Learn Regex The Easy Way

A Guide To Learning Regular Expressions

A Document By:
Nathan Warner

June 29, 2023
Computer Science
Northern Illinois University
United States

Contents

1 Introduction

Regular expressions are powerful tools for pattern matching and text manipulation. They provide a concise and flexible way to describe and search for patterns in strings.

A regular expression, often abbreviated as regex, is a sequence of characters that defines a search pattern. This pattern can be used to match, locate, or replace specific portions of text.

Regular expressions are widely used in various programming languages, text editors, and command-line tools. They are particularly useful for tasks such as data validation, parsing, and data extraction.

The syntax and features of regular expressions may vary slightly between different programming languages and implementations. However, the core concepts remain the same.

With regular expressions, you can match literal characters, define character classes to match sets of characters, specify repetition with quantifiers, and use metacharacters for more complex pattern matching. You can also use anchors to match positions within a string, escape sequences to match special characters, and grouping and capturing to extract specific parts of the matched text.

Regular expressions support advanced features such as alternation to match multiple alternatives, backreferences to refer to previously matched content, and lookarounds for lookahead and lookbehind assertions.

It is important to note that regular expressions can be quite powerful but also complex. Constructing and debugging complex regular expressions may require practice and careful consideration. It's recommended to break down complex patterns into smaller components and test them incrementally.

In this document, we will explore the key concepts and syntax of regular expressions, providing examples and practical tips to help you understand and effectively use regular expressions in your projects.

Now let's dive into the world of regular expressions and discover their immense potential for pattern matching and text manipulation.

2 Literal Characters

In regular expressions, literal characters are the simplest form of pattern matching. They represent exact characters that you want to match in the target string.

For example, the regular expression ‘cat‘ will match the exact sequence of characters ”cat” in a string. The letters ’c’, ’a’, and ’t’ are treated as literal characters and must appear in that exact order for a match to occur.

It’s important to note that literal characters are case-sensitive by default. So, the regular expression ‘cat‘ will not match the string ”Cat” or ”CAT”, but it will match ”cat” exactly.

If you need case-insensitive matching, you can specify a modifier flag. In many programming languages, the ‘i‘ flag is commonly used to enable case-insensitive matching. For example, the regular expression ‘/cat/i‘ would match ”cat”, ”Cat”, ”CAT”, or any other combination of upper and lowercase letters.

Literal characters can include alphabetic letters, digits, and various special characters. However, some characters have special meanings in regular expressions, such as metacharacters. To match these special characters as literal characters, you need to escape them using a backslash (\).

For instance, to match a literal dot (.), you would use the regular expression \. The backslash escapes the special meaning of the dot and treats it as a literal character.

In summary, literal characters in regular expressions match the exact sequence of characters you specify. They provide a simple and straightforward way to search for specific strings or patterns in your target text.

3 Metacharacters

Metacharacters in regular expressions are special characters that have predefined meanings and provide powerful pattern matching capabilities. They allow you to express more complex search patterns beyond literal characters.

Metacharacters can be combined and used in various combinations to create more complex patterns. It's important to note that some metacharacters may have different meanings or require escaping depending on the programming language or regex engine you are using.

Understanding and properly utilizing metacharacters can greatly enhance the power and flexibility of your regular expressions, enabling you to match and manipulate patterns in your target text more effectively.

3.1 Single Characters

Here are the commonly used single character metacharacters:

- **. (Dot)**: Matches any character except a newline. It can be used to represent any character in a pattern.
- **\w** Match any single alphanumeric character: 0-9, a-z, A-Z, and _ (underscore).
- **\d** To match any single digit.
- **\s** Match any single whitespace character.

Note:-

The uppercase version performs the opposite action compared to its lowercase counterpart

- **\t** Matches a tab character
- **\n** Matches a newline character
- **\r** Matches a carriage return character

These are just a few examples of the most common escape sequences in regex. There are additional escape sequences available to represent other special characters or character classes. Using escape sequences, you can match specific characters or character patterns in your regular expressions.

3.2 Quantifiers

Here are the commonly used quantifiers:

- *** (Asterisk)**: Matches zero or more occurrences of the preceding element. For example, the pattern ‘ab*‘ would match ”a”, ”ab”, ”abb”, ”abbb”, and so on.
- **+ (Plus)**: Matches one or more occurrences of the preceding element. The pattern ‘ab+‘ would match ”ab”, ”abb”, ”abbb”, and so on, but not ”a”.
- **? (Question Mark)**: Matches zero or one occurrence of the preceding element. For example, the pattern ‘ab?‘ would match ”a” or ”ab”.
- **| (Pipe)**: Acts as an OR operator and matches either the pattern on the left or the pattern on the right. For instance, the pattern ‘a|b‘ would match either ”a” or ”b”.
- **() (Parentheses)**: Creates a group that allows you to apply quantifiers and modifiers to multiple characters. It also enables capturing of matched content for later use.
- **[] (Square Brackets)**: Defines a character class, allowing you to specify a set of characters to match. For example, the pattern ‘[abc]‘ would match either ”a”, ”b”, or ”c”.
- **\(Backslash)**: Escapes special characters, allowing you to match them as literal characters. For instance, to match a literal asterisk, you would use “.
- **{ n }**: (Matches exactly n times)
- **{ n ,}**: (Matches at least n times)
- **{ n , m}**: (Matches from n to m times)

3.3 Anchors (Position)

Here are the commonly used position metacharacters:

- **^:** Beginning (Goes in the beginning of the expression)
- **$:** End (Goes at the end of the expression)
- **\b:** Word boundry
- **\B:** Non-Word boundry

Word Boundry Example:

Let's say we want to find all occurrences of the word "apple" in a given text, but we only want to match it as a whole word and not as part of another word. We can use the word boundary \b to achieve this.

\bapple\b

Non-Word Boundry Example:

Let's say we want to find all occurrences of the word "apple" in a given text, but we only want to match it as a non whole word and find where it appears in other words, for example the word **pineapple**. We can use the non-word boundary \b to achieve this.

\Bapple\B

4 Character Classes

Character classes in regular expressions allow you to specify a set of characters to match at a particular position in the target string. They provide a convenient way to define patterns for matching specific types of characters.

To define a character class, you enclose the desired characters within square brackets []. The character class will match any single character that matches one of the characters within the brackets.

Here are some examples of character classes:

- **[abc]**: Matches either 'a', 'b', or 'c'. It will match any one of these characters at the specified position.
- **[0-9]**: Matches any digit from 0 to 9. It will match any single digit character.
- **[a-z]**: Matches any lowercase letter from 'a' to 'z'. It will match any single lowercase letter.
- **[A-Z]**: Matches any uppercase letter from 'A' to 'Z'. It will match any single uppercase letter.
- **[0-9a-f]**: Matches any hexadecimal digit. It will match any single digit from 0 to 9 or any lowercase letter from 'a' to 'f'.
- **[^0-9]**: The caret symbol (^) at the beginning of the character class negates the match. It matches any character that is not a digit. In this example, it will match any non-digit character.

Character classes can also include special character ranges, such as '\w' for word characters (letters, digits, and underscore) or '\s' for whitespace characters (spaces, tabs, newlines, etc.). These shorthand character classes provide a convenient way to match commonly used character sets.

You can use metacharacters within character classes as well, but they may have different meanings. For example, the hyphen (-) has a special meaning within a character class as a range indicator. To match a literal hyphen, you can either escape it with a backslash (\-) or place it as the first or last character within the brackets.

Character classes provide a flexible way to define patterns for matching specific types of characters in your target text. By combining character classes with other regular expression constructs, you can create powerful and precise matching patterns.

5 Escaping

In regular expressions (regex), escape sequences are used to represent special characters or sequences of characters that have a predefined meaning. They allow you to match characters that would otherwise be interpreted as special regex syntax. Escape sequences in regex are represented by a backslash (followed by a specific character.

Here are some common escape sequences in regex:

- \. The backslash followed by a period (\.) matches a literal period character. The period character has a special meaning in regex, as it matches any character except a newline. By using the escape sequence \. you can match a literal period.
- \\ The backslash followed by another backslash (\\) matches a literal backslash character. Since the backslash is used as the escape character in regex, you need to escape it with another backslash to match a literal backslash.

6 Grouping and Capturing

As seen briefly in the quantifiers section, Grouping and capturing are features in regular expressions that allow you to group parts of a pattern together and capture the matched content for later use.

Grouping:

- Grouping is denoted by placing the desired pattern inside parentheses ( ). It allows you to create subexpressions within a larger regex pattern.
- By grouping parts of a pattern, you can apply quantifiers or modifiers to the entire group, treating it as a single unit.
- Grouping is useful for establishing precedence, defining alternations, or applying repetition to specific sections of a pattern.

For example, the regex (ab)+ matches one or more occurrences of the sequence "ab". The parentheses group the characters "ab" together, and the + quantifier applies to the entire group.

Capturing

- Capturing is the process of extracting the matched content of a specific group within a regex pattern.
- When a pattern contains capturing groups, the substrings that match those groups are saved in memory for later use.
- Each capturing group is assigned a number, starting from 1, based on the order of the opening parentheses from left to right.

For example, consider the regex pattern `(ab)+(\d+)`. It contains two capturing groups: `(ab)` and `(\d+)`. If the input string is "abab123", the first capturing group captures "ab" and the second capturing group captures "123".

Captured content can be referenced in a variety of ways, such as:

- By using backreferences like `\1`, `\2`, etc., to refer to the captured content later within the same regex pattern.
- By using programming language-specific methods or functions to access the captured content programmatically.

Note:-

Grouping and capturing are powerful features in regular expressions that allow you to create more complex patterns and extract specific parts of matched content for further processing or analysis.

7 Alternation

In regular expressions (regex), alternation allows you to specify multiple alternatives for matching. It allows you to create a pattern that matches one of several choices. Alternation is denoted by the pipe character (|), which acts as a logical OR operator within the regex pattern.

Here's how alternation works in regex:

Basic Alternation:

- The basic syntax for alternation is choice1|choice2, where choice1 and choice2 represent the alternative patterns you want to match.
- When the regex engine encounters the alternation, it tries to match the pattern on the left side of the | first. If it doesn't match, it tries to match the pattern on the right side.
- The alternation continues until a match is found or all alternatives have been exhausted.

For example, the regex **apple|banana** matches either the word **"apple"** or the word **"banana".** If the input string is "I like bananas", it would match the word "banana".

Grouping Alternatives:

- You can use parentheses to group alternatives together and create more complex alternation patterns.
- For example, the regex (apple|banana)(pie)? matches either "apple" or "banana" followed by an optional " pie" string. It would match "apple", "apple pie", "banana", or "banana pie".

Additional Alternation:

- Alternation can be combined with other regex constructs, such as character classes, quantifiers, or capturing groups.
- For example, the regex (apple|banana)[s|es] matches either "apple" or "banana" followed by an "s" or "es". It would match "apples", "bananas", "apple", or "banana".

Note:-

Alternation is a useful feature in regex when you want to match different options or choices within a pattern. It allows you to specify multiple alternatives and flexibly match different variations of input.

8 Backreferences

n regular expressions (regex), backreferences are a feature that allows you to refer to previously matched content within the same pattern. They are used to match repeated or duplicated patterns. Backreferences are denoted by the backslash followed by a number (\1, \2, etc.) corresponding to the capturing group you want to reference.

Here's how backreferences work in regex:

Capturing Groups:

- First, you need to create capturing groups by enclosing a pattern in parentheses.
- When a regex pattern contains capturing groups, the substrings that match those groups are saved in memory for later use.
- Each capturing group is assigned a number, starting from 1, based on the order of the opening parentheses from left to right.

Backreferencing:

- Once you have capturing groups, you can use backreferences to refer to the matched content of those groups later in the pattern.
- The backreference \1 refers to the content matched by the first capturing group, \2 refers to the content matched by the second capturing group, and so on.

For example, consider the regex pattern (ab)+\1. It contains a capturing group (ab) followed by the backreference \1. This pattern matches repetitions of the sequence "ab". The backreference \1 ensures that the repeated sequence matches the exact content captured by the first group.

Note:-

Backreferences are specific to the current pattern and **cannot** reference content captured in previous or subsequent patterns.

Backreferencing Example (neovim)

To swap the order of *firstname, lastname*, we can do something like:

:%s/\(\w\+\), \(\w\+\)/\2, \1/g

Which will turn:

nate, warner

Into:

warner, nate

9 Lookarounds (Generic)

Lookarounds are a powerful feature in regular expressions (regex) that allow you to define conditions around a pattern without including the condition itself in the match. Lookarounds assert whether a certain pattern is present or absent at a particular position in the input string, without actually including that pattern in the match. There are two types of lookarounds: positive lookaheads and negative lookaheads, as well as positive lookbehinds and negative lookbehinds. Let's explore each of them:

Positive Lookahead (?=):

- Positive lookaheads are denoted by (?=...).
- They assert that the pattern inside the lookahead must be present immediately ahead of the current position in the input string, but it is not included in the match.
- For example, the regex pattern foo(?=bar) matches the substring "foo" only if it is followed by the substring "bar".

Negative Lookahead (?!):

- Negative lookaheads are denoted by (?!...).
- They assert that the pattern inside the lookahead must not be present immediately ahead of the current position in the input string.
- For example, the regex pattern foo(?!bar) matches the substring "foo" only if it is not followed by the substring "bar".

Positive Lookbehind (?<=):

- Positive lookbehinds are denoted by (?<=...).
- They assert that the pattern inside the lookbehind must be present immediately before the current position in the input string, but it is not included in the match.
- Positive lookbehinds are only supported in some regex engines, and the length of the lookbehind must be fixed (no variable length).
- For example, the regex pattern (?<=foo)bar matches the substring "bar" only if it is preceded by the substring "foo".

Negative Lookbehind (?<!):

- Negative lookbehinds are denoted by (?<!...).
- They assert that the pattern inside the lookbehind must not be present immediately before the current position in the input string.
- Negative lookbehinds are only supported in some regex engines, and the length of the lookbehind must be fixed (no variable length).
- For example, the regex pattern (?<!foo)bar matches the substring "bar" only if it is not preceded by the substring "foo".

Note:-

Above is the syntax that is used by the python module *re*
It's important to note that the availability and syntax of lookarounds may vary between different regex engines.

10 Neovim lookarounds

The syntax for lookarounds in the neovim regex engine is a bit different so let's take a look at them as well.

In Neovim regex, lookarounds are denoted by specific symbols (\ze, \@!, \@<=, \@<!) placed at specific positions within the regex pattern. They allow you to define conditions around a pattern without including the condition itself in the match.

Positive Lookahead:

- Let's say we want to match the word "apple" only if it is followed by the word "pie".
- **Syntax**: pattern\ze lookahead-pattern
- \ze is placed after the initial pattern and indicates the position where the match should end.
- lookahead-pattern is the pattern that should be present immediately ahead of the current position for a match to occur.
- The content matched by pattern is not included in the match result.
- **Example:** /foo\ze bar will match "foo" but only if it procedes the word "bar"

Negative Lookahead:

- Suppose we want to match the word "car" only if it is not followed by the word "pet"
- **Syntax**: pattern\(lookahead-pattern\)\@!
- \@! is placed after the initial pattern and indicates that the lookahead-pattern should not be present immediately ahead of the current position.
- lookahead-pattern is the pattern that should not be present for a match to occur.
- The content matched by pattern is included in the match result.
- **Example:** /foo\(bar\)\@! will match "foo" but only if it is **does not** procede the word "bar"

Positive Lookbehind:

- Let's say we want to match the word "foo" only if it is preceded by the word "bar".
- **Syntax**: \(lookbehind-pattern\)lookbehind\@<=pattern
- <= is placed after the lookbehind pattern and indicates that the lookbehind-pattern should be present immediately before the current position for a match to occur.
- lookbehind-pattern is the pattern that should be present before the current position.
- The content matched by pattern is not included in the match result.
- **Example:** /\(bar\)\@<=foo will match "foo" but only if it is proceded by the word "bar"

Negative Lookbehind:

- **Syntax**: \(\@<!lookbehind-pattern\)pattern
- \@<! is placed before the initial pattern and indicates that the lookbehind-pattern should not be present immediately before the current position.
- lookbehind-pattern is the pattern that should not be present for a match to occur.
- The content matched by pattern is included in the match result.
- **Example:** /\(bar\)\@<!foo will match foo but only if it **is not** proceded by the word "bar"

11 Greedy and Lazy Matching

Greedy and lazy matching refer to the behavior of quantifiers in regular expressions.

By default, quantifiers in regular expressions are greedy, meaning they match as much as possible while still allowing the overall pattern to match. This behavior is often intuitive because it tries to find the longest possible match.

For example, consider the regular expression pattern a.*b applied to the input string abcabcab. The .* quantifier means "match any character (except newline) zero or more times." In a greedy match, it will match as much as possible, resulting in the longest match: abcabcab.

However, there are cases where you might want to perform a more conservative or minimal match, where the quantifier matches as little as possible. This is known as lazy or non-greedy matching.

To perform a lazy match, you can use the ? modifier after a greedy quantifier. It instructs the quantifier to match as little as possible while still allowing the overall pattern to match.

For example, if we modify the previous pattern to a.*?b and apply it to the same input string, the .*? quantifier will match as few characters as possible before encountering the subsequent b. It will result in the shortest match: ab.

Lazy matching is useful in scenarios where you want to capture the smallest possible substring that satisfies the pattern.

Note:-

It's important to note that lazy matching applies to quantifiers such as *, +, ?, and . Without the ? modifier, these quantifiers are greedy by default.
Furthermore, the neovim regex engine **does not** support lazy matching

12 Flags

In regular expressions (regex),flags are used to modify the behavior of the pattern matching. They are typically specified as options or additional characters added to the regex pattern itself. Here's an explanation of flags:

Commonly Used Flags

- g: Global matching. Finds all matches in the input string, rather than stopping after the first match.
- i: Case-insensitive matching. The pattern matches regardless of letter case.
- m: Multiline mode. Allows the ^ and $ anchors to match the start and end of each line within a multiline input string.
- s: Single-line mode. The dot (.) matches any character, including newline characters.
- x: Verbose mode. Allows the use of whitespace and comments within the pattern for improved readability.

Note:-

Neovim has an additional flag */c*, which stands for *confirm* and has you enter Y/N for each search and replace match

13 Bonus: Neovim Multiline Support

Consider the scenario:

```
1 foomane
2 barmane
```

And I want to swap the order. That is, "barmane" on line 1, and "foomane" on line 2. Our pattern would look something like:

```
:%s/\(foomane\)\(\_.*barmane\)/barmane\rfoomane/g
```

Which would produce:

```
1 barmane
2 foomane
```

So you can infer that the syntax to start matching on the next line is:

```
\_.*
```

And the syntax for start replacing on the next line is:

```
\r
```

Note:-

Capturing is not necessary here, but it is good convention to capture your matches in the case that you require Backreferencing.