Chapter 7: Advanced Regular Expressions Tutorial

October 09, 2025

1 Introduction

This tutorial covers Chapter 7, "Regular Expressions, Part II" from the book "Supercharged Python: Take Your Code to the Next Level". It explores advanced regex grammar, noncapture groups, greedy versus non-greedy matching, look-ahead features, and more, with examples and outputs.

2 Summary of Advanced Regex Grammar (7.1)

Describing advanced constructs in regex, enhancing pattern matching capabilities.

- Supports complex pattern definitions.
- Example: Matching dates 2/2/4.

3 Noncapture Groups (7.2)

Noncapture groups allow grouping without capturing, improving performance.

3.1 The Canonical Number Example (7.2.1)

Illustrating with numbers.

```
import re
text = "123 456 789"
pattern = r'(?:\d{3})\s'
result = re.findall(pattern, text)
print(result)  # Output: ['123 ', '456 ']
```

3.2 Fixing the Tagging Problem (7.2.2)

Addressing issues in tagged data.

```
text = "<tag>value</tag>"
pattern = r'<(?:/)?tag>'
result = re.sub(pattern, '', text)
print(result)  # Output: value
```

4 Greedy Versus Non-Greedy Matching (7.3)

Explaining how greediness affects matches.

```
text = "aabbcc"
pattern_greedy = r'a.*c'
pattern_nongreedy = r'a.*?c'
print(re.match(pattern_greedy, text).group())  # Output: aabbc
print(re.match(pattern_nongreedy, text).group())  # Output: aac
```

5 The Look-Ahead Feature (7.4)

Using look-ahead to assert conditions without consuming characters.

```
text = "cost123 price456"
pattern = r'\w+(?=\d{3})'
result = re.findall(pattern, text)
print(result)  # Output: ['cost', 'price']
```

6 Checking Multiple Patterns (Look-Ahead) (7.5)

Combining multiple look-aheads for complex checks.

```
text = "pass123"
pattern = r'^(?=.*[a-z])(?=.*[0-9]).+$'
result = bool(re.match(pattern, text))
print(result)  # Output: True
```

7 Negative Look-Ahead (7.6)

Ensuring patterns are not followed by certain sequences.

```
text = "file.txt file.doc"
pattern = r'file\.(?!doc)\w+'
result = re.findall(pattern, text)
print(result)  # Output: ['file.txt']
```

8 Named Groups (7.7)

Assigning names to capture groups for clarity.

```
text = "Date: 2025-10-09"
pattern = r'(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})'
match = re.match(pattern, text.split(': ')[1])
print(match.group('year'))  # Output: 2025
```

9 The "re.split" Function (7.8)

Splitting strings with regex patterns.

```
text = "apple,orange;banana"
pattern = r'[;,]'
result = re.split(pattern, text)
print(result)  # Output: ['apple', 'orange', 'banana']
```

10 The Scanner Class and the RPN Project (7.9)

Introducing Scanner for tokenizing and Reverse Polish Notation (RPN).

```
import re
scanner = re.Scanner([(r'\d+', lambda x: int(x)), (r'\+', lambda
    x: '+')])
result, remainder = scanner.scan('3 + 4')
print(result)  # Output: [3, '+', 4]
```

11 RPN: Doing Even More With Scanner (7.10)

Enhancing RPN with additional operations.

```
def evaluate_rpn(tokens):
    stack = []
    for token in tokens:
        if isinstance(token, int):
            stack.append(token)
        else:
            b, a = stack.pop(), stack.pop()
            stack.append(a + b if token == '+' else a * b)
    return stack[0]
print(evaluate_rpn([3, 4, '+']))  # Output: 7
```

12 Conclusion

This tutorial encapsulates key regex concepts with practical examples, aiding in mastering advanced pattern matching in Python.