Regular Expressions for Perl, C, PHP, Python, Java, and .NET

Regular Expression

Pocket Reference

O'REILLY®

Tony Stubblebine

Regular Expression
Pocket Reference

Regular Expression
Pocket Reference

Tony Stubblebine

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

PHP

This reference covers PHP 4.3's Perl-style regular expression support contained within the `preg` routines. PHP also provides POSIX-style regular expressions, but these do not offer additional benefit in power or speed. The `preg` routines use a Traditional NFA match engine. For an explanation of the rules behind an NFA engine, see "Introduction to Regexes and Pattern Matching."

Supported Metacharacters

PHP supports the metacharacters and metasequences listed in Tables 31 through 35. For expanded definitions of each metacharacter, see "Regex Metacharacters, Modes, and Constructs."

Table 31. Character representations

Sequence	Meaning
`\a`	Alert (bell), `x07`.
`\b`	Backspace, `x08`, supported only in character class.
`\e`	ESC character, `x1B`.
`\n`	Newline, `x0A`.
`\r`	Carriage return, `x0D`.
`\f`	Form feed, `x0C`.
`\t`	Horizontal tab, `x09`
`\`*`octal`*	Character specified by a three-digit octal code.
`\x`*`hex`*	Character specified by a one- or two-digit hexadecimal code.
`\x{`*`hex`*`}`	Character specified by any hexadecimal code.
`\c`*`char`*	Named control character.

Table 32. Character classes and class-like constructs

Class	Meaning
`[...]`	A single character listed or contained within a listed range.

Table 32. Character classes and class-like constructs

Class	Meaning
`[^...]`	A single character not listed and not contained within a listed range.
`[:`*`class`*`:]`	POSIX-style character class valid only within a regex character class.
`.`	Any character except newline (unless single-line mode, `/s`).
`\C`	One byte; however, this may corrupt a Unicode character stream.
`\w`	Word character, `[a-zA-z0-9_]`.
`\W`	Non-word character, `[^a-zA-z0-9_]`.
`\d`	Digit character, `[0-9]`.
`\D`	Non-digit character, `[^0-9]`.
`\s`	Whitespace character, `[\n\r\f\t ]`.
`\S`	Non-whitespace character, `[^\n\r\f\t ]`.

Table 33. Anchors and zero-width tests

Sequence	Meaning
`^`	Start of string, or after any newline if in multiline match mode, `/m`.
`\A`	Start of search string, in all match modes.
`$`	End of search string or before a string-ending newline, or before any newline if in multiline match mode, `/m`.
`\Z`	End of string or before a string-ending newline, in any match mode.
`\z`	End of string, in any match mode.
`\G`	Beginning of current search.
`\b`	Word boundary; position between a word character (`\w`) and a non-word character (`\W`), the start of the string, or the end of the string.
`\B`	Not-word-boundary.
`(?=...)`	Positive lookahead.
`(?!...)`	Negative lookahead.
`(?<=...)`	Positive lookbehind.

Table 33. Anchors and zero-width tests

Sequence	Meaning
`(?<!...)`	Negative lookbehind.

Table 34. Comments and mode modifiers

Modes	Meaning
`i`	Case-insensitive matching.
`m`	`^` and `$` match next to embedded `\n`.
`s`	Dot (`.`) matches newline.
`x`	Ignore whitespace and allow comments (`#`) in pattern.
`U`	Inverts greediness of all quantifiers: `*` becomes lazy and `*?` greedy.
`A`	Force match to start at search start in subject string.
`D`	Force `$` to match end of string instead of before the string ending newline. Overridden by multiline mode.
`u`	Treat regular expression and subject strings as strings of multi-byte UTF-8 characters.
`(?mode)`	Turn listed modes (`imsxU`) on for the rest of the subexpression.
`(?-mode)`	Turn listed modes (`imsxU`) off for the rest of the subexpression.
`(?mode:...)`	Turn mode (`xsmi`) on within parentheses.
`(?-mode:...)`	Turn mode (`xsmi`) off within parentheses.
`(?#...)`	Treat substring as a comment.
`#...`	Rest of line is treated as a comment in `x` mode.
`\Q`	Quotes all following regex metacharacters.
`\E`	Ends a span started with `\Q`.

Table 35. Grouping, capturing, conditional, and control

Sequence	Meaning
`(...)`	Group subpattern and capture submatch into `\1`,`\2`,`...`
`(?P<name>...)`	Group subpattern and capture submatch into named capture group, *name*.
`\n`	Contains the results of the *n*th earlier submatch from a parentheses capture group or a named capture group.

Table 35. Grouping, capturing, conditional, and control (continued)

Sequence	Meaning
`(?:...)`	Groups subpattern, but does not capture submatch.
`(?>...)`	Disallow backtracking for text matched by subpattern.
`...|...`	Try subpatterns in alternation.
`*`	Match 0 or more times.
`+`	Match 1 or more times.
`?`	Match 1 or 0 times.
`{`*n*`}`	Match exactly *n* times.
`{`*n*`,}`	Match at least *n* times.
`{`*x*`,`*y*`}`	Match at least *x* times but no more than *y* times.
`*?`	Match 0 or more times, but as few times as possible.
`+?`	Match 1 or more times, but as few times as possible.
`??`	Match 0 or 1 time, but as few times as possible.
`{`*n*`,}?`	Match at least *n* times, but as few times as possible.
`{`*x*`,`*y*`}?`	Match at least *x* times, no more than *y* times, and as few times as possible.
`*+`	Match 0 or more times, and never backtrack.
`++`	Match 1 or more times, and never backtrack.
`?+`	Match 0 or 1 times, and never backtrack.
`{`*n*`}+`	Match at least *n* times, and never backtrack.
`{`*n*`,}+`	Match at least *n* times, and never backtrack.
`{`*x*`,`*y*`}+`	Match at least *x* times, no more than *y* times, and never backtrack.
`(?(`*condition*`)...|...)`	Match with if-then-else pattern. The *condition* can be either the number of a capture group or a lookahead or lookbehind construct.
`(?(`*condition*`)...)`	Match with if-then pattern. The *condition* can be either the number of a capture group or a lookahead or lookbehind construct.

Pattern-Matching Functions

PHP provides several standalone functions for pattern matching. When creating regular expression strings, you need to escape embedded backslashes; otherwise, the backslash is interpreted in the string before being sent to the regular expression engine.

`array preg_grep (string` *`pattern`*`, array` *`input`*`)`

Return array containing every element of *`input`* matched by *`pattern`*.

`int preg_match_all (string` *`pattern`*`, string` *`subject`*`, array` *`matches [,`* `int` *`flags]`*`)`

Search for all matches of *`pattern`* against *`string`* and return the number of matches. The matched substrings are placed in the *`matches`* array. The first element of *`matches`* is an array containing the text of each full match. Each additional element *`N`* of *`matches`* is an array containing the *`N`*th capture group match for each full match. So `matches[7][3]` contains the text matches by the seventh capture group in the fourth match of *`pattern`* in *`string`*.

The default ordering of *`matches`* can be set explicitly with the `PREG_SET_ORDER` flag. `PREG_SET_ORDER` sets a more intuitive ordering where each element of *`matches`* is an array corresponding to a match. The zero element of each array is the complete match, and each additional element corresponds to a capture group. The additional flag `PREG_OFFSET_CAPTURE` causes each array element containing a string to be replaced with a two-element array containing the same string and starting character position in *`subject`*.

`int preg_match (string` *`pattern`*`, string` *`subject [,`* `array` *`matches [,`* `int` *`flags]]`*`)`

Return 1 if *`pattern`* matches in *`subject`*, otherwise return 0. If the *`matches`* array is provided, the matched substring is placed in `matches[0]` and any capture group matches are placed in subsequent elements. One allowed flag, `PREG_OFFSET_CAPTURE`, causes elements of *`matches`* to be replaced with a two-element array containing the matched string and starting character position of the match.

`string preg_quote (string` *`str`* `[, string` *`delimiter`*`])`

Return a *`str`* with all regular expression metacharacters escaped. Provide the *`delimiter`* parameter if you are using optional delimiters with your regular expression and need the delimiter escaped in *`str`*.

`mixed preg_replace_callback (mixed` *`pattern`*, `callback` *`callback`*, `mixed` *`subject`* *[*, `int` *`limit`*]`)`

Return text of *`subject`* with every occurrence of *`pattern`* replaced with the results of *`callback`*. The callback should take one parameter, an array containing the matched text and any matches from capture groups. If provided, the function performs no more than *`limit`* replacements. If *`pattern`* has the `/e` modifier, *`replacement`* is parsed for reference substitution and then executed as PHP code.

If *`pattern`* is an array, each element is replaced with *`callback`*. If *`subject`* is an array, the function iterates over each element.

`mixed preg_replace (mixed` *`pattern`*, `mixed` *`replacement`*, `mixed` *`subject`* *[*, `int` *`limit`*]`)`

Return text of *`subject`* with every occurrence of *`pattern`* replaced with *`replacement`*. If provided, the function performs no more than *`limit`* replacements. The replacement string may refer to the match or capture group matches with `$n` (preferred) or `\n` (deprecated). If *`pattern`* has the `/e` modifier, *`replacement`* is parsed for reference substitution and then executed as PHP code.

If *`pattern`* is an array, then each element is replaced with *`replacement`* or, if *`replacement`* is an array, the corresponding element in *`replacement`*. If *`subject`* is an array, the function iterates over each element.

`array preg_split (string` *`pattern`*`, string` *`subject`* *[*`, int` *`limit`* *[*`, int` *`flags`**]]*`)`

Return an array of strings broken around *`pattern`*. If specified, `preg_split( )` returns no more than *`limit`* substrings. A *`limit`* is the same as "no limit," allowing you to set flags. Available flags are: `PREG_SPLIT_NO_EMPTY`, return only non-empty pieces; `PREG_SPLIT_DELIM_CAPTURE`, return captured submatches after each split substring; and `PREG_SPLIT_OFFSET_CAPTURE`, return an array of two-element arrays where the first element is the match and the second element is the offset of the match in *`subject`*.

Examples

Example 19. Simple match

```
//Match Spider-Man, Spiderman, SPIDER-MAN, etc.
$dailybugle = "Spider-Man Menaces City!";

$regex = "/spider[- ]?man/i";

if (preg_match($regex, $dailybugle)) {
      //do something
}
```

Example 20. Match and capture group

```
//Match dates formatted like MM/DD/YYYY, MM-DD-YY,...
$date = "12/30/1969";
$p    = "!(\\d\\d)[-/](\\d\\d)[-/](\\d\\d(?:\\d\\d)?)!";

if (preg_match($p,$date,$matches) {
      $month = $matches[1];
      $day   = $matches[2];
      $year  = $matches[3];
}
```

Example 21. Simple substitution

```
//Convert <br> to <br /> for XHTML compliance
$text = "Hello world. <br>";

$pattern  = "{<br>}i";
```

Example 21. Simple substitution (continued)

```
echo preg_replace($pattern, "<br />", $text);
```

Example 22. Harder substitution

```
//urlify - turn URL's into HTML links
$text = "Check the website, http://www.oreilly.com/catalog/
repr.";
$regex =
        "{ \\b                        # start at word\n"
    .   "                             # boundary\n"
    .   "(                            # capture to $1\n"
    .   "(https?|telnet|gopher|file|wais|ftp) : \n"
    .   "                             # resource and colon\n"
    .   "[\\w/\\#~:.?+=&%@!\\-]+?   # one or more valid\n"
    .   "                             # characters\n"
    .   "                             # but take as little as\n"
    .   "                             # possible\n"
    .   ")\n"
    .   "(?=                          # lookahead\n"
    .   "[.:?\\-]*                    # for possible punct\n"
    .   "(?:[^\\w/\\#~:.?+=&%@!\\-] # invalid character\n"
    .   "|$)                          # or end of string\n"
    .   ") }x";


echo preg_replace($regex, "<a href=\"$1\">$1</a>", $text);
```

Other Resources

- PHP's online documentation at *http://www.php.net/pcre*.