Regex tutorial — A quick cheatsheet by examples

Jonny Fox
4 minutes
UPDATE! Check out my new REGEX COOKBOOK about the most commonly used (and most wanted) regex

Regular expressions (regex or regexp) are extremely useful in **extracting information from any text** by searching for one or more matches of a specific search pattern (i.e. a specific sequence of ASCII or unicode characters).

Fields of application range from validation to parsing/replacing strings, passing through translating data to other formats and web scraping.

One of the most interesting features is that once you've learned the syntax, you can actually use this tool in (almost) all programming languages (JavaScript, Java, VB, C #, C / C++, Python, Perl, Ruby, Delphi, R, Tcl, d many others) with the slightest distinctions about the support of the most advanced features and syntax versions supported by the engines).

Let's start by looking at some examples and explanations.

Anchors — ^ and $

```
^The        matches any string that starts with The -> Try it!
end$        matches a string that ends with end
^The end$   exact string match (starts and ends with The end)
roar        matches any string that has the text roar in it
```

Quantifiers — * + ? and {}

```
abc*        matches a string that has ab followed by zero or more c -> Try it!
abc+        matches a string that has ab followed by one or more c
abc?        matches a string that has ab followed by zero or one c
abc{2}      matches a string that has ab followed by 2 c
abc{2,}     matches a string that has ab followed by 2 or more c
abc{2,5}    matches a string that has ab followed by 2 up to 5 c
a(bc)*      matches a string that has a followed by zero or more copies of the sequence bc
a(bc){2,5}  matches a string that has a followed by 2 up to 5 copies of the sequence bc
```

OR operator — | or []

```
a(b|c)      matches a string that has a followed by b or c -> Try it!
a[bc]       same as previous
```

Character classes — \d \w \s and .

```
\d          matches a single character that is a digit -> Try it!
\w          matches a word character (alphanumeric character plus underscore) -> Try it!
\s          matches a whitespace character (includes tabs and line breaks)
.           matches any character -> Try it!
```

Use the `.` operator carefully since often class or negated character class (which we'll cover next) are faster and more precise.

`\d`, `\w` and `\s` also present their negations with `\D`, `\W` and `\S` respectively.

For example, `\D` will perform the inverse match with respect to that obtained with `\d`.

```
\D          matches a single non-digit character -> Try it!
```

In order to be taken literally, you must escape the characters `^.[$()|*+?{\` with a backslash `\` as they have special meaning.

`\$\d` matches a string that has a **$ before one digit** -> Try it!

Notice that you can match also **non-printable characters** like tabs `\t`, new-lines `\n`, carriage returns `\r`.

Flags

We are learning how to construct a regex but forgetting a fundamental concept: **flags**.

A regex usually comes within this form `/abc/`, where the search pattern is delimited by two slash characters `/`. At the end we can specify a flag with these values (we can also combine them each other):

- **g** (global) does not return after the first match, restarting the subsequent searches from the end of the previous match
- **m** (multi-line) when enabled `^` and `$` will match the start and end of a line, instead of the whole string
- **i** (insensitive) makes the whole expression case-insensitive (for instance **/aBc/i** would match **AbC**)

Intermediate topics

Grouping and capturing — ()

`a(bc)` parentheses create a **capturing group with value bc** -> Try it!
`a(?:bc)*` using **?:** we **disable the capturing group** -> Try it!
`a(?<foo>bc)` using **?<foo>** we put a name to the group -> Try it!

This operator is very useful when we need to extract information from strings or data using your preferred programming language. Any multiple occurrences captured by several groups will be exposed in the form of a classical array: we will access their values specifying using an index on the result of the match.

If we choose to put a name to the groups (using `(?<foo>...)`) we will be able to retrieve the group values using the match result like a dictionary where the keys will be the name of each group.

Bracket expressions—[]

`[abc]` matches a string that has **either an a or a b or a c** -> is the **same as a|b|c** -> Try it!

`[a-c]` same as previous

`[a-fA-F0-9]` a string that represents **a single hexadecimal digit, case insensitively** -> Try it!

`[0-9]%` a string that has a character **from 0 to 9 before a % sign**

`[^a-zA-Z]` a string that has **not a letter from a to z or from A to Z.** In this case the `^` is used as **negation of the expression** -> Try it!

Remember that inside bracket expressions all special characters (including the backslash `\`) lose their special powers: thus we will not apply the “escape rule”.

Greedy and Lazy match

The quantifiers (`* + {}`) are greedy operators, so they expand the match as far as they can through the provided text.

For example, `<.+>` matches `<div>simple div</div>` in `This is a <div> simple div</div> test`. In order to catch only the `div` tag we can use a `?` to make it lazy:

`<.+?>` matches **any character one or more** times included **inside <** and **>**, **expanding as needed** -> **Try it!**

Notice that a better solution should avoid the usage of `.` in favor of a more strict regex:

`<[^<>]+>` matches **any character except < or > one or more** times included **inside <** and **>** -> **Try it!**

Advanced topics

Boundaries — \b and \B

`\b`abc`\b` performs a **"whole words only" search** -> **Try it!**

`\b` represents an **anchor like caret** (it is similar to $ and ^) matching positions where **one side is a word character** (like `\w`) and the **other side is not a word character** (for instance it may be the beginning of the string or a space character).

It comes with its **negation**, `\B`. This matches all positions where `\b` doesn't match and could be if we want to find a search pattern fully surrounded by word characters.

`\B`abc`\B` matches only if the pattern is **fully surrounded by word** characters -> **Try it!**

Back-references — \1

`(`[abc]`)\1` using **\1** it matches **the same** text **that was matched by the first capturing group** -> **Try it!**

([abc])`(`[de]`)\2`\1 we can use **\2** (\3, \4, etc.) to identify **the same** text that **was matched by the second** (third, fourth, etc.) **capturing group** -> **Try it!**

`(?<foo>`[abc]`)\k<foo>` we put the name **foo t**o the group and we reference it later (**\k<foo>**). The result is the same of the first regex -> **Try it!**

Look-ahead and Look-behind — (?=) and (?<=)

d`(?=`r`)` matches a **d** only if is **followed by r**, **but r will not be** part of the overall regex **match** -> **Try it!**

`(?<=`r`)`d matches a **d** only if is **preceded by an r**, **but r will not be** part of the overall regex **match** -> **Try it!**

You can use also the negation operator!

d`(?!`r`)` matches a **d** only if is **not followed by r**, **but r will not be** part of the overall regex **match** -> **Try it!**

`(?<!`r`)`d matches a **d** only if is **not preceded by an r**, **but r will not be** part of the overall regex **match** -> **Try it!**

Summary

As you’ve seen, the application fields of regex can be multiple and I’m sure that you’ve recognized at least one of these tasks among those seen in your developer career, here a quick list:

- data validation (for example check if a time string i well-formed)
- data scraping (especially web scraping, find all pages that contain a certain set of words eventually in a specific order)
- data wrangling (transform data from “raw” to another format)
- string parsing (for example catch all URL GET parameters, capture text inside a set of parenthesis)
- string replacement (for example, even during a code session using a common IDE to translate a Java or C# class in the respective JSON object — replace “;” with “,” make it lowercase, avoid type declaration, etc.)
- syntax highlightning, file renaming, packet sniffing and many other applications involving strings (where data need not be textual)