



Quick Start Tutorial Tools & Languages Examples Reference Book Reviews



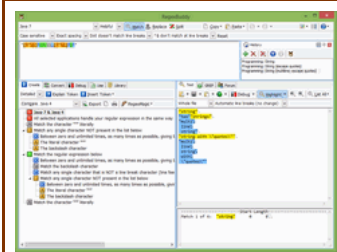
Make a Donation

Regex Tutorial

- [Introduction](#)
- [Table of Contents](#)
- [Special Characters](#)
- [Non-Printable Characters](#)
- [Regex Engine Internals](#)
- [Character Classes](#)
- [Character Class Subtraction](#)
- [Character Class Intersection](#)
- [Shorthand Character Classes](#)
- [Dot](#)
- [Anchors](#)
- [Word Boundaries](#)
- [Alternation](#)
- [Optional Items](#)
- [Repetition](#)
- [Grouping & Capturing](#)
- [Backreferences](#)
- [Backreferences, part 2](#)
- [Named Groups](#)
- [Branch Reset Groups](#)
- [Free-Spacing & Comments](#)
- [Unicode](#)
- [Mode Modifiers](#)
- [Atomic Grouping](#)
- [Possessive Quantifiers](#)
- [Lookahead & Lookbehind](#)
- [Lookaround, part 2](#)
- [Keep Text out of The Match](#)
- [Conditionals](#)
- [Balancing Groups](#)
- [Recursion](#)
- [Subroutines](#)
- [Recursion & Capturing](#)
- [Recursion & Backreferences](#)
- [Recursion & Backtracking](#)
- [POSIX Bracket Expressions](#)
- [Zero-Length Matches](#)
- [Continuing Matches](#)

More on This Site

- [Introduction](#)
- [Regular Expressions Quick Start](#)
- [Regular Expressions Tutorial](#)
- [Replacement Strings Tutorial](#)
- [Applications and Languages](#)



Easily create and understand regular expressions today.
Compose and analyze regex patterns with RegexBuddy's easy-to-use interface. Gain a new grasp of regex blocks and intuitive regex tree, instead of struggling with the traditional regex syntax. Developed by the author of this website, RegexBuddy makes learning and using regular expressions easier than ever. [Get your own copy of RegexBuddy](#)

Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called "lookaround", are zero-length assertions just like the [start of line](#), and [start and end of word](#) anchors explained earlier in this tutorial. The difference is that lookaround matches characters, but then gives up the match, returning only the result: match or no match. That is, they are called "assertions". They do not consume characters in the string, but only assert whether a match is or is not. Lookaround allows you to create regular expressions that are impossible to create without them. They would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. Explaining [character classes](#), this tutorial explained why you cannot use a negated character class to match a character followed by a `u`. Negative lookahead provides the solution: `q(?:!u)`. The negative lookahead construct is a pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation point. Inside the parentheses, we have the trivial regex `u`.

Positive lookahead works just the same. `q(?:=u)` matches a `q` that is followed by a `u`, without making the `u` part of the match. The positive lookahead construct is a pair of parentheses, with the opening parenthesis followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any regular expression can be used inside the lookahead. If it contains [capturing groups](#) then those groups work as normal and backreferences to them will work normally, even outside the lookahead. (The only exception is that which treats all groups inside lookahead as non-capturing.) The lookahead itself is not a capturing group and is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside the lookahead, you have to put capturing parentheses around the regex inside the lookahead, like this: `(?= (...))`. The other way around will not work, because the lookahead will already have discarded the regex match before the capturing group is to store its match.



Regex Engine Internals

First, let's see how the engine applies `q(?:!u)` to the string `Iraq`. The first token in the regex is the [literal](#) `q`, which the engine already knows. This causes the engine to traverse the string until the `q` in the string is matched. The position of the string is now the void after the string. The next token is the lookahead. The engine takes note that it is a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is `!`. It does not match the void after the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. The entire regex has matched, and `q` is returned as the match.

Let's try applying the same regex to `quit`. `q` matches `q`. The next token is the `!` inside the lookahead. The character is the `u`. These match. The engine advances to the next character: `i`. However, it is done with the lookahead. The engine notes success, and discards the regex match. This causes the engine to move back in the string to `u`.

[Regular Expressions Examples](#)
[Regular Expressions Reference](#)
[Replacement Strings Reference](#)
[Book Reviews](#)
[Printable PDF](#)
[About This Site](#)
[RSS Feed & Blog](#)



Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since the other permutations of this regex, the engine has to start again at the beginning. Since `q` cannot match `i` else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let `q(?:=u)i` to `quit`. The lookahead is now positive and is followed by another token. Again, `q` matches `q`, `i` matches `i`. Again, the match from the lookahead must be discarded, so the engine steps back from `i` in to `u`. The lookahead was successful, so the engine continues with `i`. But `i` cannot match `u`. So this match fails. All remaining attempts fail as well, because there are no more `q`'s in the string.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step back the string, to check if the text inside the lookbehind can be matched there. `(?<!a)b` matches a "b" that is preceded by an "a", using negative lookbehind. It doesn't match `cab`, but matches the `b` (and only the `b`) in `debt`. `(?<=a)b` (positive lookbehind) matches the `b` (and only the `b`) in `cab`, but does not match `bed` or `debt`.

The construct for positive lookbehind is `(?<=text)`: a pair of parentheses, with the opening parenthesis by a question mark, "less than" symbol, and an equals sign. Negative lookbehind is written as `(?<!\text)`, exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply `(?<=a)b` to `thingamabob`. The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if `a` can be matched there. The engine cannot step back one character because there are no characters before the `t`. So the lookbehind fails. The engine starts again at the next character, the `h`. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the `m` in the string. The engine again steps back one character, and notices that the `a` can be matched there. The positive lookbehind matches. Because the lookbehind is zero-length, the current position in the string remains at the `m`. The next token is `b`, which cannot match here. The engine steps back, and finds out that the `m` does not match `a`.

The next character is the first `b` in the string. The engine steps back and finds out that `a` satisfies the lookbehind. The engine matches `b`, and the entire regex has been matched successfully. It matches one character: the first `b` in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to match a word not ending with an "s", you could use `\b\w+(?<!s)\b`. This is definitely not the same as `\b\w+[^s]` applied to `John's`, the former matches `John` and the latter matches `John'` (including the apostrophe). I leave it up to you to figure out why. (Hint: `\b` matches between the apostrophe and the `s`). The latter also does single-letter words like "a" or "I". The correct regex without using lookbehind is `\b\w*[^s\W]\b` (star instead of plus, and `\W` in the character class). Personally, I find the lookbehind easier to understand. The last regex works correctly, has a double negation (the `\W` in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though. (Except perhaps for Tcl, which treats negated short negated character classes as an error.)

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. The regular expression engine needs to be able to figure out how many characters to step back before checking the lookbehind. When evaluating the lookbehind, the engine determines the length of the regex inside the lookbehind, steps back that many characters in the subject string, and then applies the regex inside the lookbehind from left to right just as it would with a normal regex.

Many regex flavors, including those used by [Perl](#) and [Python](#), only allow fixed-length strings. You can use [text](#), [character escapes](#), [Unicode escapes](#) other than `\X`, and [character classes](#). You cannot use [qualifiers](#) or [backreferences](#). You can use [alternation](#), but only if all alternatives have the same length. These flavors evaluate the lookbehind by first stepping back through the subject string for as many characters as the lookbehind needs, then attempting the regex inside the lookbehind from left to right.

[PCRE](#) is not fully Perl-compatible when it comes to lookbehind. While Perl requires alternatives inside lookbehind to have the same length, PCRE allows alternatives of variable length. [PHP](#), [Delphi](#), [R](#), and [Ruby](#) also allow lookbehind, but each alternative still has to be fixed-length. Each alternative is treated as a separate fixed-length lookbehind.

[Java](#) takes things a step further by allowing finite repetition. You still cannot use the [star](#) or [plus](#), but you can use the [question mark](#) and the [curly braces](#) with the `max` parameter specified. Java determines the minimum and maximum possible lengths of the lookbehind. The lookbehind in the regex `(?<!ab{2,4}c{3,5}d)te` has possible lengths. It can be between 7 to 11 characters long. When Java (version 6 or later) tries to evaluate the lookbehind, it first steps back the minimum number of characters (7 in this example) in the string and evaluates the regex inside the lookbehind as usual, from left to right. If it fails, Java steps back one more character and tries again. If the lookbehind continues to fail, Java continues to step back until the lookbehind either succeeds or it has stepped back the maximum number of characters (11 in this example). This repeated stepping back through the subject string kills performance when the number of possible lengths of the lookbehind grows. Keep this in mind. Don't choose an arbitrarily large maximum number of repetitions to work around the lack

PowerGREP 4



[PowerGREP](#) is probably the most powerful regex-based text processing tool available today. A knowledge worker's Swiss army knife for searching through, extracting information from, and updating piles of files.

Use regular expressions to search through large numbers of text and binary files. Quickly find the files you are looking for, or extract the information you need. Look through just a handful of files or folders, or scan entire drives and network shares.

Search and replace using text, binary

data or one or more regular expressions to automate repetitive editing tasks. Preview replacements before modifying files, and stay safe with flexible backup and undo options.

Use regular expressions to rename files, copy files, or merge and split the contents of files. Work with plain text files, Unicode files, binary files, compressed files, and files in proprietary formats such as MS Office, OpenOffice, and PDF. Runs on Windows 2000, XP, Vista, 7, 8, and 8.1.

[More information](#)

[Download PowerGREP now](#)

quantifiers inside lookbehind. Java 4 and 5 have bugs that cause lookbehind with alternation or variable c to fail when it should succeed in some situations. These bugs were fixed in Java 6.

The only regex engines that allow you to use a full regular expression inside lookbehind, including infinite and backreferences, are the [JGsoft engine](#) and the [.NET framework RegEx classes](#). These regex engines apply the regex inside the lookbehind backwards, going through the regex inside the lookbehind and the subject string from right to left. They only need to evaluate the lookbehind once, regardless of how many possible lengths it has.

Finally, flavors like [JavaScript](#) and [Tcl](#) do not support lookbehind at all, even though they do support looka

Lookaround Is Atomic

The fact that lookahead is zero-length automatically makes it [atomic](#). As soon as the lookahead is satisfied, the regex engine forgets about everything inside the lookahead. It will not backtrack in the lookahead to try different permutations.

The only situation in which this makes any difference is when you use [capturing groups](#) inside the look. Since the regex engine does not backtrack into the lookahead, it will not try different permutations of the groups.

For this reason, the regex `(?=(\d+))\w+\1` never matches `123x12`. First the lookahead captures `12`. `\w+` then matches the whole string and backtracks until it matches only `1`. Finally, `\w+` fails since `\1` cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex backtracking steps created by `\d+` have been discarded. It never gets to the point where the lookahead only `12`.

Obviously, the regex engine does try further positions in the string. If we change the subject string, `(?=(\d+))\w+\1` does match `56x56` in `456x56`.

If you don't use capturing groups inside lookahead, then all this doesn't matter. Either the lookahead can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

Make a Donation

Did this website just save you a trip to the bookstore? Please [make a donation](#) to support this site, and your **lifetime of advertisement-free access** to this site! Credit cards, PayPal, and Bitcoin gladly accepted.

Page URL: <http://www.regular-expressions.info/lookaround.html>

Page last updated: 13 August 2014

Site last updated: 16 March 2015

Copyright © 2003-2015 Jan Goyvaerts. All rights reserved.