Mathemagical

Align strings over a fixed number of positions

You can use the *format specifier* of the string method `format` to reserve a fixed number of positions to output a given text fragment. This is done by formatting the text as a string (indicated by the letter `s`), preceded by an integer that indicates the fixed number of positions that needs to be reserved for the string.

```
>>> f"{'abc':5s}"   # reserve 5 positions
'abc  '
```

In case the string is longer than the number of reserved positions, the placeholder is replaced by the entire string, and thus takes more than the reserved space. In case the string is shorter than the number of reserved positions, the string is left aligned by default. You can also explicitly define the type of alignment in the *format specifier*: left, right or centered. Python will automatically pad the string with additional spaces to the left and/or to the right.

```
>>> f"{'abc':<5s}"   # reserve 5 positions, left alignment
'abc  '
>>> f"{'abc':>5s}"   # reserve 5 positions, right alignment
'  abc'
>>> f"{'abc':^5s}"   # reserve 5 positions, centered alignment
' abc '
```

In case a centered alignment is chosen and the number of additional spaces is an odd number, Python prefers to add one more space to the right than to the left.

Iterate both the elements and their positions of sequence types

The built-in function `enumerate` can be used to request an iterator for a given sequence type (strings, lists, tuples, files, ...) that both returns the position and the value at that position for the next element of the sequence type. The example below illustrates how this can be used to simultaneously iterate the positions of a string and the characters on those position.

```
>>> for index, character in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'character: {character}')
...
index: 0
character: a
index: 1
character: b
index: 2
character: c
```

You can also use this to simultaneously iterate over the characters of two strings.

```
>>> first = 'abc'
>>> second = 'def'
>>> for index, character in enumerate(first):
...    print(f'{character}-{second[index]}')
...
a-d
b-e
c-f
```

However, in this case it is better to use the built-in function `zip`, which is especially equipped to iterate over multiple iterable objects at once.

```
>>> first = 'abc'
>>> second = 'def'
>>> for character1, character2 in zip(first, second):
...     print(f'{character1}-{character2}')
...
a-d
b-e
c-f
```

String repetition

To repeat a given string a fixed number of times, you can multiply that string with an integer. As with the multiplication of numbers, this uses the `*` operator. The order of the string and the integer does not matter in the multiplication

```
>>> 'a' * 3
'aaa'
>>> 5 * 'ab'
'abababababab'
```

This is very handy in case the number of repetitions of the string is not known beforehand, but for example can be retrieved from a variable.

```
>>> repetitions = 4
>>> repetitions * 'abc'
'abcabcabcabc'
```

Alchemical reduction

Specific information

The following overview can be used to construct your solution. Before implementing this algorithm it is very important to make sure you understand it fully. So take your time to test these steps with pen and paper.

1. Build a new polymer, character by character.
    1. If the last added letter to the new polymer reacts with the current letter, then remove the last letter of the new polymer.
    2. Else, add the current letter to the end of the new polymer.
2. The new polymer is the shortest representation of the polymer in question.

General

The string method `index`

The string method `index` may be used to determine the position of the leftmost occurrence of a character in a strings.

```
>>> string = 'mississippi'
>>> character = 'i'
>>> string.index(character)
1
```

This code fragment finds the leftmost occurrence of the letter `i` at position 1 in the string `mississippi`. Note that Python indexes the positions of the characters in a string starting from 0, not starting from 1.

It should be noted that this methode might be inefficient in case the position of the character can be derived directly. In order to find the position of a character that first occurs at position p, the index method must

traverse the first $p-1$ positions of the string. For a character that does not occur in the string, the string method index must even travers all characters in the string before it can decide that the character was not observed.