

Perl Compatible Regular Expressions in a Nutshell

Syntax

All features on this cheat sheet are available in latest Perl, however, the implementation and UTF-8 support in other environments may differ slightly, e.g. Ruby supports named groups and UTF-8 only since version 1.9.1.

Perl:

```
$string =~ m/pattern/modifier
$string =~ s/pattern/replace/flags
```

<http://www.perl.com/doc/manual/html/pod/perlre.html>

Ruby:

```
string.match(/pattern/flags) or string.match(/pattern/flags) {|match| ... }
string.sub(/pattern/flags, replace) or string.sub(/pattern/flags) {|replace| ... }
string.gsub(/pattern/flags, replace) or string.gsub(/pattern/flags) {|replace| ... }
```

<http://www.apidock.com/ruby/String>

Python:

```
import re
re.search("(?flags)pattern", string)
re.sub("(?flags)pattern", replace, string, count)
```

<http://docs.python.org/library/re.html>

PHP:

```
preg_match('/pattern/flags', $string)
preg_replace('/pattern/flags', 'replace', $string)
```

<http://www.php.net/manual/en/ref.pcre.php>

Java:

```
string.matches("(?flags)pattern")
string.replaceFirst("(?flags)pattern", "replace")
string.replaceAll("(?flags)pattern", "replace")
```

<http://java.sun.com/javase/6/docs/api/java/util/regex/package-summary.html>

JavaScript:

```
string.match(/pattern/flags)
string.replace(/pattern/flags, "replace")
```

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Objects:RegExp

C++: (requires PCRE library)

```
pcrcpp::RE("(?flags)pattern").PartialMatch(string)
pcrcpp::RE("(?flags)pattern").Replace("replace", &string)
```

<http://www.pcre.org>

Objective-C: (requires RegexpKit framework)

```
[string isMatchedByRegex:@"(?flags)pattern"]
[string stringByMatching:@"(?flags)pattern" replace:1 withString:@"replace"]
[string stringByMatching:@"(?flags)pattern" replace:RKReplaceAll withString:@"replace"]
```

<http://regexkit.sourceforge.net>

Shell:

```
grep -P "(?flags)pattern" file.txt
```

<http://www.gnu.org/software/grep/>

Characters

These are the usual suspects well known from any C-ish language:

a	match the character a
3	match the number 3
\$a or #{a}	match the contents of a variable \$a or a respectively
\n	newline (NL, LF)
\r	return (CR)
\f	form feed (FF)
\t	tab (TAB)
\x3C	character with the hex code 3C
\u561A	character with the hex code 561A
\e	escape character (alias \u001B)
\c...	control character

Wildcards

Wildcards match if a character belongs to the designated class of characters:

.	match any character
\...	quote single metacharacter: \. matches a dot instead of any character and \\ matches a single backslash
\w	alphanumeric + underscore (shortcut for [0-9a-zA-Z_])
\W	any character not covered by \w
\d	numeric (shortcut for [0-9])
\D	any character not covered by \d
\s	whitespace (shortcut for [\t\n\r\f])
\S	any character not covered by \s
[...]	any character listed: [a5!d-g] means a, 5, ! and d, e, f, g
[^...]	any character not listed: [^a5!d-g] means anything but a, 5, ! and d, e, f, g

Boundaries

Boundaries match the spots between characters and therefore have no width of their own (also called zero-width, → extensions):

\b	matches at a word boundary (spot between \w and \W)
\B	matches anything but a word boundary
^	matches at the beginning of a line (m) or entire string (s)

<code>\A</code>	matches at the beginning of the entire string
<code>\$</code>	matches at the end of a line (<code>m</code>) or entire string (<code>s</code>)
<code>\Z</code>	matches at the end of the entire string ignoring a trailing <code>\n</code>
<code>\z</code>	matches at the end of the entire string
<code>\G</code>	matches where the previous regex call left off (→ flag <code>g</code>)

Grouping

Any of the above constructs can be grouped to improve readability and create a reference for use in `pattern` or `replace` (→ replacing):

<code>(...)</code>	the group is assigned to the references <code>\1</code> and <code>\1</code> as well as <code>\$1</code> (outside of the regex context)
<code>(...)</code> etc	first group is <code>\1</code> , <code>\1</code> and <code>\$1</code> , second group is <code>\2</code> , <code>\2</code> and <code>\$2</code> etc
<code>(...)</code>	matches if one of the group options matches and assigns it to <code>\1</code> , <code>\1</code> and <code>\$1</code>
<code>(?<name>...)</code>	the group is named <code>name</code> and assigned to the references <code>\g<name></code> and <code>\k<name></code> (Python uses <code>(P?<name>...)</code> instead)

You can use named groups to make complex regular expressions much more readable. The following example will match IPv4 entries from a hosts file, e.g. "myhost 192.168.1.1". Whitespaces are ignored due to the `x` flag (→ flags), the `{0}` on the first three lines in effect turns those named groups into mere placeholders and the actual pattern is but on the fourth line:

```
/ (?<host> [a-z.-]+ ) {0}
  (?<byte> \d{1,3} ) {0}
  (?<ip> ( (\g<byte>\. ) {3} \g<byte> ) {0}
  \g<ip>\s+\g<host>/x
```

Named group matches are available outside of the regex context as well, however, this is of course implemented differently per each programming language, here's an example for Ruby 1.9.1:

```
m = 'aabb33dd'.match(/(?<numbers>\d+)/)
puts m[:numbers] # returns '33'
```

Extensions

Less common functionality is covered by extensions using the `(?...)` syntax. Extensions do not create a reference like grouping does.

<code>(?:... ...)</code>	same as grouping, but no reference is created
<code>(?=...)</code>	zero-width positive lookahead assertion
<code>(?!...)</code>	zero-width negative lookahead assertion
<code>(?<=...)</code>	zero-width positive lookbehind assertion (no quantifiers allowed within)
<code>(?<!...)</code>	zero-width negative lookbehind assertion (no quantifiers allowed within)
<code>(?>...)</code>	zero-width independent subexpression
<code>(? (...))</code>	conditional expression
<code>(?flags)</code>	apply the flag(s) within the current group from this point forward (→ flags)
<code>(?flags:...)</code>	apply the flag(s) for this pattern (no backreference created!)
<code>(?#...)</code>	zero-width comment (no round brackets allowed in comment text)

Quantifiers

Most of the above constructs may be quantified by adding one of the following symbols after them:

<code>?</code>	match 1 or 0 times
<code>*</code>	0 or more times
<code>+</code>	1 or more times
<code>{n}</code>	exactly <code>n</code> times
<code>{n,}</code>	at least <code>n</code> times
<code>{n,m}</code>	at least <code>n</code> but not more than <code>m</code> times, as often as possible

Greediness

This is a very important feature, ignore it and you will be destined to produce clumsy and error prone regex! Quantifiers are greedy by default which means they match as often as possible. Limit their hunger by adding a `?` after them. Here's an example applied to the title of this section:

```
G.*e      matches to Greedyne
G.*?e     matches to Gre
```

Quoting

You want to ignore all of the above for a while? Here you go:

<code>\...</code>	quote single metacharacter: <code>\.</code> matches a dot instead of any character
<code>\Q ... \E</code>	ignore all regex metacharacters in between

Replacing

The following symbols have special meanings in the `replace` part:

<code>\1</code> , <code>\2</code> etc	include the contents of the corresponding group (→ grouping)
<code>\{1\}000</code>	same as the above, use curly brackets if numbers follow the symbol
<code>\l</code>	lowercase the following character
<code>\L ... \E</code>	lowercase all characters in between
<code>\u</code>	uppercase the following character
<code>\U ... \E</code>	uppercase all characters in between

Flags

Optional flags determine the behaviour of the regex as a whole. May be used within the `(?flags)` construct (→ extensions):

<code>i</code>	case-insensitive pattern matching
<code>m</code>	multiple lines: <code>.</code> does not match <code>\n</code> (Ruby uses this per default)
<code>s</code>	single line: <code>.</code> matches <code>\n</code> (Ruby uses <code>m</code> for this instead)
<code>x</code>	ignore whitespaces in pattern for better readability

The following cannot be used within the `(?flags)` construct:

<code>g</code>	apply the regex as many times as possible (i.e. for global replace)
<code>e</code>	evaluate the <code>replace</code> part as if it were source code !! DANGER !!
<code>o</code>	compile the pattern only once and therefore perform variable substitutions only once