Learning Regular Expressions [closed]

Asked 16 years, 4 months ago Modified 2 years, 1 month ago

Viewed 157k times

166

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 13 years ago.

This question's answers are a community effort. Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

I don't really understand regular expressions. Can you explain them to me in an easy-to-follow manner? If there are any online tools or books, could you also link to them?

regex

edited Jan 7, 2015 at 14:39

community wiki
5 revs, 4 users 67%
Teifion

1 Answer

Sorted by: Highest score (default)

858

The most important part is the concepts. Once you understand how the building blocks work, differences in syntax amount to little more than mild dialects. A layer on top of your regular expression engine's syntax is the syntax of the programming language you're using. Languages such as Perl remove most of this complication, but you'll have to keep in mind other considerations if you're using regular expressions in a C program.

If you think of regular expressions as building blocks that you can mix and match as you please, it helps you learn how to write and debug your own patterns but also how to understand patterns written by others.

Start simple

Conceptually, the simplest regular expressions are literal characters. The pattern `N` matches the character 'N'.

Regular expressions next to each other match sequences. For example, the pattern `Nick` matches the sequence 'N' followed by 'i' followed by 'c' followed by 'k'.

If you've ever used `grep` on Unix—even if only to search for ordinary looking strings—you've already been using regular expressions! (The `re` in `grep` refers to regular expressions.)

Order from the menu

Adding just a little complexity, you can match either 'Nick' or 'nick' with the pattern `[Nn]ick`. The part in square brackets is a *character class*, which means it matches exactly one of the enclosed characters. You can also use ranges in character classes, so `[a-c]` matches either 'a' or 'b' or 'c'.

The pattern `.` is special: rather than matching a literal dot only, it matches *any* character[†]. It's the same conceptually as the really big character class `[-.?+%$A-Za-z0-9...]`.

Think of character classes as menus: pick just one.

Helpful shortcuts

Using `.` can save you lots of typing, and there are other shortcuts for common patterns. Say you want to match a digit: one way to write that is `[0-9]`. Digits are a frequent match target, so you could instead use the shortcut `\d`.

Others are `\s` (whitespace) and `\w` (word characters: alphanumerics or underscore).

The uppercased variants are their complements, so `\S` matches any *non*-whitespace character, for example.

Once is not enough

From there, you can repeat parts of your pattern with *quantifiers*. For example, the pattern `ab?c` matches 'abc' or 'ac' because the `?` quantifier makes the subpattern it modifies optional. Other quantifiers are

- `*` (zero or more times)
- `+` (one or more times)
- `{n}` (exactly *n* times)
- `{n,}` (at least *n* times)
- `{n,m}` (at least *n* times but no more than *m* times)

Putting some of these blocks together, the pattern `[Nn]*ick` matches all of

- ick
- Nick
- nick
- Nnick
- nNick
- nnick

- *(and so on)*

The first match demonstrates an important lesson: `*` *always succeeds!* Any pattern can match zero times.

A few other useful examples:

- `[0-9]+` (and its equivalent `\d+`) matches any non-negative integer
- `\d{4}-\d{2}-\d{2}` matches dates formatted like 2019-01-01

Grouping

A quantifier modifies the pattern to its immediate left. You might expect `0abc+0` to match '0abc0', '0abcabc0', and so forth, but the pattern *immediately* to the left of the plus quantifier is `c`. This means `0abc+0` matches '0abc0', '0abcc0', '0abccc0', and so on.

To match one or more sequences of 'abc' with zeros on the ends, use `0(abc)+0`. The parentheses denote a subpattern that can be quantified as a unit. It's also common for regular expression engines to save or "capture" the portion of the input text that matches a parenthesized group. Extracting bits this way is much more flexible and less error-prone than counting indices and `substr`.

Alternation

Earlier, we saw one way to match either 'Nick' or 'nick'. Another is with alternation as in `Nick|nick`. Remember that alternation includes everything to its left and everything to its right. Use grouping parentheses to limit the scope of `|`, *e.g.*, `(Nick|nick)`.

For another example, you could equivalently write `[a-c]` as `a|b|c`, but this is likely to be suboptimal because many implementations assume alternatives will have lengths greater than 1.

Escaping

Although some characters match themselves, others have special meanings. The pattern `\d+` doesn't match backslash followed by lowercase D followed by a plus sign: to get that, we'd use `\\d\+`. A backslash removes the special meaning from the following character.

Greediness

Regular expression quantifiers are greedy. This means they match as much text as they possibly can while allowing the entire pattern to match successfully.

For example, say the input is

> "Hello," she said, "How are you?"

You might expect `".+"` to match only 'Hello,' and will then be surprised when you see that it matched from 'Hello' all the way through 'you?'.

To switch from greedy to what you might think of as cautious, add an extra `?` to the quantifier. Now you understand how `\((.+?)\)`, the example from your question works. It matches the sequence of a literal left-parenthesis, followed by one or more characters, and terminated by a right-parenthesis.

If your input is '(123) (456)', then the first capture will be '123'. Non-greedy quantifiers want to allow the rest of the pattern to start matching as soon as possible.

(As to your confusion, I don't know of any regular-expression dialect where `((.+?))` would do the same thing. I suspect something got lost in transmission somewhere along the way.)

Anchors

Use the special pattern `^` to match only at the beginning of your input and `$` to match only at the end. Making "bookends" with your patterns where you say, "I know what's at the front and back, but give me everything between" is a useful technique.

Say you want to match comments of the form

> `-- This is a comment --`

you'd write `^--\s+(.+)\s+--$`.

Build your own

Regular expressions are recursive, so now that you understand these basic rules, you can combine them however you like.

Tools for writing and debugging regexes:

- RegExr (for JavaScript)
- Perl: YAPE: Regex Explain
- Regex Coach (engine backed by CL-PPCRE)
- RegexPal (for JavaScript)
- Regular Expressions Online Tester
- Regex Buddy
- Regex 101 (for PCRE, JavaScript, Python, Golang, Java 8)
- I Hate Regex
- Visual RegExp
- Expresso (for .NET)
- Rubular (for Ruby)

- Regular Expression Library (Predefined Regexes for common scenarios)
- Txt2RE
- Regex Tester (for JavaScript)
- Regex Storm (for .NET)
- Debuggex (visual regex tester and helper)

Books

- Mastering Regular Expressions, the 2nd Edition, and the 3rd edition.
- Regular Expressions Cheat Sheet
- Regex Cookbook
- Teach Yourself Regular Expressions

Free resources

- RegexOne - Learn with simple, interactive exercises.
- Regular Expressions - Everything you should know (PDF Series)
- Regex Syntax Summary
- How Regexes Work
- JavaScript Regular Expressions

Footnote

†: The statement above that `.` matches any character is a simplification for pedagogical purposes that is not strictly true. Dot matches any character except newline, `"\n"`, but in practice you rarely expect a pattern such as `.+` to cross a newline boundary. Perl regexes have a `/s` switch and Java `Pattern.DOTALL`, for example, to make `.` match any character at all. For languages that don't have such a feature, you can use something like `[\s\S]` to match "any whitespace or any non-whitespace", in other words anything.

Share Improve this answer Follow

edited Nov 11, 2022 at 14:30

community wiki
19 revs, 18 users 52%
Greg Bacon

17 You can also use trial and error method and than following online regex tester and debugger can be a huge help: regex101.com – Juraj.Lorinc Sep 9, 2015 at 10:25

3 It would be worth mentioning that, despite being a similar pattern, `a{,m}` isn't a thing, at least in Javascript, Perl, and Python. – anon Mar 31, 2016 at 12:12

4 It would be very worth to mention that there are different kind of regular expression engines with all have different feature set's and syntactic rules. – hek2mgl Nov 14, 2016 at 18:14

1 hackr.io/tutorials/learn-regular-expressions-regex is a great place to find best online regex tutorials. All the tutorials here are submitted and recommended (upvoted like SO) by the programming community. – Saurabh Hooda Aug 16, 2017 at 7:14

1 This can be helpful for quick reference: Quick-Start: Regex Cheat Sheet – Lod Jul 1, 2019 at 7:21