

# Regular Expressions Tutorial Table of Contents

---

This regular expressions tutorial teaches you every aspect of regular expressions. Each topic assumes you have read and understood all previous topics. If you are new to regular expressions, you should read the topics in the order presented.

## [Introduction](#)

The introduction indicates the scope of the tutorial and which regex flavors are discussed. It also introduces basic terminology.

## [Literal Characters and Special Characters](#)

The simplest regex consists of only literal characters. Certain characters have special meanings in a regex and have to be escaped. Escaping rules may get a bit complicated when using regexes in software source code.

## [Non-Printable Characters](#)

Non-printable characters such as control characters and special spacing or line break characters are easier to enter using control character escapes or hexadecimal escapes.

## [How a Regex Engine Works Internally](#)

First look at the internals of the regular expression engine's internals. Later topics build on this information. Knowing the engine's internals greatly helps you to craft regexes that match what you intended, and not match what you do not want.

## [Character Classes or Character Sets](#)

A character class or character set matches a single character out of several possible characters, consisting of individual characters and/or ranges of characters. A negated character class matches a single character not in the character class.

## [Shorthand Character Classes](#)

Shorthand character classes allow you to use common sets of characters quickly. You can use shorthands on their own or as part of character classes.

## [Character Class Subtraction](#)

Character class subtraction allows you to match one character that is present in one set of characters but not present in another set of characters.

## [Character Class Intersection](#)

Character class intersection allows you to match one character that is present in one set of characters and also present in another set of characters.

## [The Dot](#)

The dot matches any character, though usually not line break characters unless you change an option.

## [Anchors](#)

Anchors are zero-length. They do not match any characters, but rather a position. There are anchors to match at the start and end of the subject string, and anchors to match at the start and end of each line.

## [Word Boundaries](#)

Word boundaries are like anchors, but match at the start of a word and/or the end of a word.

## [Alternation](#)

By separating different sub-regexes with vertical bars, you can tell the regex engine to attempt them from left to right, and return success as soon as one of them can be matched.

## [Optional Items](#)

Putting a question mark after an item tells the regex engine to match the item if possible, but continue anyway (rather than admit defeat) if it cannot be matched.

## [Repetition Using Various Quantifiers](#)

Three styles of operators, the star, the plus, and curly braces, allow you to repeat an item zero or more times, once or more, or an arbitrary number of times. It is important to understand that these quantifiers are “greedy” by default, unless you explicitly make them “lazy”.

## [Grouping](#)

By placing parentheses around part of the regex, you tell the engine to treat that part as a single item when applying quantifiers or to group alternatives together. Parentheses also create capturing groups allow you to reuse the text matched by part of the regex.

## [Backreferences](#)

Backreferences to capturing groups match the same text that was previously matched by that capturing group, allowing you to match patterns of repeated text.

## [Named Groups and Backreferences](#)

Regular expressions that have multiple groups are much easier to read and maintain if you use named capturing groups and named backreferences.

## [Branch Reset Groups](#)

When using alternation to match different variants of the same thing, you can put the alternatives in a branch reset group. Then all the alternatives share the same capturing groups. This allows you to use backreferences or retrieve part of the matched text without having to check which of the alternatives captured it.

## [Free-Spacing and Comments](#)

Splitting a regular expression into multiple lines, adding comments and whitespace, makes it easier to read and understand.

## [Unicode Characters and Properties](#)

If your regular expression flavor supports Unicode, then you can use special Unicode regex tokens to match specific Unicode characters, or to match any character that has a certain Unicode property or is part of a particular Unicode script or block.

## [Mode Modifiers](#)

Change matching modes such as “case insensitive” for specific parts of the regular expression.

## [Atomic Grouping and Possessive Quantifiers](#)

Nested quantifiers can cause an exponentially increasing amount of backtracking that brings the regex engine to a grinding halt. Atomic grouping and possessive quantifiers provide a solution.

## [Lookaround with Zero-Length Assertions, \[part 1\]\(#\) and \[part 2\]\(#\)](#)

With lookahead and lookbehind, collectively called lookahead, you can find matches that are followed or not followed by certain text, and preceded or not preceded by certain text, without having the preceding or following

text included in the overall regex match. You can also use lookahead to test the same part of the match for multiple requirements.

### [Keep The Text Matched So Far out of The Overall Regex Match](#)

Keeping the text matched so far out of the overall regex match allows you to find matches that are preceded by certain text, without having that preceding text included in the overall regex match. This method is primarily of interest with regex flavors that have no or limited support for lookbehind.

### [Conditionals](#)

A conditional is a special construct that first evaluates a lookahead or backreference, and then execute one sub-regex if the lookahead succeeds, and another sub-regex if the lookahead fails.

### [Recursion](#)

Recursion matches the whole regex again at a particular point inside the regex, which makes it possible to match balanced constructs.

### [Subroutine Calls](#)

Subroutine calls allow you to write regular expressions that match the same constructs in multiple places without having to duplicate parts of your regular expression.

### [Recursion, Subroutines, & Capturing](#)

Capturing groups inside recursion and subroutine calls are handled differently by the regex flavors that support them.

### [Backreferences with Recursion Level](#)

Special backreferences match the text stored by a capturing group at a particular recursion level, instead of the text most recently matched by that capturing group.

### [Recursion, Subroutines, & Backtracking](#)

The regex flavors that support recursion and subroutine calls backtrack differently after a recursion or subroutine call fails.

### [POSIX Bracket Expressions](#)

If you are using a [POSIX-compliant](#) regular expression engine, you can use POSIX bracket expressions to match locale-dependent characters.

### [Issues with Zero-Length Matches](#)

When a regex can find zero-length matches, regex engines use different strategies to avoid getting stuck on a zero-length match when you want to iterate over all matches in a string. This may lead to different match results.

### [Continuing from The Previous Match Attempt](#)

Forcing a regex match to start at the end of a previous match provides an efficient way to parse text data.

## Make a Donation

---

Did this website just save you a trip to the bookstore? Please [make a donation](#) to support this site, and you'll get a **lifetime of advertisement-free access** to this site!

Site last updated: 08 April 2020

Copyright © 2003-2020 Jan Goyvaerts. All rights reserved.