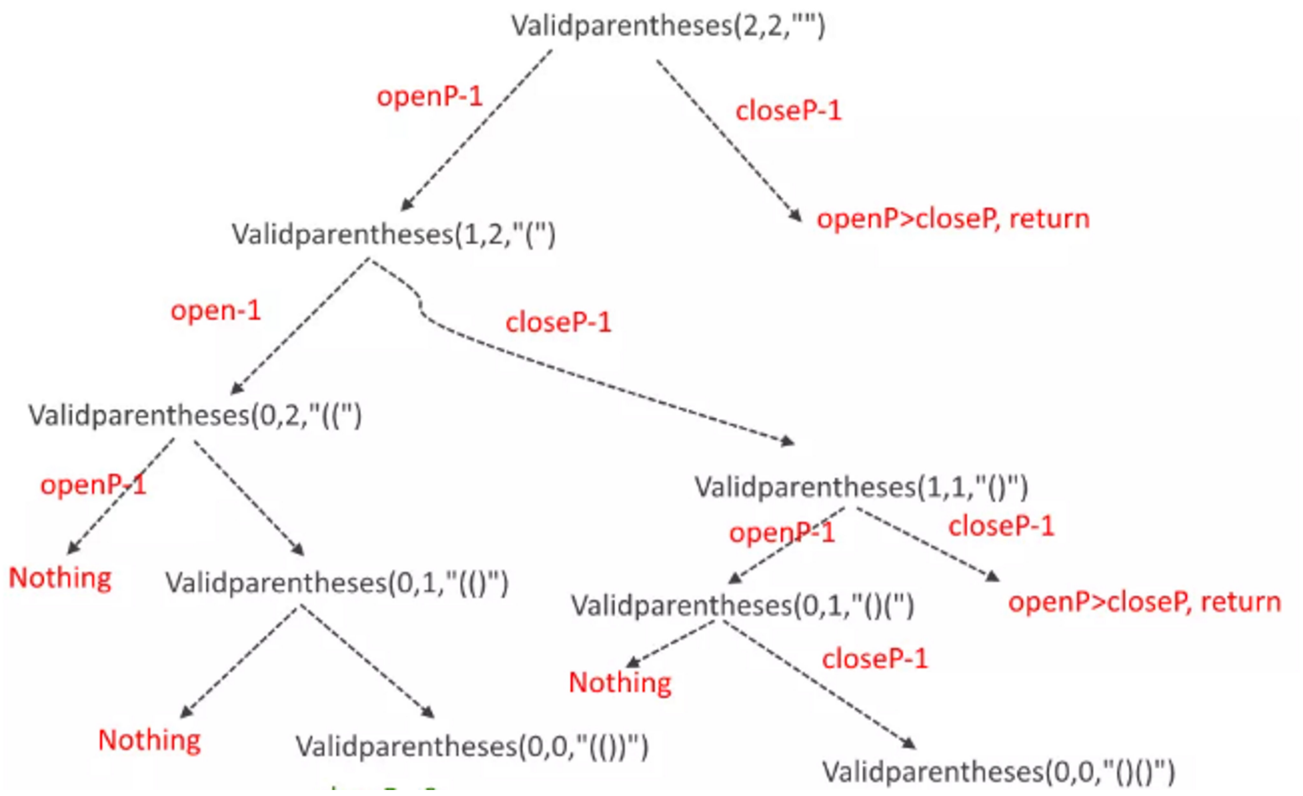


```
Input: n=1
Output: {}
Explantaion: This the the only sequence of balanced
parenthesis formed using 1 pair of balanced parenthesis.

Input : n=2
Output:
{}{}
{{}}
Explantaion: This the the only two sequences of balanced
parenthesis formed using 2 pair of balanced parenthesis.
```

- Recursion is the key here.
- Divide the N into N/2 and N/2 (Count for open and closed parentheses).
- Select the open parentheses, add it to the result string and reduce its count and make a recursive call.
- Select the close parentheses, add it to the result string and reduce its count and make a recursive call.
- To print only valid parentheses, make sure at any given point of time, close parentheses count is not less than open parentheses count because it means close parentheses has been printed with its respective open parentheses.

See picture for better explanation.



Validparentheses(2,2,"")
openP-1
closeP-1
openP>closeP, return
Validparentheses(1,2,"(")
open-1
closeP-1
Validparentheses(0,2,"((")
openP-1
Nothing
Validparentheses(0,1,"(()")
Nothing
Validparentheses(0,0,"(())")
Validparentheses(1,1,"()")
openP-1
closeP-1
openP>closeP, return
Validparentheses(0,1,"()(")
Nothing
closeP-1
Validparentheses(0,0,"()()")

Algorithm:

- Create a recursive function that accepts a string (s), count of opening brackets (o) and count of closing brackets (c) and the value of n.
- if the value of opening bracket and closing bracket is equal to n then print the string and return.
- If the count of opening bracket is greater than count of closing bracket then call the function recursively with the following parameters String s + “}”, count of opening bracket o, count of closing bracket c + 1, and n.
- If the count of opening bracket is less than n then call the function recursively with the following parameters String s + “{“, count of opening bracket o + 1, count of closing bracket c, and n.

❖ Find the smallest window in a string containing all characters of another string

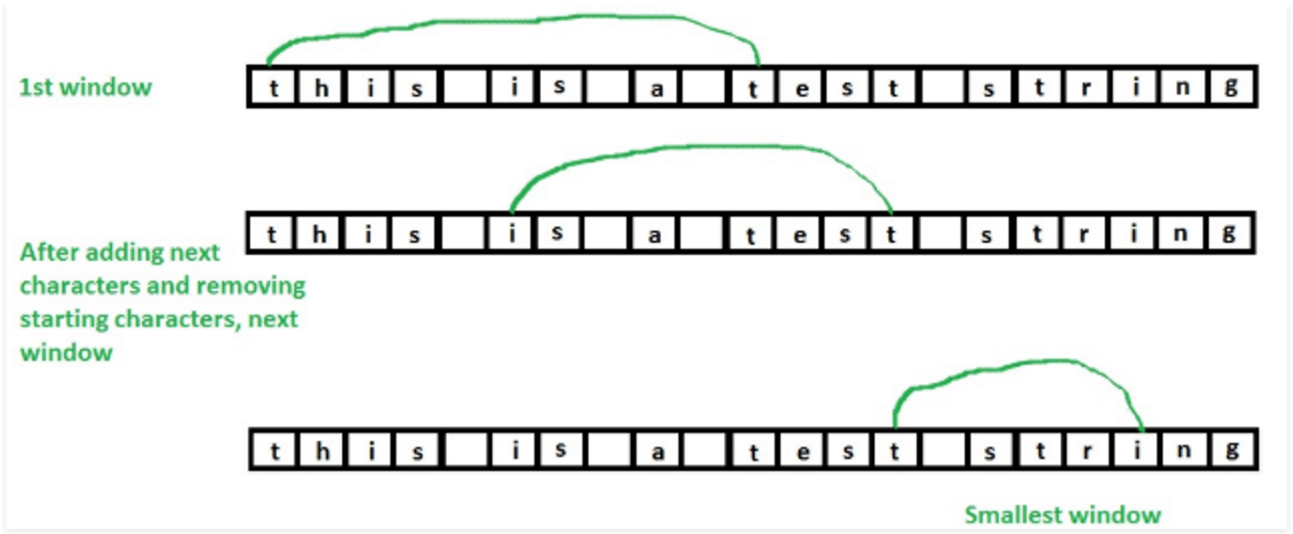
Given two strings string1 and string2, the task is to find the smallest substring in string1 containing all characters of string2 **efficiently**.

Example

Input: *string = “this is a test string”, pattern = “tist”*

Output: *Minimum window is “t stri”*

Explanation: *“t stri” contains all the characters of pattern.*


1st window
After adding next characters and removing starting characters, next window
Smallest window
t h i s i s a t e s t s t r i n g

Method 1 (Brute force solution)
1- Generate all substrings of string1 (“this is a test string”)
2- For each substring, check whether the substring contains all characters of string2 (“tist”)
3- Finally, print the smallest substring containing all characters of string2.

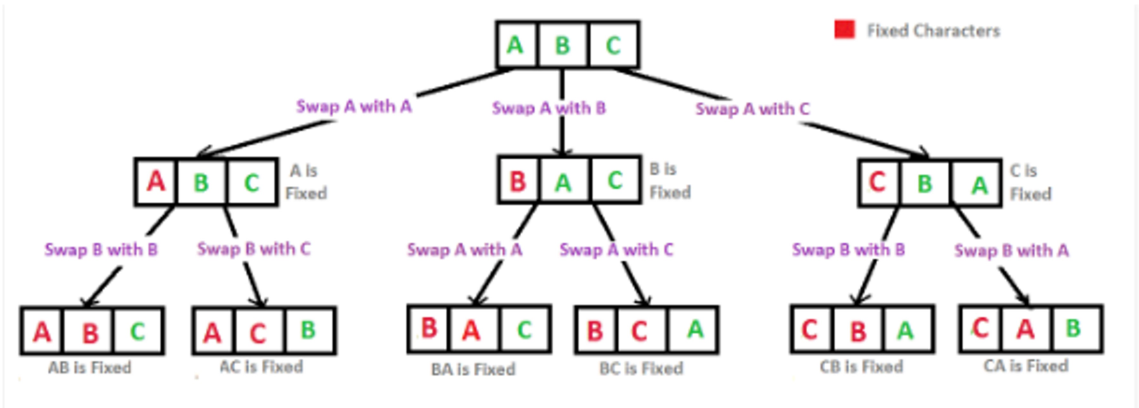
Method 2 (Efficient Solution)

- First check if the length of string is less than the length of the given pattern, if yes then “no such window can exist “.
- Store the occurrence of characters of the given pattern in a hash_pat[].
- Start matching the characters of pattern with the characters of string i.e. increment count if a character matches.
- Check if (count == length of pattern) this means a window is found.
- If such window found, try to minimize it by removing extra characters from the beginning of the current window.
- Update min_length.
- Print the minimum length window.

❖ **Write a program to print all permutations of a given string**

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.

Below are the permutations of string ABC.
ABC ACB BAC BCA CBA CAB
Here is a solution that is used as a basis in backtracking.



Recursion Tree for Permutations of String "ABC"

Algorithm Paradigm: Backtracking
Time Complexity: O(n*n!) Note that there are n! permutations and it requires O(n) time to print a a permutation.

Questions to attempt.

1. **Check Whether two strings are anagram of each other.**
2. **Find strings given length containing balanced parentheses using recursion. (Input n= 4, n=6)**
3. **Given a set of characters CHARS and a input string INPUT, find the minimum window in str**
   which will contain all the characters in CHARS in complexity O(n).
   For example, INPUT = ABBACBAA and CHARS = AAB has the minimum window BAA.
4. **Give an algorithm for printing all possible permutations of the characters in a string. Unlike combinations, two permutations are considered distinct if they contain the same characters but in a different order.**
   For simplicity assume that each occurrence of a repeated character is a distinct character. That is, if the input is "AAA", the output should be six repetitions of "AAA".The permutations may be output in any order.