**MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

**Laboratory work 4:**

# Topic: Regular expressions

**Course: Formal Languages & Finite Automata**

**Author:Ostafi Eugen, FAF-222**

**Chișinău, 2024**

**THEORY**

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages.     The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

**OBJECTIVES**

1. Write and cover what regular expressions are, what they are used for;
2. Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:

a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).

b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

c. **Bonus point**: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

**IMPLEMENTATION DESCRIPTION**

**1.** **Expression pattern to tokens**

****

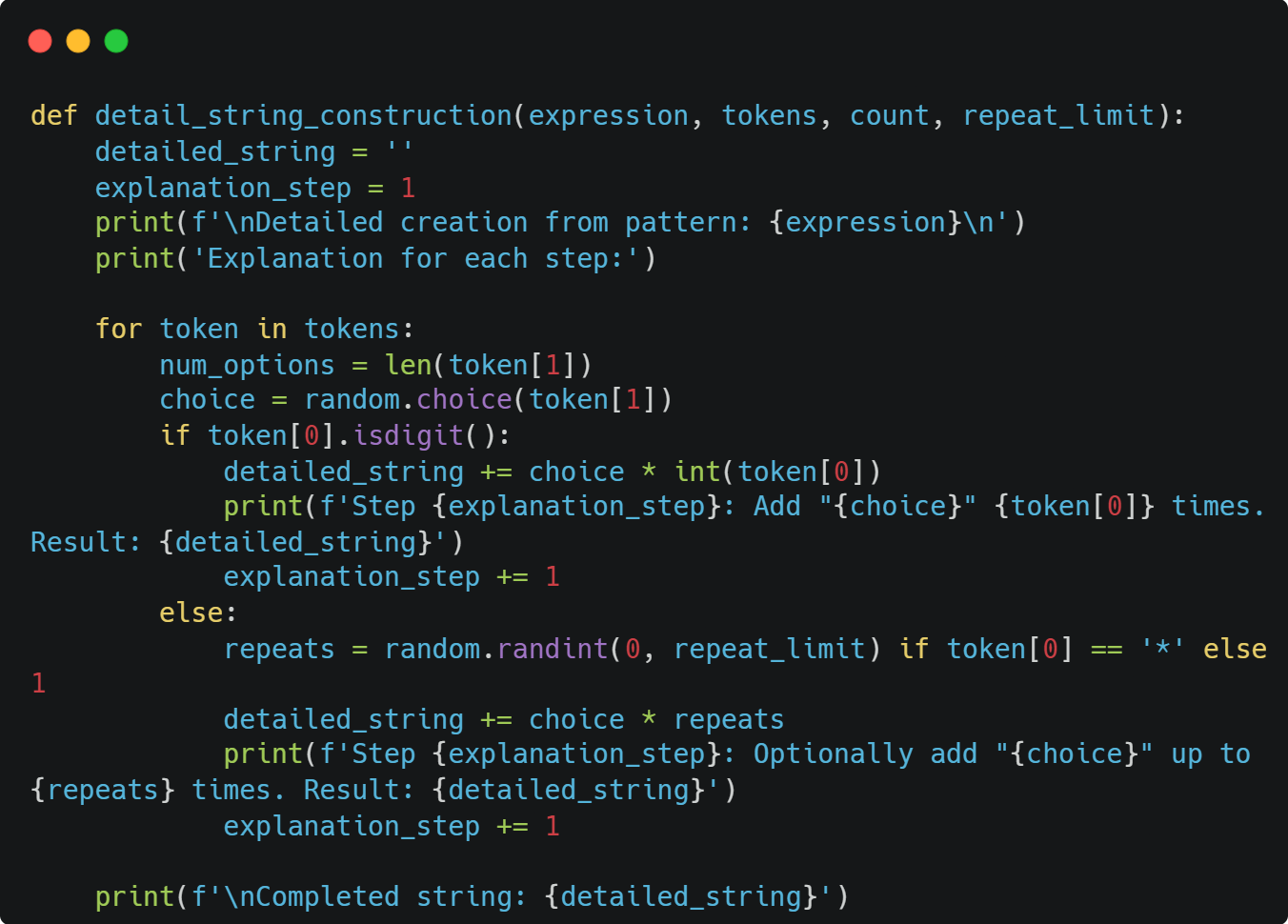
The tokenize\_pattern function converts a regular expression into a list of tokens, each representing a sequence of characters or a choice of characters along with how many times they can appear. It interprets special characters like \*, +, ?, and specific repetition counts given by ^ and numbers, aligning them with the appropriate character sets. This tokenization is essential for later functions to generate strings that match the provided regular expression by understanding the structure and rules of the pattern.

**2. Strings based on a sequence of tokens**



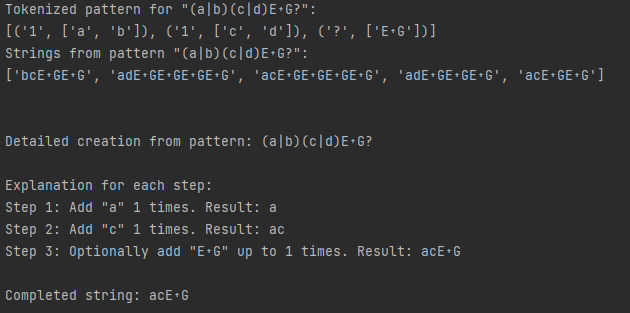
The build\_strings function generates a list of strings based on a sequence of tokens that represent a regular expression. It determines how many times a character or a group of characters should be repeated according to the token rules (like exactly once, optionally, one or more times, or a specific number of times) and composes the strings accordingly. It uses randomness to vary the output within the constraints of the pattern and repeats this process for the number of strings (count) specified.

**3. Explanation for each step**



The detail\_string\_construction function takes a regular expression, its tokenized form, and parameters to control the generation process, and then it creates a single string. As it builds the string, it provides a detailed explanation at each step, outlining how each segment of the pattern contributes to the final result. The function uses randomness within the defined limits to create a string that matches the pattern and documents the decision process in an easily understandable manner.

**Output**



Output shows the process of generating strings that match a specific regular expression pattern (a|b)(c|d)E?G?. It begins with the tokenized pattern, which breaks down the regex into manageable parts. Then, it displays a list of strings that were generated based on this pattern, demonstrating the variability allowed within the regex's rules. Lastly, it details the construction of one such string step by step, explaining the inclusion of each character according to the regex and resulting in a final generated string.

**Conclusion**

Code and its output offer a comprehensive view into the mechanics of regular expression processing and string generation in Python. Initially, the tokenize\_pattern function deconstructs a given regular expression into tokens, which are pairs that associate specific characters or groups of characters with their designated repetition rules. This tokenization is crucial for interpreting the regex in a way that a string generator can utilize.

The build\_strings function then takes these tokens and employs randomness within the bounds defined by the tokens to produce a series of strings that adhere to the original pattern. The output showcases the variety of strings generated from a single regex, highlighting the flexibility of the pattern.

Further clarity is provided by the detail\_string\_construction function, which not only generates a string following the regex rules but also provides a verbose, step-by-step narrative of the generation process. This narrative delineates how each token influences the string being built, making the often opaque process of regex matching more transparent and understandable.

Overall, this set of functions and their output demonstrate a powerful method for regex-driven development, testing, or educational purposes, where understanding and visualizing how patterns translate into potential strings is essential. It reveals the underlying logic of regex string generation, demystifying the complexity of regex patterns through a clear and methodical approach.