Regular Expressions

Course: Formal Languages & Finite Automata

Author: Dimbitchi Sergiu

Theory

Formal Language Theory: Regular expressions are closely tied to formal language theory, which deals with the study of languages in a mathematical context. In this theory, languages are described by sets of strings over some alphabet. Regular expressions are used to describe regular languages, which are a subset of formal languages defined by certain properties, such as being expressible by finite automata.

Syntax: Regular expressions have a specific syntax that defines the patterns they represent. This syntax typically includes a combination of literal characters, metacharacters, and special symbols that denote repetitions, alternatives, character classes, and more. Different implementations of regular expressions may have slightly different syntaxes, but the core concepts remain consistent.

Automata Theory: Regular expressions are closely related to finite automata, which are abstract machines used to recognize patterns in strings. There is a formal equivalence between regular expressions and finite automata, meaning that for every regular expression, there exists a corresponding finite automaton, and vice versa. This relationship forms the basis for many algorithms used to process regular expressions efficiently.

Operations: Regular expressions support various operations for constructing complex patterns from simpler ones. These operations include concatenation, alternation, repetition, and grouping. Concatenation allows combining multiple patterns sequentially, alternation allows specifying alternative choices, repetition allows specifying the number of times a pattern can occur, and grouping allows applying operations to subpatterns as a unit.

Matching: The primary operation performed with regular expressions is pattern matching, which involves searching a string for occurrences of a pattern described by the regular expression. Matching algorithms typically involve traversing the input string and the regular expression simultaneously, using techniques such as backtracking, finite automata simulation, or efficient data structures like DFA (Deterministic Finite Automaton) or NFA (Nondeterministic Finite Automaton).

Applications: Regular expressions find applications in various domains, including text processing, string searching, lexical analysis (tokenization), data validation, and pattern extraction. They are widely used in programming languages, text editors, command-line utilities, database systems, and networking protocols.

.

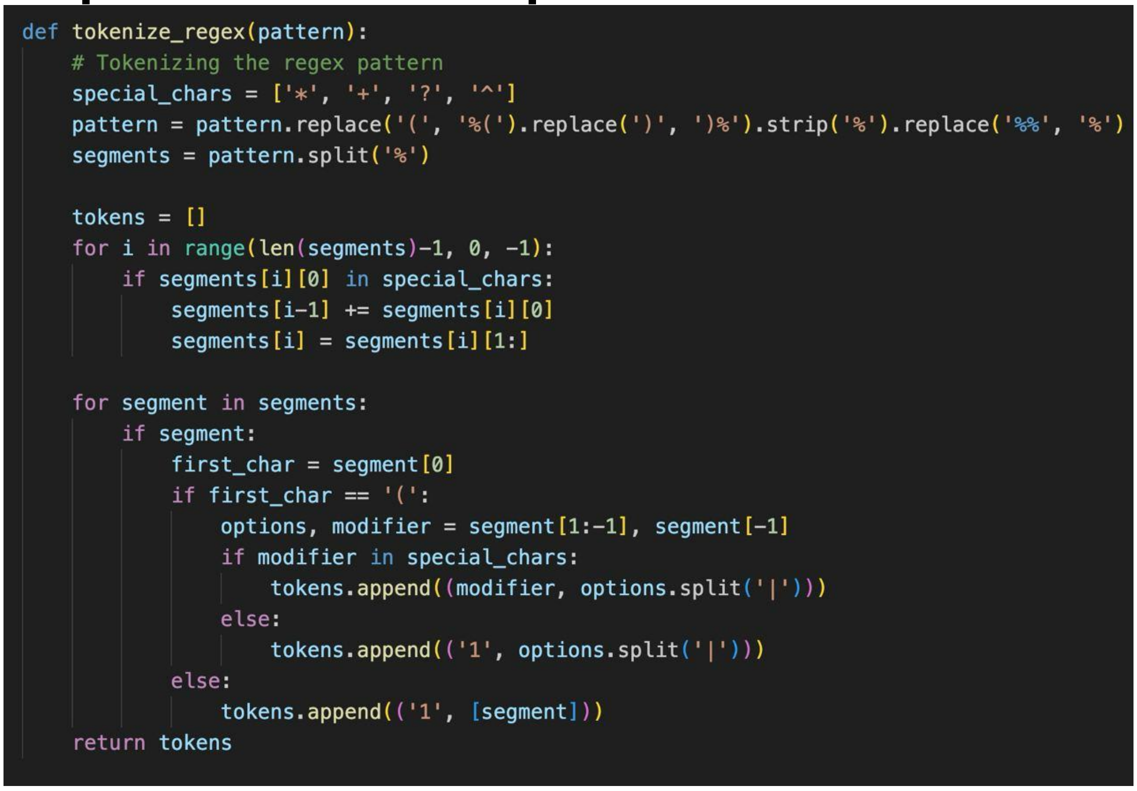
Objectives:

- Write and cover what regular expressions are, what they are used for;
- 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:

1. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).
2. 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);
3. Bonus point: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

- Write a good report covering all performed actions and faced difficulties.
-

Implementation description:

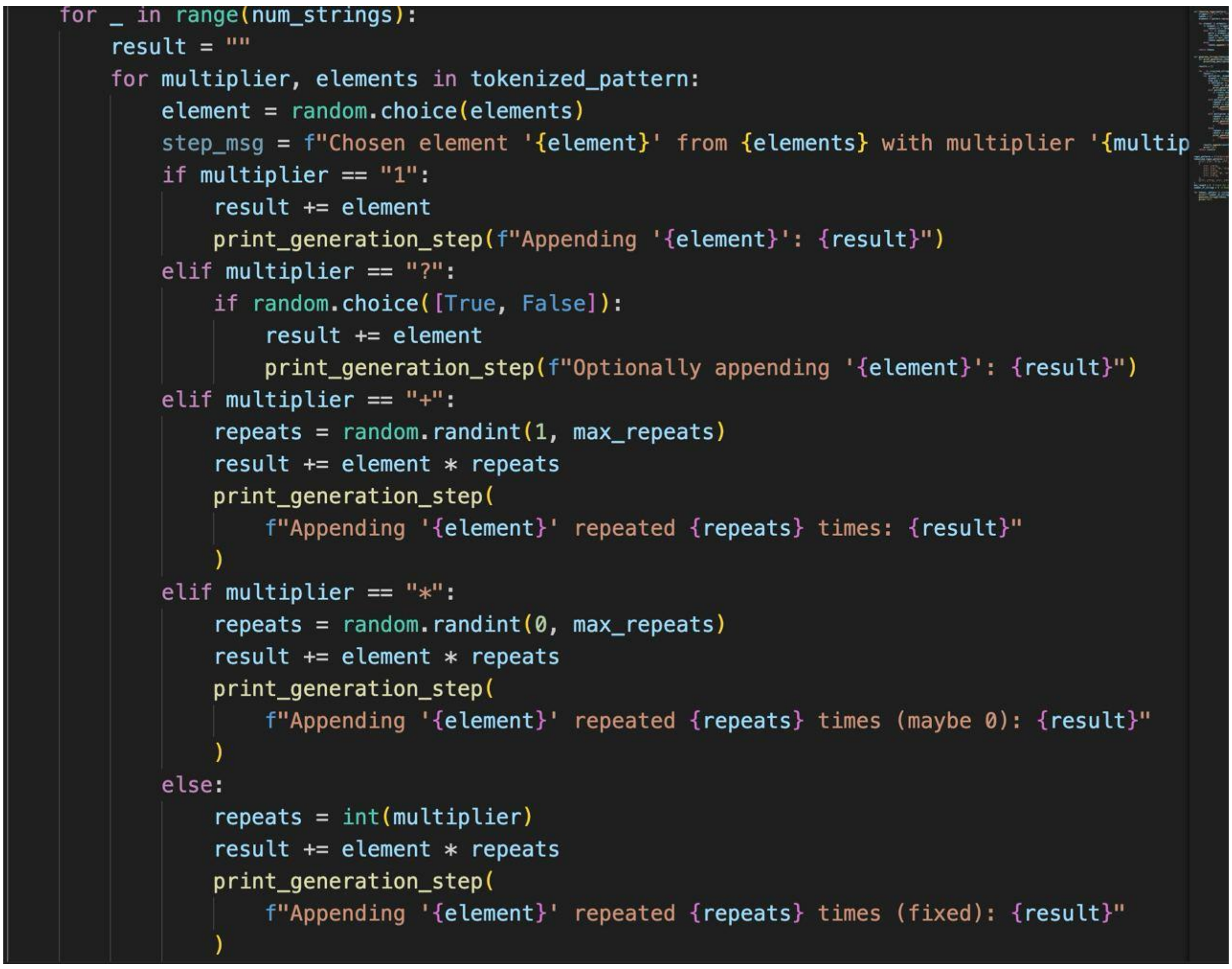


```
def tokenize_regex(pattern):
    # Tokenizing the regex pattern
    special_chars = ['*', '+', '?', '^']
    pattern = pattern.replace('(', '%(').replace(')', ')%').strip('%').replace('%%', '%')
    segments = pattern.split('%')

    tokens = []
    for i in range(len(segments)-1, 0, -1):
        if segments[i][0] in special_chars:
            segments[i-1] += segments[i][0]
            segments[i] = segments[i][1:]

    for segment in segments:
        if segment:
            first_char = segment[0]
            if first_char == '(':
                options, modifier = segment[1:-1], segment[-1]
                if modifier in special_chars:
                    tokens.append((modifier, options.split('|')))
                else:
                    tokens.append(('1', options.split('|')))
            else:
                tokens.append(('1', [segment]))
    return tokens
```

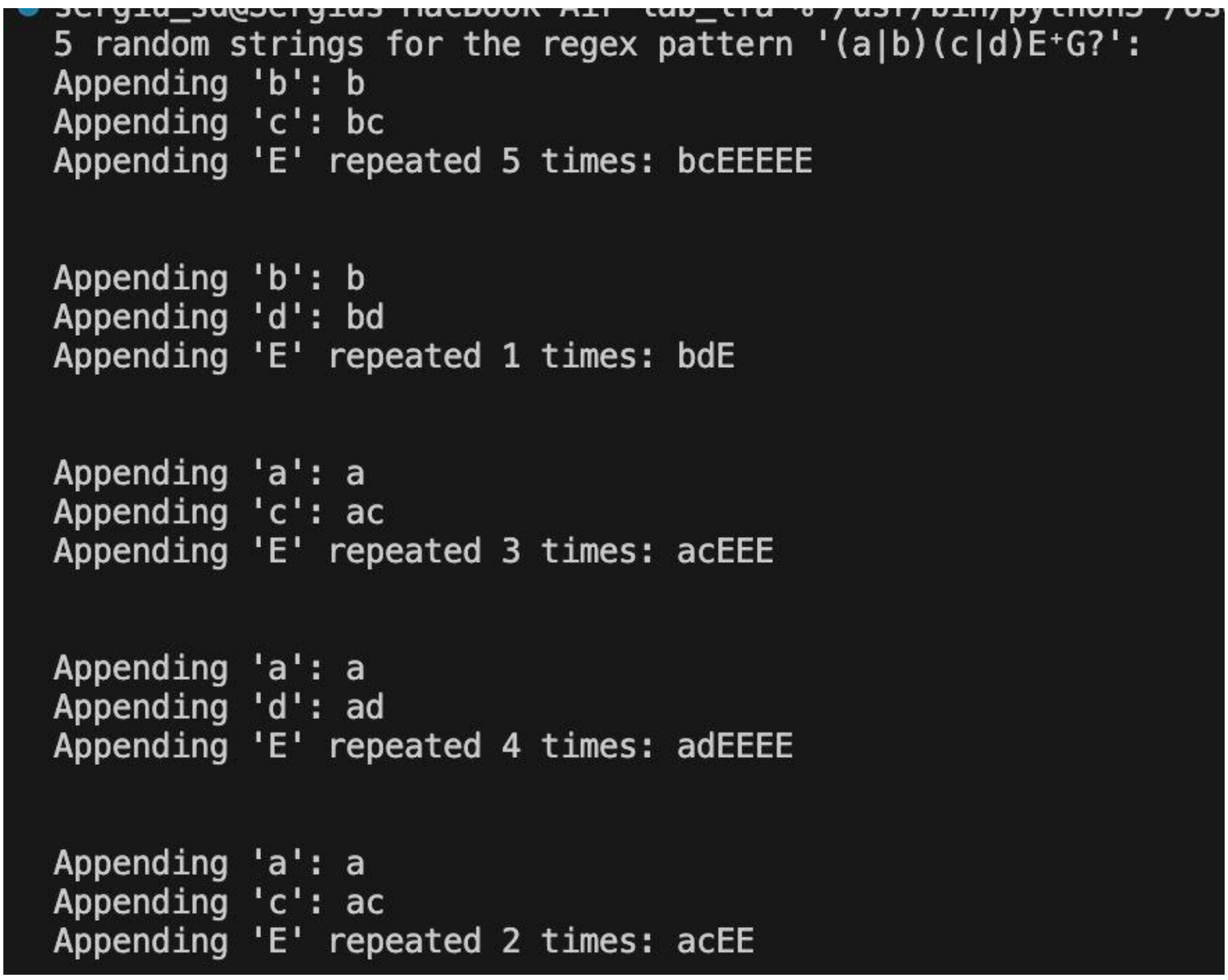
This function `tokenize_regex` is designed to tokenize a regular expression pattern into components, where each token represents a segment of the regex pattern. The code handles special characters commonly used in regex, such as '*', '+', '?', and '|'. It first processes the pattern by splitting it at parentheses and then recombines parts of the pattern that are modified by special characters. Afterward, it iterates through the segments, creating tokens that include a modifier and the options available for that segment of the pattern. Finally, the function returns a list of tokens, each a tuple indicating the modifier and the corresponding segment of the original pattern.



```python
    for _ in range(num_strings):
        result = ""
        for multiplier, elements in tokenized_pattern:
            element = random.choice(elements)
            step_msg = f"Chosen element '{element}' from {elements} with multiplier '{multip
            if multiplier == "1":
                result += element
                print_generation_step(f"Appending '{element}': {result}")
            elif multiplier == "?":
                if random.choice([True, False]):
                    result += element
                    print_generation_step(f"Optionally appending '{element}': {result}")
            elif multiplier == "+":
                repeats = random.randint(1, max_repeats)
                result += element * repeats
                print_generation_step(
                    f"Appending '{element}' repeated {repeats} times: {result}"
                )
            elif multiplier == "*":
                repeats = random.randint(0, max_repeats)
                result += element * repeats
                print_generation_step(
                    f"Appending '{element}' repeated {repeats} times (maybe 0): {result}"
                )
            else:
                repeats = int(multiplier)
                result += element * repeats
                print_generation_step(
                    f"Appending '{element}' repeated {repeats} times (fixed): {result}"
                )
```

This is a part of the function `generate_strings` that generates a number of strings based on the tokenized pattern, where each token contains an element and a multiplier indicating how the element should be used. For each string to be generated, the code iterates over each token, randomly chooses an element from those available, and then appends the element to the result string according to the multiplier: '1' for direct addition, '?' for a 50% chance addition, '+' for one or more repetitions, and '' for zero or more repetitions. The exact number of repetitions for '+' and '' is determined randomly within the bounds of 1 to max_repeats and 0 to max_repeats, respectively. There is also handling for fixed multipliers as integers. For each step, a helper function print_generation_step is called to output the action being taken.

The output :



```
sergiu_sd@Sergius MacBook Air lab_tfa % /usr/bin/python3 /Us
5 random strings for the regex pattern '(a|b)(c|d)E+G?':
Appending 'b': b
Appending 'c': bc
Appending 'E' repeated 5 times: bcEEEEE


Appending 'b': b
Appending 'd': bd
Appending 'E' repeated 1 times: bdE


Appending 'a': a
Appending 'c': ac
Appending 'E' repeated 3 times: acEEE


Appending 'a': a
Appending 'd': ad
Appending 'E' repeated 4 times: adEEEE


Appending 'a': a
Appending 'c': ac
Appending 'E' repeated 2 times: acEE
```

Conclucions :

In this lab, we learned how to use Python's regular expression (regex) module to create strings that match specific patterns. Regular expressions help us handle text in a very efficient way, allowing us to make and check strings against certain rules quickly.

We tried different patterns and used them to make strings. This practice helped us get better at using regular expressions and showed us how useful they are for tasks like checking data and preparing it for use in programs.

Using regular expressions can be very handy, especially when we need to test software with many different types of data. The work we did in the lab today proved that knowing how to use regular expressions can make programming tasks much easier.

Overall, this lab showed us that regular expressions are an important tool for any programmer who wants to work more effectively with text data. The skills we've developed here will be useful for many future projects.

References

1. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
2. https://medium.com/@pythonmembers.club/building-a-lexer-in-python-a-tutorial-3b6de161fe84
3. https://www.youtube.com/watch?v=BI3K-ME3L74
4. Theory lessons from else