

## NAME

perlref - Perl Regular Expressions Reference

## DESCRIPTION

This is a quick reference to Perl's regular expressions. For full information see *perlre* and *perlop*, as well as the *SEE ALSO* section in this document.

## OPERATORS

`=~` determines to which variable the regex is applied. In its absence, `$_` is used.

```
$var =~ /foo/;
```

`!~` determines to which variable the regex is applied, and negates the result of the match; it returns false if the match succeeds, and true if it fails.

```
$var !~ /foo/;
```

`m/pattern/msixpogcdual` searches a string for a pattern match, applying the given options.

```
m Multiline mode - ^ and $ match internal lines
s match as a Single line - . matches \n
i case-Insensitive
x eXtended legibility - free whitespace and comments
p Preserve a copy of the matched string -
  ${^PREMATCH}, ${^MATCH}, ${^POSTMATCH} will be defined.
o compile pattern Once
g Global - all occurrences
c don't reset pos on failed matches when using /g
a restrict \d, \s, \w and [:posix:] to match ASCII only
aa (two a's) also /i matches exclude ASCII/non-ASCII
l match according to current locale
u match according to Unicode rules
d match according to native rules unless something indicates
  Unicode
```

If 'pattern' is an empty string, the last *successfully* matched regex is used. Delimiters other than '/' may be used for both this operator and the following ones. The leading `m` can be omitted if the delimiter is '/'.

`qr/pattern/msixpodual` lets you store a regex in a variable, or pass one around. Modifiers as for `m//`, and are stored within the regex.

`s/pattern/replacement/msixpogcdual` substitutes matches of 'pattern' with 'replacement'. Modifiers as for `m//`, with two additions:

```
e Evaluate 'replacement' as an expression
r Return substitution and leave the original string untouched.
```

'e' may be specified multiple times. 'replacement' is interpreted as a double quoted string unless a single-quote (') is the delimiter.

`?pattern?` is like `m/pattern/` but matches only once. No alternate delimiters can be used. Must be reset with `reset()`.

## SYNTAX

```
\      Escapes the character immediately following it
.      Matches any single character except a newline (unless /s is
```

|                          |   |  |
|--------------------------|---|--|
|                          | used)   |  |
| ^                        | Matches at the beginning of the string (or line, if /m is used) |  |
| \$                       | Matches at the end of the string (or line, if /m is used)       |  |
| *                        | Matches the preceding element 0 or more times                   |  |
| +                        | Matches the preceding element 1 or more times                   |  |
| ?                        | Matches the preceding element 0 or 1 times                      |  |
| {...}                    | Specifies a range of occurrences for the element preceding it   |  |
| [...]                    | Matches any one of the characters contained within the brackets |  |
| (...)                    | Groups subexpressions for capturing to \$1, \$2...              |  |
| (?:...)                  | Groups subexpressions without capturing (cluster)               |  |
|                          | Matches either the subexpression preceding or following it      |  |
| \g1 or \g{1}, \g2 ...    | Matches the text from the Nth group                             |  |
| \1, \2, \3 ...           | Matches the text from the Nth group                             |  |
| \g-1 or \g{-1}, \g-2 ... | Matches the text from the Nth previous group                    |  |
| \g{name}                 | Named backreference   |  |
| \k<name>                 | Named backreference   |  |
| \k'name'                 | Named backreference   |  |
| (?P=name)                | Named backreference (python syntax)                             |  |

## ESCAPE SEQUENCES

These work as in normal strings.

|            |  |
|------------|--|
| \a         | Alarm (beep)   |
| \e         | Escape   |
| \f         | Formfeed   |
| \n         | Newline  |
| \r         | Carriage return                                      |
| \t         | Tab  |
| \037       | Char whose ordinal is the 3 octal digits, max \777   |
| \o{2307}   | Char whose ordinal is the octal number, unrestricted |
| \x7f       | Char whose ordinal is the 2 hex digits, max \xFF     |
| \x{263a}   | Char whose ordinal is the hex number, unrestricted   |
| \cx        | Control-x  |
| \N{name}   | A named Unicode character or character sequence      |
| \N{U+263D} | A Unicode character by hex ordinal                   |
| \l         | Lowercase next character                             |
| \u         | Titlecase next character                             |
| \L         | Lowercase until \E                                   |
| \U         | Uppercase until \E                                   |
| \F         | Foldcase until \E                                    |
| \Q         | Disable pattern metacharacters until \E              |
| \E         | End modification                                     |

For Titlecase, see *Titlecase*.

This one works differently from normal strings:

|    |  |
|----|--|
| \b | An assertion, not backspace, except in a character class |
|----|--|

## CHARACTER CLASSES

|        |   |
|--------|---|
| [amy]  | Match 'a', 'm' or 'y'                                     |
| [f-j]  | Dash specifies "range"                                    |
| [f-j-] | Dash escaped or at start or end means 'dash'              |
| [^f-j] | Caret indicates "match any character <i>except</i> these" |

The following sequences (except `\N`) work within or without a character class. The first six are locale aware, all are Unicode aware. See *perllocale* and *perlunicode* for details.

```

\d      A digit
\D      A nondigit
\w      A word character
\W      A non-word character
\s      A whitespace character
\S      A non-whitespace character
\h      An horizontal whitespace
\H      A non horizontal whitespace
\N      A non newline (when not followed by '{NAME}'; experimental;
        not valid in a character class; equivalent to [^\n]; it's
        like '.' without /s modifier)
\v      A vertical whitespace
\V      A non vertical whitespace
\R      A generic newline          (?>\v|\x0D\x0A)

\C      Match a byte (with Unicode, '.' matches a character)
\p      Match P-named (Unicode) property
\p{...} Match Unicode property with name longer than 1 character
\PP     Match non-P
\P{...} Match lack of Unicode property with name longer than 1 char
\X      Match Unicode extended grapheme cluster
    
```

POSIX character classes and their Unicode and Perl equivalents:

| POSIX<br>[[:...:]] | ASCII-<br>range<br>\p{...} | Full-<br>range<br>\p{...} | backslash<br>sequence | Description  |
|--------------------|----------------------------|---------------------------|-----------------------|--|
| alnum              | PosixAlnum                 | XPosixAlnum               |                       | Alpha plus Digit   |
| alpha              | PosixAlpha                 | XPosixAlpha               |                       | Alphabetic characters  |
| ascii              | ASCII                      |                           |                       | Any ASCII character  |
| blank              | PosixBlank                 | XPosixBlank               | \h                    | Horizontal whitespace;<br>full-range also<br>written as<br>\p{HorizSpace} (GNU<br>extension) |
| cntrl              | PosixCntrl                 | XPosixCntrl               |                       | Control characters   |
| digit              | PosixDigit                 | XPosixDigit               | \d                    | Decimal digits   |
| graph              | PosixGraph                 | XPosixGraph               |                       | Alnum plus Punct   |
| lower              | PosixLower                 | XPosixLower               |                       | Lowercase characters   |
| print              | PosixPrint                 | XPosixPrint               |                       | Graph plus Print, but<br>not any Cntrls  |
| punct              | PosixPunct                 | XPosixPunct               |                       | Punctuation and Symbols<br>in ASCII-range; just<br>punct outside it                          |
| space              | PosixSpace<br>PerlSpace    | XPosixSpace<br>XPerlSpace | \s                    | [\s\cK]<br>Perl's whitespace def'n   |
| upper              | PosixUpper                 | XPosixUpper               |                       | Uppercase characters   |
| word               | PosixWord                  | XPosixWord                | \w                    | Alnum + Unicode marks +<br>connectors, like '_'<br>(Perl extension)                          |
| xdigit             | ASCII_Hex_Digit            | XPosixDigit               |                       | Hexadecimal digit,   |

ASCII-range is  
[0-9A-Fa-f]

Also, various synonyms like `\p{Alpha}` for `\p{XPosixAlpha}`; all listed in *"Properties accessible through `\p{}` and `\P{}`" in perluniprops*

Within a character class:

| POSIX                   | traditional     | Unicode                |
|-------------------------|-----------------|------------------------|
| <code>[:digit:]</code>  | <code>\d</code> | <code>\p{Digit}</code> |
| <code>[:^digit:]</code> | <code>\D</code> | <code>\P{Digit}</code> |

## ANCHORS

All are zero-width assertions.

- `^` Match string start (or line, if `/m` is used)
- `$` Match string end (or line, if `/m` is used) or before newline
- `\b` Match word boundary (between `\w` and `\W`)
- `\B` Match except at word boundary (between `\w` and `\w` or `\W` and `\W`)
- `\A` Match string start (regardless of `/m`)
- `\Z` Match string end (before optional newline)
- `\z` Match absolute string end
- `\G` Match where previous `m//g` left off
- `\K` Keep the stuff left of the `\K`, don't include it in `$&`

## QUANTIFIERS

Quantifiers are greedy by default and match the **longest** leftmost.

| Maximal            | Minimal             | Possessive          | Allowed range   |
|--------------------|---------------------|---------------------|---|
| -----              | -----               | -----               | -----   |
| <code>{n,m}</code> | <code>{n,m}?</code> | <code>{n,m}+</code> | Must occur at least <code>n</code> times<br>but no more than <code>m</code> times |
| <code>{n,}</code>  | <code>{n,}?</code>  | <code>{n,}+</code>  | Must occur at least <code>n</code> times  |
| <code>{n}</code>   | <code>{n}?</code>   | <code>{n}+</code>   | Must occur exactly <code>n</code> times   |
| <code>*</code>     | <code>*?</code>     | <code>*+</code>     | 0 or more times (same as <code>{0,}</code> )                                      |
| <code>+</code>     | <code>+?</code>     | <code>++</code>     | 1 or more times (same as <code>{1,}</code> )                                      |
| <code>?</code>     | <code>??</code>     | <code>?+</code>     | 0 or 1 time (same as <code>{0,1}</code> )   |

The possessive forms (new in Perl 5.10) prevent backtracking: what gets matched by a pattern with a possessive quantifier will not be backtracked into, even if that causes the whole match to fail.

There is no quantifier `{,n}`. That's interpreted as a literal string.

## EXTENDED CONSTRUCTS

|                                  |   |
|----------------------------------|---|
| <code>(?#text)</code>            | A comment   |
| <code>(?:...)</code>             | Groups subexpressions without capturing (cluster)         |
| <code>(?pimsx-imsx:...)</code>   | Enable/disable option (as per <code>m//</code> modifiers) |
| <code>(?=...)</code>             | Zero-width positive lookahead assertion                   |
| <code>(?!...)</code>             | Zero-width negative lookahead assertion                   |
| <code>(?&lt;=...)</code>         | Zero-width positive lookbehind assertion                  |
| <code>(?&lt;!=...)</code>        | Zero-width negative lookbehind assertion                  |
| <code>(?&gt;...)</code>          | Grab what we can, prohibit backtracking                   |
| <code>(? ...)</code>             | Branch reset  |
| <code>(?&lt;name&gt;...)</code>  | Named capture   |
| <code>(?'name'...)</code>        | Named capture   |
| <code>(?P&lt;name&gt;...)</code> | Named capture (python syntax)                             |

|                              |  |
|------------------------------|--|
| <code>(?{ code })</code>     | Embedded code, return value becomes <code>\$\$R</code> |
| <code>(??{ code })</code>    | Dynamic regex, return value used as regex              |
| <code>(?N)</code>            | Recurse into subpattern number N                       |
| <code>(?-N), (?+N)</code>    | Recurse into Nth previous/next subpattern              |
| <code>(?R), (?0)</code>      | Recurse at the beginning of the whole pattern          |
| <code>(?&amp;name)</code>    | Recurse into a named subpattern                        |
| <code>(?P&gt;name)</code>    | Recurse into a named subpattern (python syntax)        |
| <code>(?(cond)yes no)</code> |  |
| <code>(?(cond)yes)</code>    | Conditional expression, where "cond" can be:           |
| <code>(?=pat)</code>         | look-ahead   |
| <code>(?!pat)</code>         | negative look-ahead                                    |
| <code>(?&lt;=pat)</code>     | look-behind  |
| <code>(?&lt;!pat)</code>     | negative look-behind                                   |
| <code>(N)</code>             | subpattern N has matched something                     |
| <code>&lt;name&gt;</code>    | named subpattern has matched something                 |
| <code>'name'</code>          | named subpattern has matched something                 |
| <code>{code}</code>          | code condition   |
| <code>(R)</code>             | true if recursing                                      |
| <code>(RN)</code>            | true if recursing into Nth subpattern                  |
| <code>(R&amp;name)</code>    | true if recursing into named subpattern                |
| <code>(DEFINE)</code>        | always false, no no-pattern allowed                    |

## VARIABLES

|                             |                                       |
|-----------------------------|---------------------------------------|
| <code>\$_</code>            | Default variable for operators to use |
| <code>\$`</code>            | Everything prior to matched string    |
| <code>\$&amp;</code>        | Entire matched string                 |
| <code>\$'</code>            | Everything after to matched string    |
| <code>\$\$^PREMATCH</code>  | Everything prior to matched string    |
| <code>\$\$^MATCH</code>     | Entire matched string                 |
| <code>\$\$^POSTMATCH</code> | Everything after to matched string    |

The use of `$``, `$&` or `$'` will slow down **all** regex use within your program. Consult *perlvar* for `@-` to see equivalent expressions that won't cause slow down. See also *Devel::SawAmpersand*. Starting with Perl 5.10, you can also use the equivalent variables `$$^PREMATCH`, `$$^MATCH` and `$$^POSTMATCH`, but for them to be defined, you have to specify the `/p` (preserve) modifier on your regular expression.

|                           |   |
|---------------------------|---|
| <code>\$1, \$2 ...</code> | hold the Xth captured expr  |
| <code>\$+</code>          | Last parenthesized pattern match  |
| <code>\$\$^N</code>       | Holds the most recently closed capture  |
| <code>\$\$^R</code>       | Holds the result of the last <code>{...}</code> expr                          |
| <code>@-</code>           | Offsets of starts of groups. <code>\$\$-[0]</code> holds start of whole match |
| <code>@+</code>           | Offsets of ends of groups. <code>\$\$+[0]</code> holds end of whole match     |
| <code>%+</code>           | Named capture groups  |
| <code>%-</code>           | Named capture groups, as array refs   |

Captured groups are numbered according to their *opening* paren.

## FUNCTIONS

|                      |                                  |
|----------------------|----------------------------------|
| <code>lc</code>      | Lowercase a string               |
| <code>lcfirst</code> | Lowercase first char of a string |
| <code>uc</code>      | Uppercase a string               |

|                        |  |
|------------------------|--|
| <code>ucfirst</code>   | Titlecase first char of a string         |
| <code>fc</code>        | Foldcase a string                        |
| <code>pos</code>       | Return or set current match position     |
| <code>quotemeta</code> | Quote metacharacters                     |
| <code>reset</code>     | Reset <code>?pattern?</code> status      |
| <code>study</code>     | Analyze string for optimizing matching   |
| <code>split</code>     | Use a regex to split a string into parts |

The first five of these are like the escape sequences `\L`, `\l`, `\U`, `\u`, and `\F`. For Titlecase, see *Titlecase*; For Foldcase, see *Foldcase*.

## TERMINOLOGY

### Titlecase

Unicode concept which most often is equal to uppercase, but for certain characters like the German "sharp s" there is a difference.

### Foldcase

Unicode form that is useful when comparing strings regardless of case, as certain characters have complex one-to-many case mappings. Primarily a variant of lowercase.

## AUTHOR

Iain Truskett. Updated by the Perl 5 Porters.

This document may be distributed under the same terms as Perl itself.

## SEE ALSO

- *perlretut* for a tutorial on regular expressions.
- *perlrequick* for a rapid tutorial.
- *perlre* for more details.
- *perlvar* for details on the variables.
- *perlop* for details on the operators.
- *perlfunc* for details on the functions.
- *perlfaq6* for FAQs on regular expressions.
- *perlrebackslash* for a reference on backslash sequences.
- *perlrecharclass* for a reference on character classes.
- The *re* module to alter behaviour and aid debugging.
- "*Debugging Regular Expressions*" in *perldebug*
- *perluniintro*, *perlunicode*, *charnames* and *perllocale* for details on regexes and internationalisation.
- *Mastering Regular Expressions* by Jeffrey Friedl (<http://oreilly.com/catalog/9780596528126/>) for a thorough grounding and reference on the topic.

## THANKS

David P.C. Wollmann, Richard Soderberg, Sean M. Burke, Tom Christiansen, Jim Cromie, and Jeffrey Goff for useful advice.

