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

Regular Expression

Pocket Reference

O'REILLY®

Tony Stubblebine

Regular Expression
Pocket Reference

Tony Stubblebine

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Regular Expression Pocket Reference

Regular expressions (known as regexps or regexes) are a way to describe text through pattern matching. You might want to use regular expressions to validate data, to pull pieces of text out of larger blocks, or to substitute new text for old text.

Regular expression syntax defines a language you use to describe text. Today, regular expressions are included in most programming languages as well as many scripting languages, editors, applications, databases, and command-line tools. This book aims to give quick access to the syntax and pattern-matching operations of the most popular of these languages.

About This Book

This book starts with a general introduction to regular expressions. The first section of this book describes and defines the constructs used in regular expressions and establishes the common principles of pattern matching. The remaining sections of the book are devoted to the syntax, features, and usage of regular expressions in various implementations.

The implementations covered in this book are Perl, Java, .NET and C#, Python, PCRE, PHP, the *vi* editor, JavaScript, and shell tools.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Used for emphasis, new terms, program names, and URLs

`Constant width`
: Used for options, values, code fragments, and any text that should be typed literally

`Constant width italic`
: Used for text that should be replaced with user-supplied values

Acknowledgments

The world of regular expressions is complex and filled with nuance. Jeffrey Friedl has written the definitive work on the subject, *Mastering Regular Expressions* (O'Reilly), a work on which I relied heavily when writing this book. As a convenience, this book provides page references to *Mastering Regular Expressions*, Second Edition (MRE) for expanded discussion of regular expression syntax and concepts.

This book simply would not have been written if Jeffrey Friedl had not blazed a trail ahead of me. Additionally, I owe him many thanks for allowing me to reuse the structure of his book and for his suggestions on improving this book. Nat Torkington's early guidance raised the bar for this book. Philip Hazel, Ron Hitchens, A.M. Kuchling, and Brad Merrill reviewed individual chapters. Linda Mui saved my sanity and this book. Tim Allwine's constant regex questions helped me solidify my knowledge of this topic. Thanks to Schuyler Erle and David Lents for letting me bounce ideas off of them. Lastly, many thanks to Sarah Burcham for her contributions to the "Shell Tools" sections and for providing the inspiration and opportunity to work and write for O'Reilly.

Introduction to Regexes and Pattern Matching

A *regular expression* is a string containing a combination of normal characters and special metacharacters or metasequences. The normal characters match themselves. *Metacharacters* and *metasequences* are characters or sequences of characters that represent ideas such as quantity, locations, or types of characters. The list in the section "Regex Metacharacters, Modes, and Constructs" shows the most common metacharacters and metasequences in the regular expression world. Later sections list the availability of and syntax for supported metacharacters for particular implementations of regular expressions.

Pattern matching consists of finding a section of text that is described (matched) by a regular expression. The underlying code that searchs the text is the *regular expression engine*. You can guess the results of most matches by keeping two rules in mind:

1. *The earliest (leftmost) match wins*
   Regular expressions are applied to the input starting at the first character and proceeding toward the last. As soon as the regular expression engine finds a match, it returns. (See MRE 148-149, 177–179.)

2. *Standard quantifiers are greedy*
   Quantifiers specify how many times something can be repeated. The standard quantifiers attempt to match as many times as possible. They settle for less than the maximum only if this is necessary for the success of the match. The process of giving up characters and trying less-greedy matches is called backtracking. (See MRE 151–153.)

Regular expression engines have subtle differences based on their type. There are two classes of engines: Deterministic Finite Automaton (DFA) and Nondeterministic Finite Automaton (NFA). DFAs are faster but lack many of the features of

an NFA, such as capturing, lookaround, and non-greedy quantifiers. In the NFA world there are two types: Traditional and POSIX.

DFA engines
: DFAs compare each character of the input string to the regular expression, keeping track of all matches in progress. Since each character is examined at most once, the DFA engine is the fastest. One additional rule to remember with DFAs is that the alternation metasequence is greedy. When more than one option in an alternation (`foo|foobar`) matches, the longest one is selected. So, rule #1 can be amended to read "the longest leftmost match wins." (See MRE 155–156.)

Traditional NFA engines
: Traditional NFA engines compare each element of the regex to the input string, keeping track of positions where it chose between two options in the regex. If an option fails, the engine backtracks to the most recently saved position. For standard quantifiers, the engine chooses the greedy option of matching more text; however, if that option leads to the failure of the match, the engine returns to a saved position and tries a less greedy path. The traditional NFA engine uses ordered alternation, where each option in the alternation is tried sequentially. A longer match may be ignored if an earlier option leads to a successful match. So, rule #1 can be amended to read "the first leftmost match after greedy quantifiers have had their fill." (See MRE 153–154.)

POSIX NFA engines
: POSIX NFA Engines work similarly to Traditional NFAs with one exception: a POSIX engine always picks the longest of the leftmost matches. For example, the alternation `cat|category` would match the full word "category" whenever possible, even if the first alternative ("cat") matched and appeared earlier in the alternation. (See MRE 153–154.)

Regex Metacharacters, Modes, and Constructs

The metacharacters and metasequences shown here represent most available types of regular expression constructs and their most common syntax. However, syntax and availability vary by implementation.

Character representations

Many implementations provide shortcuts to represent some characters that may be difficult to input. (See MRE 114–117.)

Character shorthands
: Most implementations have specific shorthands for the `alert`, `backspace`, `escape character`, `form feed`, `newline`, `carriage return`, `horizontal tab`, and `vertical tab` characters. For example, `\n` is often a shorthand for the newline character, which is usually LF (012 octal) but can sometimes be CR (15 octal) depending on the operating system. Confusingly, many implementations use `\b` to mean both `backspace` and word boundary (between a "word" character and a non-word character). For these implementations, `\b` means `backspace` in a character class (a set of possible characters to match in the string) and word boundary elsewhere.

Octal escape: `\num`
: Represents a character corresponding to a two- or three-octal digit number. For example, `\015\012` matches an ASCII CR/LF sequence.

Hex and Unicode escapes: `\x`num, `\x{`num`}`, `\u`num, `\U`num
: Represents a character corresponding to a hexadecimal number. Four-digit and larger hex numbers can represent the range of Unicode characters. For example, `\x0D\x0A` matches an ASCII CR/LF sequence.

Control characters: `\c`*char*

Corresponds to ASCII control characters encoded with values less than 32. To be safe, always use an uppercase *char*—some implementations do not handle lowercase representations. For example, `\cH` matches Control-H, an ASCII backspace character.

Character classes and class-like constructs

Character classes are ways to define or specify a set of characters. A character class matches one character in the input string that is within the defined set. (See MRE 117–127.)

Normal classes: `[...]` *and* `[^...]`

Character classes, `[...]`, and negated character classes, `[^...]`, allow you to list the characters that you do or do not want to match. A character class always matches one character. The - (dash) indicates a range of characters. To include the dash in the list of characters, list it first or escape it. For example, `[a-z]` matches any lowercase ASCII letter.

Almost any character: dot (.)

Usually matches any character except a newline. The match mode can often be changed so that dot also matches newlines.

Class shorthands: `\w`, `\d`, `\s`, `\W`, `\D`, `\S`

Commonly provided shorthands for digit, word character, and space character classes. A *word character* is often all ASCII alphanumeric characters plus the underscore. However, the list of alphanumerics can include additional locale or Unicode alphanumerics, depending on the implementation. For example, `\d` matches a single digit character and is usually equivalent to `[0-9]`.

POSIX character class: `[:`*alnum*`:]`

POSIX defines several character classes that can be used only within regular expression character classes (see Table 1). For example, `[:lower:]`, when written as `[[:lower:]]`, is equivalent to `[a-z]` in the ASCII locale.

Unicode properties, scripts, and blocks: `\p{prop}`, `\P{prop}`
: The Unicode standard defines classes of characters that have a particular property, belong to a script, or exist within a block. *Properties* are characteristics such as being a letter or a number (see Table 2). *Scripts* are systems of writing, such as Hebrew, Latin, or Han. *Blocks* are ranges of characters on the Unicode character map. Some implementations require that Unicode properties be prefixed with `Is` or `In`. For example, `\p{Ll}` matches lowercase letters in any Unicode supported language, such as a or α.

Unicode combining character sequence: `\X`
: Matches a Unicode base character followed by any number of Unicode combining characters. This is a shorthand for `\P{M}\p{M}`. For example, `\X` matches è as well as the two characters `e'`.

Table 1. POSIX character classes

Class	Meaning
alnum	Letters and digits.
alpha	Letters.
blank	Space or tab only.
cntrl	Control characters.
digit	Decimal digits.
graph	Printing characters, excluding space.
lower	Lowercase letters.
print	Printing characters, including space.
punct	Printing characters, excluding letters and digits.
space	Whitespace.
upper	Uppercase letters.
xdigit	Hexadecimal digits.

Table 2. Standard Unicode properties

Property	Meaning
`\p{L}`	Letters.
`\p{Ll}`	Lowercase letters.
`\p{Lm}`	Modifier letters.
`\p{Lo}`	Letters, other. These have no case and are not considered modifiers.
`\p{Lt}`	Titlecase letters.
`\p{Lu}`	Uppercase letters.
`\p{C}`	Control codes and characters not in other categories.
`\p{Cc}`	ASCII and Latin-1 control characters.
`\p{Cf}`	Non-visible formatting characters.
`\p{Cn}`	Unassigned code points.
`\p{Co}`	Private use, such as company logos.
`\p{Cs}`	Surrogates.
`\p{M}`	Marks meant to combine with base characters, such as accent marks.
`\p{Mc}`	Modification characters that take up their own space. Examples include "vowel signs."
`\p{Me}`	Marks that enclose other characters, such as circles, squares, and diamonds.
`\p{Mn}`	Characters that modify other characters, such as accents and umlauts.
`\p{N}`	Numeric characters.
`\p{Nd}`	Decimal digits in various scripts.
`\p{Nl}`	Letters that are numbers, such as Roman numerals.
`\p{No}`	Superscripts, symbols, or non-digit characters representing numbers.
`\p{P}`	Punctuation.
`\p{Pc}`	Connecting punctuation, such as an underscore.
`\p{Pd}`	Dashes and hyphens.
`\p{Pe}`	Closing punctuation complementing `\p{Ps}`.
`\p{Pi}`	Initial punctuation, such as opening quotes.
`\p{Pf}`	Final punctuation, such as closing quotes.
`\p{Po}`	Other punctuation marks.
`\p{Ps}`	Opening punctuation, such as opening parentheses.

Table 2. Standard Unicode properties (continued)

Property	Meaning
`\p{S}`	Symbols.
`\p{Sc}`	Currency.
`\p{Sk}`	Combining characters represented as individual characters.
`\p{Sm}`	Math symbols.
`\p{So}`	Other symbols.
`\p{Z}`	Separating characters with no visual representation.
`\p{Zl}`	Line separators.
`\p{Zp}`	Paragraph separators.
`\p{Zs}`	Space characters.

Anchors and zero-width assertions

Anchors and "zero-width assertions" match positions in the input string. (See MRE 127–133.)

Start of line/string: `^`, `\A`
: Matches at the beginning of the text being searched. In multiline mode, `^` matches after any newline. Some implementations support `\A`, which only matches at the beginning of the text.

End of line/string: `$`, `\Z`, `\z`
: `$` matches at the end of a string. Some implementations also allow `$` to match before a string-ending newline. If modified by multiline mode, `$` matches before any newline as well. When supported, `\Z` matches the end of string or before a string-ending newline, regardless of match mode. Some implementations also provide `\z`, which only matches the end of the string, regardless of newlines.

Start of match: `\G`
: In iterative matching, `\G` matches the position where the previous match ended. Often, this spot is reset to the beginning of a string on a failed match.

Word boundary: `\b`, `\B`, `\<`, `\>`
: Word boundary metacharacters match a location where a word character is next to a non-word character. `\b` often specifies a word boundary location, and `\B` often specifies a not-word-boundary location. Some implementations provide separate metasequences for start- and end-of-word boundaries, often `\<` and `\>`.

Lookahead: `(?=...)`, `(?!...)`
Lookbehind: `(?<=...)`, `(?<!...)`
: *Lookaround constructs* match a location in the text where the subpattern would match (lookahead), would not match (negative lookahead), would have finished matching (lookbehind), or would not have finished matching (negative lookbehind). For example, `foo(?=bar)` matches `foo` in `foobar` but not `food`. Implementations often limit lookbehind constructs to subpatterns with a predetermined length.

Comments and mode modifiers

Mode modifiers are a way to change how the regular expression engine interprets a regular expression. (See MRE 109–112, 133–135.)

Multiline mode: `m`
: Changes the behavior of `^` and `$` to match next to newlines within the input string.

Single-line mode: `s`
: Changes the behavior of `.` (dot) to match all characters, including newlines, within the input string.

Case-insensitive mode: `i`
: Treat as identical letters that differ only in case.

Free-spacing mode: `x`
: Allows for whitespace and comments within a regular expression. The whitespace and comments (starting with `#` and extending to the end of the line) are ignored by the regular expression engine.

Mode modifiers: `(?i)`, `(?-i)`, `(?mod:…)`
: Usually, mode modifiers may be set within a regular expression with `(?mod)` to turn modes on for the rest of the current subexpression; `(?-mod)` to turn modes off for the rest of the current subexpression; and `(?mod:...)` to turn modes on or off between the colon and the closing parentheses. For example, `"use (?i:perl)"` matches "use perl", "use Perl", "use PeRl", etc.

Comments: `(?#...)` *and* `#`
: In free-spacing mode, `#` indicates that the rest of the line is a comment. When supported, the comment span `(?#...)` can be embedded anywhere in a regular expression, regardless of mode. For example, `.{0,80}(?#Field limit is 80 chars)` allows you to make notes about why you wrote `.{0,80}`.

Literal-text span: `\Q...\E`
: Escapes metacharacters between `\Q` and `\E`. For example, `\Q(.*)\E` is the same as `\(\.\*\)`.

Grouping, capturing, conditionals, and control

This section covers the syntax for grouping subpatterns, capturing submatches, conditional submatches, and quantifying the number of times a subpattern matches. (See MRE 135–140.)

Capturing and grouping parentheses: `(...)` *and* `\1`, `\2`, …
: Parentheses perform two functions: grouping and capturing. Text matched by the subpattern within parentheses is captured for later use. Capturing parentheses are numbered by counting their opening parentheses from the left. If backreferences are available, the submatch can be referred to later in the same match with `\1`, `\2`, etc. The captured text is made available after a match by implementation-specific methods. For example, `\b(\w+)\b\s+\1\b` matches duplicate words, such as `the the`.

Grouping-only parentheses: `(?:...)`
: Groups a subexpression, possibly for alternation or quantifiers, but does not capture the submatch. This is useful for efficiency and reusability. For example, `(?:foobar)` matches `foobar`, but does not save the match to a capture group.

Named capture: `(?<name>…)`
: Performs capturing and grouping, with captured text later referenced by *name*. For example, `Subject: (?<subject>.*)` captures the text following `Subject:` to a capture group that can be referenced by the name `subject`.

Atomic grouping: `(?>...)`
: Text matched within the group is never backtracked into, even if this leads to a match failure. For example, `(?>[ab]*)\w\w` matches `aabbcc` but not `aabbaa`.

Alternation: `...|...`
: Allows several subexpressions to be tested. Alternation's low precedence sometimes causes subexpressions to be longer than intended, so use parentheses to specifically group what you want alternated. For example, `\b(foo|bar)\b` matches either of the words `foo` or `bar`.

Conditional: `(?`*`if then | else`*`)`
: The *`if`* is implementation dependent, but generally is a reference to a captured subexpression or a lookaround. The *`then`* and *`else`* parts are both regular expression patterns. If the *`if`* part is true, the *`then`* is applied. Otherwise, *`else`* is applied. For example, `(<)?foo(?(1)>|bar)` matches `<foo>` and `foobar`.

Greedy quantifiers: `*`, `+`, `?`, `{`*num,num*`}`
: The greedy quantifiers determine how many times a construct may be applied. They attempt to match as many times as possible, but will backtrack and give up matches if necessary for the success of the overall match. For example, `(ab)+` matches all of `abababababab`.

Lazy quantifiers: `*?`, `+?`, `??`, `{num,num }?`

Lazy quantifiers control how many times a construct may be applied. However, unlike greedy quantifiers, they attempt to match as few times as possible. For example, `(an)+?` matches only `an` of `banana`.

Possessive Quantifiers: `*+`, `++`, `?+`, `{num,num }+`

Possessive quantifiers are like greedy quantifiers, except that they "lock in" their match, disallowing later backtracking to break up the sub-match. For example, `(ab)++ab` will not match `abababab`.

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*.