A run-length-compressed FIFO buffer

- Total points: 11 (divided among 8 tasks)
- Number of submissions allowed: 6

Introduction

We want to implement a buffer: an object that you can use for temporary storage. You `add` things to it, one at a time, and later you can `get` them out, one at a time. We want it to be FIFO (first-in-first-out): the items come out in the same order in which they were added.

For simplicity, we are only going to use this buffer to store single characters. So for example you can `add` the characters `'a'`, then `'a'` again, then `'b'`. If you `get` at this point you obtain `'a'`. If you `get` again you obtain another `'a'`. Now the buffer only contains the `'b'`. You could `get` it right away, or you could `add` more characters after that.

Our buffer will not have a maximum size. But it will use a trick to save storage memory: internally, the data will be represented using run-length encoding. Here is an example that should clarify what it means: the data `"aabbbbbcaa"` is represented as `[('a', 2), ('b', 5), ('c', 1), ('a', 2)]'`. As you can see, we store a list of pairs, each pair having a character and a counter.

Whenever we `add` a new character, if it is the same as the last one that we had added, we will just update a counter. In the previous example, adding an `'a'` would result in the buffer becoming `[('a', 2), ('b', 5), ('c', 1), ('a', 3)]'`. If, instead, we `add` a different character than the last one, we will create a new tuple, with its counter initialized to `1`, and append it at the end. In the previous example, adding a `'b'` would result in the buffer becoming `[('a', 2), ('b', 5), ('c', 1), ('a', 2), ('b', 1)]'`.

Conversely, whenever we `get` a character, we look at the first tuple. We read the character, and decrease the counter by 1. But if the counter becomes 0, we remove the tuple altogether. We then return the character that we had read. If we try to `get` from an empty buffer we get an error.

We can observe that using this scheme the buffer will always respect some constraints: 1) the second entry (the counter) in each tuple will always be a positive integer; 2) the first entry in each tuple will always be a single character (a length-1 string); 3) two adjacent tuples in the buffer will never have the same character as their first element.

Tasks

Each task is worth 1 or 2 points. They are all tested independently, but task 3 partially depends on task 2.

The test file `"runtest.py"` contains code that tests all tasks sequentially. You will keep getting errors until you complete all the tasks. But after each successful task, it prints a "test ok" message, so that you know you can proceed to the next task. If you can't complete a task, you can comment out the corresponding section of the test file.

1. **[1 point]** Write a class called `RunLengthBuffer` and a constructor that takes no arguments and simply initializes an internal attribute called `buffer` to an empty list.
2. **[2 points]** Write an internal method called `_check` (notice the initial underscore). This method must take no arguments, and it must check that the internal `buffer` attribute respects the first two constraints described in the last paragraph of the introduction. Thus, it must scan all the elements of the `buffer` list, and for each tuple check the type and value of its first and second elements. It must use `assert` statements when performing the checks.
3. **[1 point]** Expand the `_check` method to also verify the last constraint: at the end of the first checks, scan the buffer again and ensure that successive tuples have different characters. Also use `assert` statements.
4. **[2 points]** Write a method that allows to call the `len` built-in function on a `RunLengthBuffer` object. It should return the number of items in the buffer, i.e. the number of `get` operations that you could perform. For example, if the buffer contains `[('a', 2), ('b', 5), ('c', 1)]` the length should be `2+5+1=8`.
5. **[2 points]** Write a method that allows to print or represent as a string the `RunLengthBuffer` object. The representation should be in non-compressed form, as follows: suppose that the buffer is `[('a', 2), ('b', 5), ('c', 1), ('a',`

`2)]`. This should be represented in string form as `"RLB[aabbbbbcaa]"`. So, in general: the prefix `"RLB"`, and then the characters that you would obtain with `get`, enclosed within square brackets. Of course, an empty buffer would just be represented as `"RLB[]"`.

6. **[1 point]** Write the `add` method as described in the introduction. It should take one argument, the character to add, and return `None`. It should check this argument and raise `TypeError` if it's not a string and a `ValueError` if it's a string of the wrong length. **Tips:** Remember that 1) the buffer might be empty; 2) you want to start by looking at the last tuple, if present; 3) tuples are immutable: you cannot update just the second entry of a tuple, however you can create a new tuple with the first entry equal to the previous case and an updated second entry.
7. **[1 point]** Write the `get` method as described in the introduction. It should take no arguments, and return a character. If the buffer is empty, raise an `Exception`. If not, it should update the first tuple of the buffer, or remove it, depending on the situation. When you update it, keep in mind that tuples are immutable (see the third tip in the previous point).
8. **[1 point]** Write a method that allows you to check whether some item is contained in the buffer, such that you can write for example `'a' in b` or `'c' not in b` when `b` is a `RunLengthBuffer`. It should only be possible to get `True` when testing single characters, in all other cases you should simply return `False`. For example, `'aaa' in b` should always return `False` even if the buffer contains the data `"aaaabbb"`.

General rules

In all methods, you can use whatever constructs you want. Only the behavior matters, as long as you follow the indications.

The class methods must not print anything. When the file `"ex.py"` is loaded nothing should be printed. Do not write anything else in the file, just the class(es) and, if needed, `import` statements.

You can do tests by typing `python runtest.py` in the terminal, or pressing the `RUN` button. Edit the file `"runtest.py"` as much as you want when you test. That file is for your own use and it's not used when grading.

Remember that copies of the original files are always present in the `"startercode"` directory, in case you need them.

As for examples of how everything should work, refer to the test script.