Regular expressions

Course: Formal Languages & Finite Automata

Author: Popov Nichita

Variant: 2

Theory

Regular expressions (regex) are essential in computing for pattern matching and text manipulation. They originated from formal language theory and enable identifying text strings that match specified patterns. Regex uses a set of characters to define these patterns, making it possible to perform complex search, validation, and text transformation operations efficiently.The syntax of regex includes literals, character classes, quantifiers, and position anchors to craft patterns for matching a wide range of strings. Despite their versatility, regex can be complex due to their concise syntax and variations across different programming environments. This complexity, however, is offset by their powerful capability to automate and streamline text processing tasks. Regular expressions are a critical tool for programmers, data scientists, and system administrators, enhancing text data interaction and analysis. Mastery of regex opens up vast possibilities for text manipulation, making it a valuable skill in computer science.

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:
    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

- Regex Class: The Regex class serves as the foundation of this tool, encapsulating methods that illustrate the mechanics of regular expressions through systematic pattern generation and explanation.

This class is designed not just to generate combinations but to explain the sequential processing of regex components, thereby demystifying complex regex patterns.

Core Functionalities

1. Methodological Explanation: Through the `explain_process` method, the class offers a step-by-step breakdown of how each part of a regex pattern is processed. This pedagogical approach aids users in understanding the logic behind pattern matching and the significance of regex syntax elements.

```
def explain_process(self, steps):
"""
Explain the sequence of processing the regular expression.

Parameters:
- steps: A list of strings, each describing a step in
        processing the regex.
"""
print("\nProcessing Sequence:")
for i, step in enumerate(steps, start=1):
    print(f"{i}. {step}")
print("\n")
```

1. Dynamic Pattern Generation: The `generate_combinations` method leverages Python's `itertools.product` to dynamically produce all conceivable string combinations from specified pattern components. This method underscores the power of regex in capturing a wide array of string variations with concise syntax.

```
def generate_combinations(self, parts, description, steps):
    """
    Generalized function to generate and print all
        combinations based on the given parts.

    Parameters:
    - parts: A list of lists, where each inner list
        represents possible variations of a regex part.
    - description: A string describing the regular expression
        being processed.
    - steps: A list of strings, each describing a step in
        processing the regex.
    """
    self.explain_process(steps)
```

```
    print(f"=========================================================\
    print(f"All possible combinations for {description}")
    print(f"\n=========================================================
    for combination in itertools.product(*parts):
        print(f''.join(combination))
```

Method Descriptions

1. `first_regex` Method: Constructs a multifaceted regex pattern, highlighting optional characters, fixed repetitions, alternative selections, and quantified occurrences. This method exemplifies the nuanced control regex offers over string matching criteria, providing a broad spectrum of potential matches from a single pattern description.

```
def first_regex(self):
    parts = [
        ['', 'M'],  # M?
        ['NN'],  # N^2
        [''.join(x) for x in itertools.product('OP', repeat=3)],
        # (O|P)^3
        [''.join(['Q']*i) for i in range(6)],  # Q^*
        [''.join(['R']*i) for i in range(1, 6)],  # R^+
    ]
    steps = [
        "Process 'M?' - M is optional",
        "Process 'N^2' - N appears exactly twice",
        "Process '(O|P)^3' - Either O or P appears exactly
        three times in any combination",
        "Process 'Q^*' - Q appears any number of times up to
        5",
        "Process 'R^+' - R appears at least once and up to 5
        times"
    ]
    self.generate_combinations(parts, "M? N^2 (O|P)^3 Q^*
        R^+", steps)
```

1. `second_regex` Method: This method showcases a different aspect of regex capabilities, focusing on character sets, quantifiers, and choice operators. It's particularly useful for demonstrating how regex can be used to define patterns with varying character repetitions and selections, essential for tasks like parsing and data extraction.

```
def second_regex(self):
    parts = [
```

```
        [''.join(x) for x in itertools.product('XYZ',
        repeat=3)],  # (X|Y|Z)^3
        [''.join(['8']*i) for i in range(1, 6)],  # 8^+
        ['9', '0'],  # (9|0)
    ]
    steps = [
        "Process '(X|Y|Z)^3' - X, Y, or Z appears exactly
        three times in any combination",
        "Process '8^+' - 8 appears at least once and up to 5
        times",
        "Process '(9|0)' - Either 9 or 0 appears exactly
        once"
    ]
    self.generate_combinations(parts, "(X|Y|Z)^3 8^+ (9|0)",
        steps)
```

1. `third_regex` Method: Emphasizes the flexibility of regex in managing optional elements, repetitions, and specific character choices. It's tailored to exhibit how subtle variations in regex patterns can significantly alter the scope of matched strings, illustrating the importance of precise pattern crafting.

```
def third_regex(self):
    parts = [
        ['H', 'i'],  # (H|i)
        ['J', 'K'],  # (J|K)
        [''.join(['L']*i) for i in range(6)],  # L^*
        ['', 'N'],  # N?
    ]
    steps = [
        "Process '(H|i)' - Either H or i appears exactly
        once",
        "Process '(J|K)' - Either J or K appears exactly
        once",
        "Process 'L^*' - L appears any number of times up to
        5",
        "Process 'N?' - N is optional"
    ]
    self.generate_combinations(parts, "(H|i) (J|K) L^* N?",
        steps)
```

- `main.py` Driver Script: This script acts as the entry point for regex pattern exploration, sequentially invoking the `Regex` class methods to generate and explain multiple regex patterns. Its straightforward structure makes it easy for users to add new patterns for exploration,

enhancing the tool’s versatility and adaptability to different learning or development needs.

Results

Here is the output for method `third_regex` including explanation, all possible combinations and remark about which regex is being processed:

```
Processing Sequence:
1. Process '(H|i)' - Either H or i appears exactly once
2. Process '(J|K)' - Either J or K appears exactly once
3. Process 'L^*' - L appears any number of times up to 5
4. Process 'N?' - N is optional


============================================================

All possible combinations for (H|i) (J|K) L^* N?

============================================================

HJ
HJN
HJL
HJLN
HJLL
HJLLN
HJLLL
HJLLLN
HJLLLL
HJLLLLN
HJLLLLL
HJLLLLLN
HK
HKN
HKL
HKLN
HKLL
HKLLN
HKLLL
HKLLLN
HKLLLL
HKLLLLN
HKLLLLL
HKLLLLLN
```

```
iJ
iJN
iJL
iJLN
iJLL
iJLLN
iJLLL
iJLLLN
iJLLLL
iJLLLLN
iJLLLLL
iJLLLLLN
iK
iKN
iKL
iKLN
iKLL
iKLLN
iKLLL
iKLLLN
iKLLLL
iKLLLLN
iKLLLLL
iKLLLLLN
```

Conclusion

This lab has highlighted the indispensable role of regular expressions in text manipulation and pattern matching, integral to various computing tasks. Through the implementation and exploration of complex regex patterns, we've not only met the lab's educational objectives but also gained a profound understanding of regex's capabilities and applications. The process of generating valid combinations and explaining the processing sequence of regex patterns provided practical insights into their power and flexibility. While focused on specific regex constructs, the skills and knowledge acquired are transferable to a broad array of computing contexts. This experience not only fulfills the lab's goals but also sets the stage for further exploration into advanced text processing and pattern recognition challenges.

References

- https://www.youtube.com/watch?v=sXQxhojSdZM
- https://www.youtube.com/watch?v=upu_TeZImN0&t
- https://www.youtube.com/watch?v=dQw4w9WgXcQ