Supercharged Python: Chapter 6 - Regular Expressions, Part I

Tutorial by Dr. Ayes Chinmay

October 09, 2025

Introduction to Regular Expressions

Introducing the concept of regular expressions, explaining their role in pattern matching within Python.

Increasingly, the most sophisticated computer software deals with patternsfor example, speech patterns and the recognition of images. This chapter deals with the former: how to recognize patterns of words and characters. Although you can't construct a human language translator with these techniques alone, they are a start.

That's what regular expressions are for. A regular expression is a pattern you specify, using special characters to represent combinations of specified characters, digits, and words. It amounts to learning a new language, but it's a relatively simple one, and once you learn it, this technology lets you do a great deal in a small spacesometimes only a statement or twothat would otherwise require many lines.

- **Note**: Regular expression syntax has a variety of flavors. The Python regular-expression package conforms to the Perl standard, which is an advanced and flexible version.

1 Section 6.1: Introduction to Regular Expressions

Detailing the basics of regular expressions and their application in Python.

A regular expression can be as simple as a series of characters that match a given word. For example, the following pattern matches the word "cat"; no surprise there.

```python
import re
pattern = r'cat'
text = 'cat'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

But what if you wanted to match a larger set of words? For example, let's say you wanted to match the following combination of letters:

- Match a "c" character.
- Match any number of "a" characters, but at least one.

- Match a "t" character.

Here's the regular expression that implements these criteria: `ca+t`.

1.1 Subsection 6.1.1: Literal vs. Special Characters

Explaining the distinction between literal and special characters.

With regular expressions (as with formatting specifiers in the previous chapter), there's a fundamental difference between literal and special characters. Literal characters, such as "c" and "t" in this example, must be matched exactly, or the result is failure to match. Most characters are literal characters, and you should assume that a character is literal unless a special character changes its meaning. All letters and digits are, by themselves, literal characters; in contrast, punctuation characters are usually special; they change the meaning of nearby characters.

The plus sign (+) is a special character. It does not cause the regular-expression processor to look for a plus sign. Instead, it forms a subexpression, together with "a" that says, "Match one or more 'a' characters." The pattern `ca+t` therefore matches any of the following:

- `cat`
- `caat`
- `caaat`
- `caaaat`

```python
import re
pattern = r'ca+t'
texts = ['cat', 'caat', 'caaaat', 'ct']
for text in texts:
    if re.match(pattern, text):
        print(f"Match found for '{text}'")
    else:
        print(f"No match for '{text}'")
```

Output:

```
Match found for 'cat'
Match found for 'caat'
Match found for 'caaaat'
No match for 'ct'
```

1.2 Subsection 6.1.2: Escaping Special Characters

Describing how to match an actual plus sign using escape sequences.

What if you wanted to match an actual plus sign? In that case, you'd use a backslash (to create an escape sequence. One of the functions of escape sequences is to turn a special character back into a literal character. So the following regular expression matches `ca+t` exactly: `cat`.

```python
import re
pattern = r'ca\+t'
text = 'ca+t'
if re.match(pattern, text):
    print("Match found!")
else:
    print("No match!")
```

Output: Match found!

1.3 Subsection 6.1.3: Multiplication Operator (*)

Introducing the multiplication sign (*) for zero or more occurrences.

Another important operator is the multiplication sign (*), which means "zero or more occurrences of the preceding expression." Therefore, the expression `ca*t` matches any of the following:

- `ct`
- `cat`
- `caat`
- `caaaat`

```python
import re
pattern = r'ca*t'
texts = ['ct', 'cat', 'caat', 'caaaat']
for text in texts:
    if re.match(pattern, text):
        print(f"Match found for '{text}'")
```

Output:

```
Match found for 'ct'
Match found for 'cat'
Match found for 'caat'
Match found for 'caaaat'
```

1.4 Subsection 6.1.4: Parsing Expressions

Breaking down the syntax of regular expressions.

You can break this down syntactically, as shown in Figure 6.1. The literal characters "c" and "t" each match a single character, but a* forms a unit that says, "Match zero or more occurrences of 'a'."

2 Section 6.2: A Practical Example: Phone Numbers

Providing a practical application of regular expressions.

Suppose you want to write a verification function for phone numbers. We might think of the pattern as follows, in which # represents a digit:

```
###-###-####
```

With regular-expression syntax, you'd write the pattern this way:

```
\d\d\d-\d\d\d-\d\d\d\d
```

2.1 Subsection 6.2.1: Digit Matching with `\d`

Explaining the use of `\d` for digit matching.

In this case, the backslash (continues to act as the escape character, but its action here is not to make "d" a literal character but to create a special meaning. The subexpression . means to match any one-digit character. Another way to express a digit character is to use the following subexpression:

```
[0-9]
```

However, . is more succinct.

```python
import re
pattern = r'\d\d\d-\d\d\d-\d\d\d\d'
text = '123-456-7890'
if re.match(pattern, text):
    print("Number accepted!")
else:
    print("Incorrect format!")
```

Output: Number accepted!

2.2 Subsection 6.2.2: Complete Verification Program

Presenting a full program for phone number verification.

Here's a complete Python program that implements this regular-expression pattern for verifying a telephone number.

```python
import re
pattern = r'\d{3}-\d{3}-\d{4}'
s = input('Enter tel. number: ')
if re.match(pattern, s):
    print('Number accepted.')
else:
    print('Incorrect format.')
```

Example Output (Input: 123-456-7890):

```
Enter tel. number: 123-456-7890
Number accepted.
```

Example Output (Input: 12-456-7890):

```
Enter tel. number: 12-456-7890
Incorrect format.
```

3 Section 6.3: Character Classes

Introducing character classes like [a-z] and [0-9].

```python
import re
pattern = r'[a-z]+'
text = 'hello'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

4 Section 6.4: Quantifiers

Explaining quantifiers like {n,m}.

```python
import re
pattern = r'\d{2,4}'
text = '1234'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

5 Section 6.5: Groups and Capturing

Describing the use of parentheses for grouping.

```python
import re
pattern = r'(\d{3})-(\d{3})-(\d{4})'
text = '123-456-7890'
match = re.match(pattern, text)
if match:
    print(f"Area code: {match.group(1)}")
    print(f"Prefix: {match.group(2)}")
    print(f"Line: {match.group(3)}")
```

Output:

```
Area code: 123
Prefix: 456
Line: 7890
```

6 Section 6.6: Anchors

Using ˆ and $ for start and end matching.

```python
import re
pattern = r'^\d{3}$'
text = '123'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

7 Section 6.7: Alternation

Using | for multiple patterns.

```python
import re
pattern = r'cat|dog'
text = 'dog'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

8 Section 6.8: Lookahead Assertions

Implementing positive and negative lookaheads.

```python
import re
pattern = r'\d+(?=\$)'
text = '100$'
match = re.search(pattern, text)
if match:
    print(f"Match: {match.group()}")
```

Output: Match: 100

9 Section 6.9: Lookbehind Assertions

Using lookbehinds for pattern constraints.

```python
import re
pattern = r'(?<=č)\d+'
text = 'č200'
match = re.search(pattern, text)
if match:
    print(f"Match: {match.group()}")
```

Output: Match: 200

10 Section 6.10: Modifiers

Exploring flags like re.IGNORECASE.

```python
import re
pattern = r'cat'
text = 'CAT'
if re.match(pattern, text, re.IGNORECASE):
    print("Match found!")
```

Output: Match found!

11 Section 6.11: Substitution with re.sub

Performing string substitutions.

```python
import re
text = 'Call 123-456-7890'
new_text = re.sub(r'(\d{3})-(\d{3})-(\d{4})', r'(\1) \2-\3',
   text)
print(new_text)
```

Output: Call (123) 456-7890

12 Section 6.12: Advanced Patterns

Combining multiple techniques.

```python
import re
pattern = r'(?=^\d{3}-)[0-9]{3}-[0-9]{4}$'
text = '123-4567'
if re.match(pattern, text):
    print("Match found!")
```

Output: Match found!

13 Section 6.13: Practical Exercises

Providing exercises for practice.

```python
import re
# Exercise: Validate email (simplified)
pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
text = 'user@domain.com'
if re.match(pattern, text):
    print("Valid email!")
```

Output: Valid email!

14 Summary

Summarizing the key points of regular expressions from sections 6.1 to 6.13, encouraging practice with pattern matching and real-world applications.