Part 3:

Create a Jupyter Notebook, create 6 of the following headings, and complete the following for your partner's assignment 1:

- Paraphrase the problem in your own words.

The problem consists on identifying whether a string consisting of parenthesis, curly braces and squared brackets are a valid sequence. It follows a base from math where every opened bracket has to closed, inside of it can exist multiple other brackets, but each of them has to be closed properly.

For example in math, something like the following would work:

((a + b) * (c + d) - [(x - 1)]) -> each inner parenthesis is closed properly before going into the next layer

Contrary to this example:

(a + b)) - [x{*y-1]} -> where there's extra characters or thing are not closed properly, even vscode tells me with red characters that something in that formula is wrong

- Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it.

For a proper example more exactly related to the problem:

Input: s = "([{}{{()}}])" Output: True

Input: s = "(){{[([])]]}}" Output: False

Let's trace the following example that Darling set: "([{({}[])}])"

For easy visualization, we'll indent it in multiple lines until we find wether it closes properly or not: ([{ ({} []) }])

^ Base on the above we can easily see the example follows the structure, so it would return True

However for this other example "{[}]", if we try to do the same:

{ [}]

We'll see that symbols don't match, so we can assume this is incorrect and should return False

- Copy the solution your partner wrote.

In [ ]:
```python
def is_valid_brackets(s: str) -> bool:
    #In here I will define an empty stack to track the opening brackets as a
    stack = []
    # Now, I will define a dictionary that maps closing brackets to opening
    # just to remember a dictionary is a pair of keys and values in the form
    #               key:value

    bracket_dict = {')': '(',
                    '}': '{',
                    ']': '['
                    }

    # I will loop through each character in the input string
    for char in s:

        #In here I will check if the character is an opening or closing brac

        #For opening brackets I will append it to the stack
        if char in bracket_dict.values():  # opening brackets check
            stack.append(char)

        #For closing brackets I will compare it with the top of the stack
        elif char in bracket_dict:  # closing brackets
            if not stack or stack[-1] != bracket_dict[char]: #is the stack e
                return False
            stack.pop()    #if not return false, then I will pop the top of
        else:
            # If there are any invalid characters (not expected in this prob
            return False
    return not stack  # stack should be empty if all brackets matched
```

- Explain why their solution works in your own words.

Starting with the base values, using a dictionary with the closing brackets as keys and opening brackets as values is the right approach for a clean solution.

All characters need to be looped since even the last character could be the one offending and causing the solution to turn False.

Each time there's an opening bracket, it will be added to the stack, as in waiting for the closing bracket in order to be removed in a FILO or LIFO order, that way it maintains the valid structure.

If there's a closing bracket without a match, or if there's any other character left at the end of the string, a false will be returned.

- Explain the problem's time and space complexity in your own words.

Since there's only one iteration of the input list, the time complexity is O(n).

As for the space complexity, the bracket dict would be O(1), and the stack itself would be at worst case O(n), therefore the space complexity is O(n) too.

- Critique your partner's solution, including explanation, and if there is anything that should be adjusted.

Since the solution is working, it's hard to find a flaw that can be mentioned, let's start with the non-technical aspect that's easy to identify. The excessive comments reduce readibility, instead a common standard for writing methods would be as follows:

Comments explaining how function works

Adding multiple lines if needed

def function(): logic

Maybe a comment here for this extra hard line to understand (very niche situations)

very_hard_line_to_understand return result

On the time and space complexity, I don't see any improvements that are possible.

One other improvement would be the cleanup of input, meaning identifying whether an input has valid or invalid characters, whether to discard them automatically or simply remove those characters and process the string as valid or not. In my opinion, cleaning or rejection of input should happen in a separate function, as in principle, we should keep functions as short as possible and doing only one thing at a time.

Reflection

Reviewing Darling's bracket validation solution was an insightful experience that helped me understandand both the problem and practice my code review skills. I approached the review by starting with paraphrasing the problem using mathematical analogies, which helped me grasp the core concept more deeply.

Creating my own examples and tracing through Darling's test cases was interesting. The visual indentation technique I used to demonstrate bracket nesting made the logic crystal clear - it's something I hadn't considered before but proves effective for explaining nested structures.

Darling's solution was solid technically, making my critique focus more on code quality and best practices rather than algorithmic flaws. I identified that excessive commenting actually hurt readability, which taught me about finding the right balance in documentation. My suggestion about input validation and keeping functions focused on single responsibilities reflects my understanding of clean code principles.

The review process reinforced my belief that functions should do one thing well. Overall, this exercise tested my ability to analyze code objectively and provide constructive feedback while understanding different coding styles and approaches.