Start of String and End of String Anchors

Thus far, we have learned about literal characters, character classes, and the dot. Putting one of these in a regex tells the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after, or between characters. They can be used to "anchor" the regex match at a certain position. The caret `^` matches the position before the first character in the string. Applying `^a` to `abc` matches `a`. `^b` does not match `abc` at all, because the `b` cannot be matched right after the start of the string, matched by `^`. See below for the inside view of the regex engine.

Similarly, `$` matches right after the last character in the string. `c$` matches `c` in `abc`, while `a$` does not match at all.

A regex that consists solely of an anchor can only find zero-length matches. This can be useful, but can also create complications that are explained near the end of this tutorial.

Useful Applications

When using regular expressions in a programming language to validate user input, using anchors is very important. If you use the code `if ($input =~ m/\d+/)` in a Perl script to see if the user entered an integer number, it will accept the input even if the user entered `qsdf4ghjk`, because `\d+` matches the `4`. The correct regex to use is `^\d+$`. Because "start of string" must be matched before the match of `\d+`, and "end of string" must be matched right after it, the entire string must consist of digits for `^\d+$` to be able to match.

It is easy for the user to accidentally type in a space. When Perl reads from a line from a text file, the line break is also be stored in the variable. So before validating input, it is good practice to trim leading and trailing whitespace. `^\s+` matches leading whitespace and `\s+$` matches trailing whitespace. In Perl, you could use `$input =~ s/^\s+|\s+$//g`. Handy use of alternation and /g allows us to do this in a single line of code.

Using ^ and $ as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like `first line\nsecond line` (where \n indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, most regex engines discussed in this tutorial have the option to expand the meaning of both anchors. `^` can then match at the start of the string (before the `f` in the above string), as well as after each line break (between `\n` and `s`). Likewise, `$` still matches at the end of the string (after the last `e`), and also before every line break (between `e` and `\n`).

In text editors like EditPad Pro or GNU Emacs, and regex tools like PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because those applications are designed to work with entire files, rather than short strings. In Ruby and std::regex the caret and dollar also always match at the start and end of each line. In Boost they match at the start and end of each line by default. Boost allows you to turn this off with `regex_constants::no_mod_m` when using the ECMAScript grammar.

In all other programming languages and libraries discussed on this website , you have to explicitly activate this extended functionality. It is traditionally called "multi-line mode". In Perl, you do this by adding an m after the regex code, like this: `m/^regex$/m;`. In .NET, the anchors match before and after newlines when you specify `RegexOptions.Multiline`, such as in `Regex.Match("string", "regex", RegexOptions.Multiline)`.

Line Break Characters

The tutorial page about the dot already discussed which characters are seen as line break characters by the various regex flavors. This affects the anchors just as much when in multi-line mode, and when the dollar matches before the end of the final break. The anchors handle line breaks that consist of a single character the same way as the dot in each regex flavor.

For anchors there's an additional consideration when CR and LF occur as a pair and the regex flavor treats both these characters as line breaks. Delphi, Java, and the JGsoft flavor treat CRLF as an indivisible pair. `^` matches after CRLF and `$` matches before CRLF, but neither match in the middle of a CRLF pair. JavaScript and XPath treat CRLF pairs as two line breaks. `^` matches in the middle of and after CRLF, while `$` matches before and in the middle of CRLF.

Permanent Start of String and End of String Anchors

`\A` only ever matches at the start of the string. Likewise, `\Z` only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on "multiline mode". In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, `\A` and `\Z` only match at the start and the end of the entire file.

JavaScript, POSIX, XML, and XPath do not support `\A` and `\Z`. You're stuck with using the caret and dollar for this purpose.

The GNU extensions to POSIX regular expressions use `\`` (backtick) to match the start of the string, and `\'` (single quote) to match the end of the string.

Strings Ending with a Line Break

Because Perl returns a string with a newline at the end when reading a line from a file, Perl's regex engine matches `$` at the position before the line break at the end of the string even when multi-line mode is turned off. Perl also matches `$` at the very end of the string, regardless of whether that character is a line break. So `^\d+$` matches `123` whether the subject string is `123` or `123\n`.

Most modern regex flavors have copied this behavior. That includes .NET, Java, PCRE, Delphi, PHP, and Python. This behavior is independent of any settings such as "multi-line mode".

In all these flavors except Python, `\Z` also matches before the final line break. If you only want a match at the absolute very end of the string, use `\z` (lowercase z instead of uppercase Z). `\A\d+\z` does not match `123\n`. `\z` matches after the line break, which is not matched by the shorthand character class.

In Python, `\Z` matches only at the very end of the string. Python does not support `\z`.

Strings Ending with Multiple Line Breaks

If a string ends with multiple line breaks and multi-line mode is off then `$` only matches before the last of those line breaks in all flavors where it can match before the final break. The same is true for `\Z` regardless of multi-line mode.

Boost is the only exception. In Boost, `\Z` can match before any number of trailing line breaks as well as at the very end of the string. So if the subject string ends with three line breaks, Boost's `\Z` has four positions that it can match at. Like in all other flavors, Boost's `\Z` is independent of multi-line mode. Boost's `$` only matches at the very end of the string when you turn off multi-line mode (which is on by default in Boost).

Looking Inside The Regex Engine

Let's see what happens when we try to match `^4$` to `749\n486\n4` (where \n represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: `7`. The first token in the regular expression is `^`. Since this token is a zero-length token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. `^` indeed matches the position before `7`. The engine then advances to the next regex token: `4`. Since the previous token was zero-length, the regex engine does *not* advance to the next character in the string. It remains at `7`. `4` is a literal character, which does not match `7`. There are no other permutations of the regex, so the engine starts again with the first regex token, at the next character: `4`. This time, `^` cannot match at the position before the `4`. This position is preceded by a character, and that character is not a newline. The engine continues at `9`, and fails again. The next attempt, at `\n`, also fails. Again, the position before `\n` is preceded by a character, `9`, and that character is not a newline.

Then, the regex engine arrives at the second `4` in the string. The `^` can match at the position before the `4`, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, `4`, but does not advance the character position in the string. `4` matches `4`, and the engine advances both the regex token and the string character. Now the engine attempts to match `$` at the position before (indeed: before) the `8`. The dollar cannot match here, because this position is followed by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second `4`, so the engine continues at the next character, `8`, where the caret does not match. Same at the `6` and the newline.

Finally, the regex engine tries to match the first token at the third `4` in the string. With success. After that, the engine successfully matches `4` with `4`. The current regex token is advanced to `$`, and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-length, so it tries to match the position before the current character. It does not matter that this "character" is the void after the string. In fact, the dollar checks the current character. It must be either a newline, or the void after the string, for `$` to match the position before the current character. Since that is the case after the example, the dollar matches successfully.

Since `$` was the last token in the regex, the engine has found a successful match: the last `4` in the string.

Make a Donation

Page URL: https://www.regular-expressions.info/anchors.html
Page last updated: 30 January 2020
Site last updated: 08 April 2020