Zero-Length Regex Matches

We saw that anchors, word boundaries, and lookaround match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, word boundaries, or lookarounds, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable.

In email, for example, it is common to prepend a “greater than” symbol and a space to each line of the quoted message. In VB.NET, we can easily do this with `Dim Quoted As String = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)`. We are using multi-line mode, so the regex `^` matches at the start of the quoted message, and after each newline. The Regex.Replace method removes the regex match from the string, and inserts the replacement string (greater than symbol and a space). Since the match does not include any characters, nothing is deleted. However, the match does include a starting position. The replacement string is inserted there, just like we want it.

Using `^\d*$` to test if the user entered a number would give undesirable results. It causes the script to accept an empty string as a valid input. Let’s see why.

There is only one “character” position in an empty string: the void after the string. The first token in the regex is `^`. It matches the position before the void after the string, because it is preceded by the void before the string. The next token is `\d*`. One of the star’s effects is that it makes the `\d`, in this case, optional. The engine tries to match `\d` with the void after the string. That fails. But the star turns the failure of the `\d` into a zero-length success. The engine proceeds with the next regex token, without advancing the position in the string. So the engine arrives at `$`, and the void after the string. These match. At this point, the entire regex has matched the empty string, and the engine reports success.

The solution is to use the regex `^\d+$` with the proper quantifier to require at least one digit to be entered. If you always make sure that your regexes cannot find zero-length matches, other than special cases such as matching the start or end of each line, then you can save yourself the headache you’ll get from reading the remainder of this topic.

Skipping Zero-Length Matches

Not all flavors support zero-length matches. The TRegEx class in Delphi XE5 and prior always skips zero-length matches. The TPerlRegEx class does too by default in XE5 and prior, but allows you to change this via the State property. In Delphi XE6 and later, TRegEx never skips zero-length matches while TPerlRegEx does not skip them by default but still allows you to skip them via the State property. PCRE finds zero-length matches by default, but can skip them if you set PCRE_NOTEMPTY.

Advancing After a Zero-Length Regex Match

If a regex can find zero-length matches at any position in the string, then it will. The regex `\d*` matches zero or more digits. If the subject string does not contain any digits, then this regex finds a zero-length match at every position in the string. It finds 4 matches in the string `abc`, one before each of the three letters, and one at the end of the string.

Things get tricky when a regex can find zero-length matches at any position as well as certain non-zero-length matches. Say we have the regex `\d*|x`, the subject string `x1`, and a regex engine allows zero-length matches. Which and how many matches do we get when iterating over all matches? The answer depends on how the regex engine advances after zero-length matches. The answer is tricky either way.

The first match attempt begins at the start of the string. `\d` fails to match `x`. But the `*` makes `\d` optional. The first alternative finds a zero-length match at the start of the string. Until here, all regex engines that allow zero-length matches do the same.

Now the regex engine is in a tricky situation. We're asking it to go through the entire string to find all non-overlapping regex matches. The first match ended at the start of the string, where the first match attempt began. The regex engine needs a way to avoid getting stuck in an infinite loop that forever finds the same zero-length match at the start of the string.

The simplest solution, which is used by most regex engines, is to start the next match attempt one character after the end of the previous match, if the previous match was zero-length. In this case, the second match attempt begins at the position between the `x` and the `1` in the string. `\d` matches `1`. The end of the string is reached. The quantifier `*` is satisfied with a single repetition. `1` is returned as the overall match.

The other solution, which is used by Perl, is to always start the next match attempt at the end of the previous match, regardless of whether it was zero-length or not. If it was zero-length, the engine makes note of that, as it must not allow a zero-length match at the same position. Thus Perl begins the second match attempt also at the start of the string. The first alternative again finds a zero-length match. But this is not a valid match, so the engine backtracks through the regular expression. `\d*` is forced to give up its zero-length match. Now the second alternative in the regex is attempted. `x` matches `x` and the second match is found. The third match attempt begins at the position after the `x` in the string. The first alternative matches `1` and the third match is found.

But the regex engine isn't done yet. After `x` is matched, it makes one more match attempt starting at the end of the string. Here too `\d*` finds a zero-length match. So depending on how the engine advances after zero-length matches, it finds either three or four matches.

One exception is the JGsoft engine. The JGsoft engine advances one character after a zero-length match, like most engines do. But it has an extra rule to skip zero-length matches at the position where the previous match ended, so you can never have a zero-length match immediately adjacent to a non-zero-length match. In our example the JGsoft engine only finds two matches: the zero-length match at the start of the string, and `1`.

Python 3.6 and prior advance after zero-length matches. The `gsub()` function to search-and-replace skips zero-length matches at the position where the previous non-zero-length match ended, but the `finditer()` function returns those matches. So a search-and-replace in Python gives the same results as the Just Great Software applications, but listing all matches adds the zero-length match at the end of the string.

Python 3.7 changed all this. It handles zero-length matches like Perl. `gsub()` does now replace zero-length matches that are adjacent to another match. This means regular expressions that can find zero-length matches are not compatible between Python 3.7 and prior versions of Python.

PCRE 8.00 and later and PCRE2 handle zero-length matches like Perl by backtracking. They no longer advance one character after a zero-length match like PCRE 7.9 used to do.

The regexp functions in R and PHP are based on PCRE, so they avoid getting stuck on a zero-length match by backtracking like PCRE does. But the `gsub()` function to search-and-replace in R also skips zero-length matches at the position where the previous non-zero-length match ended, like `gsub()` in Python 3.6 and prior does. The other regexp functions in R and all the functions in PHP do allow zero-length matches immediately adjacent to non-zero-length matches, just like PCRE itself.

Caution for Programmers

A regular expression such as `$` all by itself can find a zero-length match at the end of the string. If you would query the engine for the character position, it would return the length of the string if string indexes are zero-based, or the

length+1 if string indexes are one-based in your programming language. If you would query the engine for the length of the match, it would return zero.

What you have to watch out for is that String[Regex.MatchPosition] may cause an access violation or segmentation fault, because MatchPosition can point to the void after the string. This can also happen with ^ and ^$ in multi-line mode if the last character in the string is a newline.

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!

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