Stack

A Stack is a special kind of data structure that follows the LIFO (Last In First Out) principle. In Python, we can implement a stack using different data structures like list, collections.deque, and queue.LifoQueue.

Stack operations include `push`, which adds an element to the top of the stack, and `pop`, which removes an element from the top of the stack. These operations are both O(1) operations, as they only involve a single step that does not vary with the size of the stack.

Stacks are useful in a variety of programming scenarios, such as parsing (like in HTML tags, or in Python's own syntax), call stack (like when functions call other functions), and backtracking algorithms.

The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)
- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

Implementation:

There are various ways from which a stack can be implemented in Python. This article covers the implementation of a stack using data structures and modules from the Python library.

Homework Question 1:

Given a string, reverse it using stack.

Example:

> Input: str = “GeeksQuiz”Output: ziuQskeeGInput: str = “abc”Output: cba

Approach:

> The idea is to create an empty stack and push all the characters from the string into it. Then pop each character one by one from the stack and put them back into the input string starting from the 0’th index. As we all know, stacks work on the principle of first in, last out. After popping all the elements and placing them back to string, the formed string would be reversed.

Follow the steps given below to reverse a string using stack.

- Create an empty stack.
- One by one push all characters of string to stack.
- One by one pop all characters from stack and put them back to string.

Homework Question 2:

Given an expression string **exp**, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the given expression.

Example:

> Input: exp = “[()]{}{[()()]()}” Output: BalancedExplanation: all the brackets are well-formedInput: exp = “[(])” Output: Not Balanced Explanation: 1 and 4 brackets are not balanced because there is a closing ‘]’ before the closing ‘(‘

Approach:

> The idea is to put all the opening brackets in the stack. Whenever you hit a closing bracket, search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration. In the end if the stack is empty, it means all brackets are balanced or well-formed. Otherwise, they are not balanced.

Follow the steps mentioned below to implement the idea:

- Declare a character **stack** (say **temp**).
- Now traverse the string exp.
    - If the current character is a starting bracket (**‘(‘ or ‘{‘ or ‘[‘**) then push it to stack.
    - If the current character is a closing bracket (**‘)’ or ‘}’ or ‘]’**) then pop from the stack and if the popped character is the matching starting bracket then fine.
    - Else brackets are **Not Balanced**.
- After complete traversal, if some starting brackets are left in the stack then the expression is **Not balanced**, else **Balanced**.