Keep The Text Matched So Far out of The Overall Regex Match

Lookbehind is often used to match certain text that is preceded by other text, without including the other text in the overall regex match. `(?<=h)d` matches only the second `d` in `adhd`. While a lot of regex flavors support lookbehind, most regex flavors only allow a subset of the regex syntax to be used inside lookbehind. Perl and Boost require the lookbehind to be of fixed length. PCRE and Ruby allow alternatives of different length, but still don't allow quantifiers other than the fixed-length `{n}`.

To overcome the limitations of lookbehind, Perl 5.10, PCRE 7.2, Ruby 2.0, and Boost 1.42 introduce a new feature that can be used instead of lookbehind for its most common purpose. `\K` keeps the text matched so far out of the overall regex match. `h\Kd` matches only the second `d` in `adhd`.

The JGsoft flavor has always supported unrestricted lookbehind, which is much more flexible than `\K`. Still, JGsoft V2 adds support for `\K` if you prefer this way of working.

Looking Inside The Regex Engine

Let's see how `h\Kd` works. The engine begins the match attempt at the start of the string. `h` fails to match `a`. There are no further alternatives to try. The match attempt at the start of the string has failed.

The engine advances one character through the string and attempts the match again. `h` fails to match `d`.

Advancing again, `h` matches `h`. The engine advances through the regex. The regex has now reached `\K` in the regex and the position between `h` and the second `d` in the string. `\K` does nothing other than to tell that if this match attempt ends up succeeding, the regex engine should pretend that the match attempt started at the present position between `h` and `d`, rather than between the first `d` and `h` where it really started.

The engine advances through the regex. `d` matches the second `d` in the string. An overall match is found. Because of the position saved by `\K`, the second `d` in the string is returned as the overall match.

`\K` only affects the position returned after a successful match. It does not move the start of the match attempt during the matching process. The regex `hhh\Kd` matches the `d` in `hhhhd`. This regex first matches `hhh` at the start of the string. Then `\K` notes the position between `hhh` and `hd` in the string. Then `d` fails to match the fourth `h` in the string. The match attempt at the start of the string has failed.

Now the engine must advance one character in the string before starting the next match attempt. It advances from the actual start of the match attempt, which was at the start of the string. The position stored by `\K` does not change this. So the second match attempt begins at the position after the first `h` in the string. Starting there, `hhh` matches `hhh`, `\K` notes the position, and `d` matches `d`. Now, the position remembered by `\K` is taken into account, and `d` is returned as the overall match.

\K Can Be Used Anywhere

You can use `\K` pretty much anywhere in any regular expression. You should only avoid using it inside lookbehind. You can use it inside groups, even when they have quantifiers. You can have as many instances of `\K` in your regex as you like. `(ab\Kc|d\Ke)f` matches `cf` when preceded by `ab`. It also matches `ef` when preceded by `d`.

`\K` does not affect capturing groups. When `(ab\Kc|d\Ke)f` matches `cf`, the capturing group captures `abc` as if the `\K` weren't there. When the regex matches `ef`, the capturing group stores `de`.

Limitations of \K

Because \K does not affect the way the regex engine goes through the matching process, it offers a lot more flexibility than lookbehind in Perl, PCRE, and Ruby. You can put anything to the left of \K, but you're limited to what you can put inside lookbehind.

But this flexibility does come at a cost. Lookbehind really goes backwards through the string. This allows lookbehind check for a match before the start of the match attempt. When the match attempt was started at the end of the previous match, lookbehind can match text that was part of the previous match. \K cannot do this, precisely because it does not affect the way the regex engine goes through the matching process.

If you iterate over all matches of (?<=a)a in the string aaaa, you will get three matches: the second, third, and fourth a in the string. The first match attempt begins at the start of the string and fails because the lookbehind fails. The second match attempt begins between the first and second a, where the lookbehind succeeds and the second a is matched. The third match attempt begins after the second a that was just matched. Here the lookbehind succeeds too. It doesn't matter that the preceding a was part of the previous match. Thus the third match attempt matches the third a. Similarly, the fourth match attempt matches the fourth a. The fifth match attempt starts at the end of the string. The lookbehind still succeeds, but there are no characters left for a to match. The match attempt fails. The engine has reached the end of the string and the iteration stops. Five match attempts have found three matches.

Things are different when you iterate over a\Ka in the string aaaa. You will get only two matches: the second and the fourth a. The first match attempt begins at the start of the string. The first a in the regex matches the first a in the string. \K notes the position. The second a matches the second a in the string, which is returned as the first match. The second match attempt begins after the second a that was just matched. The first a in the regex matches the third a in the string. \K notes the position. The second a matches the fourth a in the string, which is returned as the first match. The third match attempt begins at the end of the string. a fails. The engine has reached the end of the string and the iteration stops. Three match attempts have found two matches.

Basically, you'll run into this issue when the part of the regex before the \K can match the same text as the part of the regex after the \K. If those parts can't match the same text, then a regex using \K will find the same matches than the same regex rewritten using lookbehind. In that case, you should use \K instead of lookbehind as that will give you better performance in Perl, PCRE, and Ruby.

Another limitation is that while lookbehind comes in positive and negative variants, \K does not provide a way to negate anything. (?<!a)b matches the string b entirely, because it is a "b" not preceded by an "a". [^a]\Kb does not match the string b at all. When attempting the match, [^a] matches b. The regex has now reached the end of the string. \K notes this position. But now there is nothing left for b to match. The match attempt fails. [^a]\Kb is the same as (?<=[^a])b, which are both different from (?<!a)b.

Make a Donation

Page URL: https://www.regular-expressions.info/keep.html
Page last updated: 22 November 2019
Site last updated: 08 April 2020