Determine if Brackets are Balanced

This lesson will teach us how to determine whether or not a string has balanced usage of brackets by using a stack.

WE'LL COVER THE FOLLOWING

In this lesson, we're going to determine whether or not a set of brackets are balanced or not by making use of the stack data structure that we defined in the previous lesson.

Let's first understand what a balanced set of brackets looks like.

A balanced set of brackets is one where the number and type of opening and closing brackets match and that is also properly nested within the string of brackets.

Examples of Balanced Brackets

- `{}`
- `{}{}`
- `(({[]}))`

Examples of Unbalanced Brackets

- `(()`
- `{{{)}]`

- `[][][]`

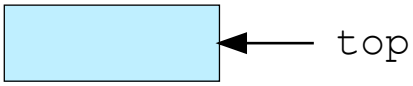
Algorithm

Check out the slides below to have a look at the approach we'll use to solve this problem:

Balanced Example

([ ])

We have a balanced example of brackets and an empty stack.

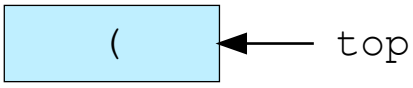


1 of 6

Balanced Example

([ ])

We iterate through the string and push the bracket onto the stack if it's an opening bracket.

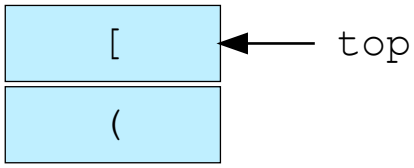


2 of 6

Balanced Example

([ ])

We iterate through the string and push the bracket onto the stack if it's an opening bracket.



Balanced Example

([ ])

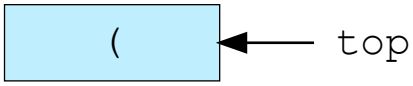
Now we encounter a closing square bracket so we pop off from the stack to check for a match. It is a match so we move on.

The top of the stack should be an opening square bracket to match the closing square bracket.

'[' (popped off)

matches

']'



Balanced Example

([ ])

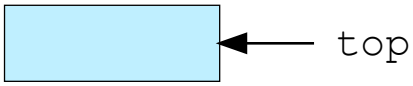
Now we encounter another closing bracket and we check for a match from the popped off element from stack.

The top of the stack should be an opening parenthesis to match the closing parenthesis.

'(' (popped off)

matches

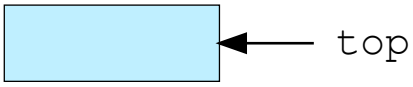
')'



Balanced Example

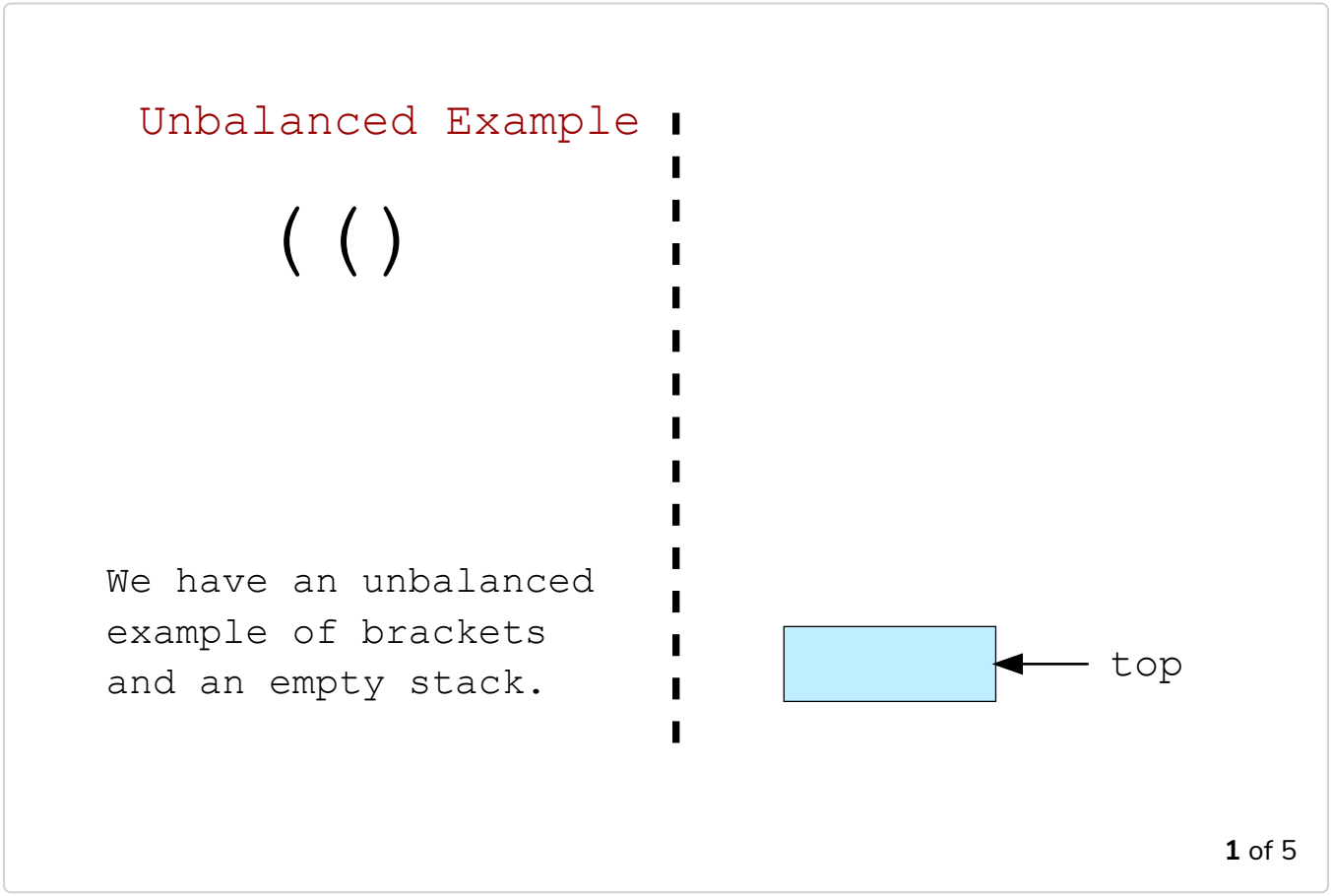
([ ])

As we have iterated over all the characters of the string and the stack is empty, we conclude that the string has a balanced usage of brackets.



As shown above, our algorithm is as follows:

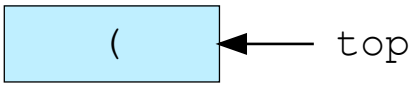
- We iterate through the characters of the string.
- If we get an opening bracket, push it onto the stack.
- If we encounter a closing bracket, pop off an element from the stack and match it with the closing bracket. If it is an opening bracket and of the same type as the closing bracket, we conclude it is a successful match and move on. If it's not, we will conclude that the set of brackets is not balanced.
- The stack will be empty at the end of iteration for a balanced example of brackets while we'll be left with some elements in the stack for an unbalanced example.



Unbalanced Example

(( )

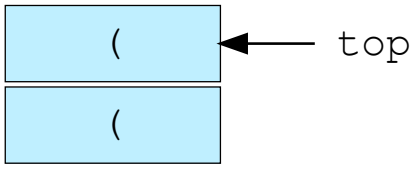
We iterate through the string and push the bracket onto the stack if it's an opening bracket.



Unbalanced Example

(( )

We iterate through the string and push the bracket onto the stack if it's an opening bracket.



Unbalanced Example

(( )

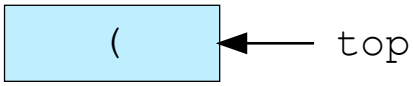
Now we encounter a closing bracket so we pop off from the stack to check for a match.It is a match so we move on.

The top of the stack should be an opening parenthesis to match the closing parenthesis.

'(' (popped off)

matches

')'

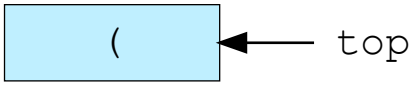


4 of 5

Unbalanced Example

(( )

We have iterated over the entire string but we still have a non-empty stack. Therefore, we conclude that brackets are not balanced in this string.



5 of 5

We've covered an example for both the balanced set of brackets and an unbalanced set of brackets, let's move on to a special case.

Special Case

Example:))

What if we encounter a closing bracket but don't have any elements to pop off from the stack? For example, in the case described above, we don't have an opening parenthesis, but we encounter a closing parenthesis. In this case, we immediately know that the string does not have a balanced usage of brackets. Therefore, we need to watch out for an empty stack in our implementation.

Now that you have got a decent idea of the algorithm, we'll go over the implementation of it in Python.

Let's start with the `is_paren_balanced` function:

```
def is_paren_balanced(paren_string):
    s = Stack()
    is_balanced = True
    index = 0

    while index < len(paren_string) and is_balanced:
        paren = paren_string[index]
        if paren in "([{":
            s.push(paren)
        else:
            if s.is_empty():
                is_balanced = False
            else:
                top = s.pop()
                if not is_match(top, paren):
                    is_balanced = False
        index += 1

    if s.is_empty() and is_balanced:
        return True
    else:
        return False
```

is_paren_balanced(paren_string)

Explanation

On **lines 2-4**, we declare a stack, `s` and two variables `is_balanced` and `index`, which are set to `True` and `0`, respectively.

The `while` loop on **line 6** will execute if the `index` is less than the length of `paren-string` **and** `is_balanced` is equal to `True`. If any of the conditions evaluate to `False`, our program will exit the `while` loop. In the while loop, we

iterate over each character of the `paren_string` by indexing using the `index` variable and save the indexed element in `paren` variable.

We check on **line 8** whether `paren` is any type of the opening brackets and if it is, we push it onto the stack. If it's not any type of the opening brackets, we check if stack `s` is empty and set `is_balanced` to `False`. This handles our special case, which we discussed in the previous section.

If the stack is not empty, we pop off the top element and check if the current `paren`, i.e., a closing bracket matches the type of the top element which is supposed to be an opening bracket. If the types don't match, then we update `is_balanced` to `False`.

We increment the `index` for the next iteration. The `while` loop keeps executing until the index is equal to or greater than the length of `paren_string` or `is_balanced` equals `False`.

After we exit the while loop, on **line 19**, we check if the stack is empty and `is_balanced` is `True`, then we return `True`. Otherwise, we return `False`.

The code given above is a simple implementation of the algorithm you were introduced to.

Let's implement the `is_match` function now:

```
def is_match(p1, p2):
    if p1 == "(" and p2 == ")":
        return True
    elif p1 == "{" and p2 == "}":
        return True
    elif p1 == "[" and p2 == "]":
        return True
    else:
        return False
```

is_match(p1, p2)

Explanation

The `is_match` function takes in two characters as `p1` and `p2` and evaluates whether they are a valid pair of brackets. For `p1` and `p2` to match, `p1` has to be an opening bracket while `p2` has to be the corresponding closing bracket. If `p1` and `p2` don't fall in any of the valid conditions, we return `False`.

Easy peasy, right?

Easy peasy, right?

The entire implementation of the solution to determine whether a string has a balanced usage of brackets is given below. Feel free to play around with it!

main.py

stack.py

```
from stack import Stack

def is_match(p1, p2):
    if p1 == "(" and p2 == ")":
        return True
    elif p1 == "{" and p2 == "}":
        return True
    elif p1 == "[" and p2 == "]":
        return True
    else:
        return False


def is_paren_balanced(paren_string):
    s = Stack()
    is_balanced = True
    index = 0

    while index < len(paren_string) and is_balanced:
        paren = paren_string[index]
        if paren in "([{":
            s.push(paren)
        else:
            if s.is_empty():
                is_balanced = False
            else:
                top = s.pop()
                if not is_match(top, paren):
                    is_balanced = False
        index += 1

    if s.is_empty() and is_balanced:
        return True
    else:
        return False

print("String : (((({}))))) Balanced or not?")
print(is_paren_balanced("(((({})))))"))

print("String : [][]]] Balanced or not?")
print(is_paren_balanced("[][]]]"))

print("String : [][] Balanced or not?")
print(is_paren_balanced("[][]"))
```

In the next lesson, we will go over another problem, i.e. reversing a string and solving it using a stack. See you there!

solving it using a stack. See you there.