Regular Expressions (Regex): A Comprehensive Guide

Regular expressions (regex or regexp) are powerful patterns used to match character combinations in strings. They provide a concise and flexible means for identifying strings of text, such as particular characters, words, or patterns of characters.

Table of Contents

Basic Concepts

Regular expressions consist of a sequence of characters that define a search pattern.

Literal Characters

The simplest regex patterns match exact sequences of characters:

```
hello
```

This pattern matches the exact string “hello” in the text.

Meta Characters

These characters have special meanings in regex:

Character	Meaning
.	Matches any single character except newline

Character	Meaning
`^`	Matches beginning of string
`$`	Matches end of string
`*`	Matches 0 or more of the preceding element
`+`	Matches 1 or more of the preceding element
`?`	Matches 0 or 1 of the preceding element
`\`	Escapes special characters
`|`	Acts as an OR operator
`()`	Creates a capture group
`[]`	Defines a character class

Examples:

```
c.t         # Matches "cat", "cot", "c@t", etc.
\.          # Matches a literal period (escaped)
```

Character Classes

Character classes match any one character from a set of characters.

Basic Character Classes

```
[abc]       # Matches any one of "a", "b", or "c"
[^abc]      # Matches any character EXCEPT "a", "b", or "c"
[a-z]       # Matches any lowercase letter from "a" to "z"
[A-Z]       # Matches any uppercase letter from "A" to "Z"
[0-9]       # Matches any digit from 0 to 9
[a-zA-Z0-9] # Matches any alphanumeric character
```

Predefined Character Classes

```
\d          # Matches a digit [0-9]
\D          # Matches a non-digit [^0-9]
\w          # Matches a word character [a-zA-Z0-9_]
\W          # Matches a non-word character
\s          # Matches a whitespace character (space, tab,
newline)
\S          # Matches a non-whitespace character
```

Examples:

```
file\d+\.txt     # Matches "file1.txt", "file42.txt", etc.
[aeiou]          # Matches any vowel
\d{3}-\d{2}-\d{4} # Matches social security number format:
"123-45-6789"
```

Anchors

Anchors don't match characters but rather positions in the text.

```
^           # Matches the beginning of a string or line
$           # Matches the end of a string or line
\b          # Matches a word boundary
\B          # Matches a non-word boundary
```

Examples:

```
^Hello      # Matches "Hello" only at the beginning of a line
world$      # Matches "world" only at the end of a line
\bcat\b     # Matches the whole word "cat" but not "category"
```

Quantifiers

Quantifiers specify how many instances of a character, group, or character class must be present in the input for a match to be found.

```
*           # Matches 0 or more occurrences
+           # Matches 1 or more occurrences
?           # Matches 0 or 1 occurrence
{n}         # Matches exactly n occurrences
{n,}        # Matches n or more occurrences
{n,m}       # Matches from n to m occurrences
```

Examples:

```
colou?r     # Matches "color" or "colour"
go{2,4}gle  # Matches "google", "gooogle", "goooogle"
\d+         # Matches one or more digits
\d*         # Matches zero or more digits
```

Greedy vs. Lazy Quantifiers

By default, quantifiers are greedy, meaning they match as much as possible:

```
.*          # Greedy: Matches as much as possible
.*?         # Lazy: Matches as little as possible
```

Examples:

```
".*"        # In the text "A "quote" in a string", this matches
'"quote" in a'
".*?"       # In the same text, this matches "quote" (minimal
match)
```

Grouping and Capturing

Parentheses `()` create capture groups that can be referenced later.

```
(regex)     # Creates a capture group
(?:regex)   # Creates a non-capturing group
\1, \2, etc. # Back-references to capture groups
```

Examples:

```
(\d{3})-(\d{3})-(\d{4})  # Captures each segment of a phone
number
\b(\w+)\s+\1\b           # Matches repeated words like "the the"
```

Alternation

The vertical bar `|` acts as an OR operator.

```
cat|dog     # Matches either "cat" or "dog"
```

Examples:

```
(https?|ftp)://  # Matches "http://", "https://", or "ftp://"
(jpg|jpeg|png|gif)$  # Matches common image extensions at the end
of a string
```

Lookaround Assertions

Lookaround assertions check if a pattern exists without including it in the match.

```
(?=regex)   # Positive lookahead: Asserts regex can be matched after current position
(?!regex)   # Negative lookahead: Asserts regex cannot be matched after current position
(?<=regex)  # Positive lookbehind: Asserts regex can be matched before current position
(?<!regex)  # Negative lookbehind: Asserts regex cannot be matched before current position
```

Examples:

```
\b\w+(?=ing\b)     # Matches the word part before "ing" (e.g., "jump" in "jumping")
\b\w+(?!ing\b)     # Matches words not ending with "ing"
(?<=\$)\d+         # Matches digits that have a "$" before them
(?<!\$)\d+         # Matches digits that don't have a "$" before them
```

Common Patterns

Email Validation

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
```

This pattern matches most email addresses: - Username part: one or more letters, numbers, dots, underscores, percent signs, plus signs, or hyphens - @ symbol - Domain part: one or more letters, numbers, dots, or hyphens - TLD: two or more letters

URL Validation

```
https?://(?:www\.)?[-a-zA-Z0-9@:%._\+~#=]{1,256}\.[a-zA-Z0-9()]{1,6}\b(?:[-a-zA-Z0-9()@:%_\+.~#?&//=]*)
```

This pattern matches most URLs, including those with or without “www” and with various protocols.

Phone Number (US Format)

```
\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}
```

This pattern matches US phone numbers in formats like: - (123) 456-7890 - 123-456-7890 - 123.456.7890 - 1234567890

Date Formats

```
# MM/DD/YYYY format
\b(0?[1-9]|1[0-2])/(0?[1-9]|[12][0-9]|3[01])/\d{4}\b

# YYYY-MM-DD format (ISO)
\b\d{4}-(0?[1-9]|1[0-2])-(0?[1-9]|[12][0-9]|3[01])\b
```

Strong Password

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]
{8,}$
```

This pattern enforces: - At least 8 characters - At least one lowercase letter - At least one uppercase letter - At least one digit - At least one special character

Regex in Different Languages

JavaScript

```javascript
// Test if a pattern matches
const regex = /pattern/;
const isMatch = regex.test("string to
        test");  // Returns true or false

// Find matches
const matches = "string to search".match(/pattern/
        g);  // Returns array of matches or null

// Replace using regex
const result = "string to modify".replace(/pattern/g,
        "replacement");

// Using capture groups
const str = "John Smith";
const nameRegex = /(\w+)\s(\w+)/;
const [fullName, firstName, lastName] = str.match(nameRegex);
console.log(firstName, lastName);  // "John Smith"
```

Python

```python
import re

# Check if pattern matches
pattern = r"pattern"
```

```python
is_match = bool(re.search(pattern, "string to test"))

# Find all matches
matches = re.findall(pattern, "string to search")

# Replace using regex
result = re.sub(pattern, "replacement", "string to modify")

# Using capture groups
text = "John Smith"
match = re.search(r"(\w+)\s(\w+)", text)
if match:
    first_name = match.group(1)
    last_name = match.group(2)
    print(first_name, last_name)  # "John Smith"
```

Java

```java
import java.util.regex.*;

// Check if pattern matches
Pattern pattern = Pattern.compile("pattern");
Matcher matcher = pattern.matcher("string to test");
boolean isMatch = matcher.find();

// Find all matches
Pattern pattern = Pattern.compile("pattern");
Matcher matcher = pattern.matcher("string to search");
while (matcher.find()) {
    System.out.println(matcher.group());
}

// Replace using regex
String result = "string to modify".replaceAll("pattern",
        "replacement");

// Using capture groups
Pattern pattern = Pattern.compile("(\\w+)\\s(\\w+)");
Matcher matcher = pattern.matcher("John Smith");
if (matcher.find()) {
    String firstName = matcher.group(1);
    String lastName = matcher.group(2);
    System.out.println(firstName + " " + lastName);  // "John
        Smith"
}
```

PHP

```php
// Check if pattern matches
$isMatch = preg_match("/pattern/", "string to test");

// Find all matches
preg_match_all("/pattern/", "string to search", $matches);
print_r($matches);

// Replace using regex
$result = preg_replace("/pattern/", "replacement", "string to
        modify");

// Using capture groups
$text = "John Smith";
preg_match("/(\w+)\s(\w+)/", $text, $matches);
$firstName = $matches[1];
$lastName = $matches[2];
echo $firstName . " " . $lastName;  // "John Smith"
```

Best Practices

1. **Start Simple**: Begin with simpler patterns and gradually add complexity as needed.
2. **Test Thoroughly**: Test your regex patterns with various inputs, including edge cases.
3. **Use Tools**: Utilize regex testing tools to visualize and debug your patterns.
4. **Comment Complex Patterns**: For complex patterns, break them down with comments or use named capture groups.
5. **Consider Performance**: Avoid excessive backtracking by using efficient patterns.
6. **Avoid Overuse**: Don’t use regex for parsing structured data like HTML/XML/JSON when dedicated parsers exist.
7. **Use Anchors**: Add ^ and $ to ensure the entire string matches your pattern, not just a substring.
8. **Be Specific**: Make your patterns as specific as possible to avoid false positives.

Online Tools and Resources

1. **Regex101**: https://regex101.com - Interactive regex tester with syntax highlighting and explanation.
2. **RegExr**: https://regexr.com - Another excellent interactive testing tool.
3. **RegexPal**: https://www.regexpal.com - Simple and clean regex testing interface.
4. **Debuggex**: https://www.debuggex.com - Visual regex debugging with railroad diagrams.
5. **Regular-Expressions.info**: https://www.regular-expressions.info - Comprehensive tutorials and references.

Real-World Examples

Extracting YouTube Video ID

```
(?:youtube\.com\/(?:[^\/\n\s]+\/\S+\/|(?:v|e(?:mbed)?)\/|\S*?[?&]v=)|youtu\.be\/)([a-zA-Z0-9_-]{11})
```

This pattern extracts the 11-character YouTube video ID from various YouTube URL formats.

Parsing Log Files

```
(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}) - - \[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2} [+-]\d{4})\] "(\w+) (.*?) HTTP/\d\.\d" (\d{3}) (\d+)
```

This pattern parses common web server log entries, capturing: 1. IP address 2. Date and time 3. HTTP method 4. Requested path 5. HTTP status code 6. Response size

Validating Hexadecimal Color Codes

```
#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})
```

This pattern matches hexadecimal color codes in both 6-digit (#RRGGBB) and 3-digit (#RGB) formats.

Extracting Hashtags from Text

```
#[a-zA-Z0-9_]+
```

This pattern identifies hashtags in social media text.

Validating Credit Card Numbers

```
# Visa
^4[0-9]{12}(?:[0-9]{3})?$

# MasterCard
^5[1-5][0-9]{14}$

# American Express
^3[47][0-9]{13}$

# Discover
^6(?:011|5[0-9]{2})[0-9]{12}$
```

These patterns validate common credit card number formats for major card brands.

With regular expressions, you have a powerful tool for text processing, validation, and extraction. While they may seem intimidating at first, understanding the basic principles and practicing with examples will help you become proficient in using them effectively.