Previous Page | Next Page

Tables of Perl Regular Expression (PRX) Metacharacters

General Constructs

General Constructs

Metacharacter	Description
( )	indicates grouping.
non-metacharacter	matches a character.
{ } [ ] ( ) ^ $. | * + ? \	to match these characters, override (escape) with \.
\	overrides the next metacharacter.
\n	matches capture buffer *n*.
(?:...)	specifies a non-capturing group.

Basic Perl Metacharacters

The following table lists the metacharacters that you can use to match patterns in Perl regular expressions.

Basic Perl Metacharacters and Their Descriptions

Metacharacter	Description
\a	matches an alarm (bell) character.
\A	matches a character only at the beginning of a string.
\b	matches a word boundary (the position between a word and a space): • "er\b" matches the "er" in "never" • "er\b" does not match the "er" in "verb"
\B	matches a non-word boundary: • "er\B" matches the "er" in "verb" • "er\B" does not match the "er" in "never"
\cA-\cZ	matches a control character. For example, \cX matches the control character control-X.
\C	matches a single byte.
\d	matches a digit character that is equivalent to [0−9].
\D	matches a non-digit character that is equivalent to [^0−9].
\e	matches an escape character.
\E	specifies the end of case modification.

\f	matches a form feed character.
\l	specifies that the next character is lowercase.
\L	specifies that the next string of characters, up to the \E metacharacter, is lowercase.
\n	matches a newline character.
\num $num	matches capture buffer *num*, where *num* is a positive integer. Perl variable syntax ($num) is valid when referring to capture buffers, but not in other cases.
\Q	escapes (places a backslash before) all non-word characters.
\r	matches a return character.
\s	matches any whitespace character, including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].
\S	matches any character that is not a whitespace character and is equivalent to [^\f\n\r\t\v].
\t	matches a tab character.
\u	specifies that the next character is uppercase.
\U	specifies that the next string of characters, up to the \E metacharacter, is uppercase.
\v	vertical white space.
\w	matches any word character or alphanumeric character, including the underscore.
\W	matches any non-word character or non-alphanumeric character, and excludes the underscore.
ddd	matches the octal character *ddd*.
\x*dd*	matches the hexadecimal character *dd*.
\z	matches a character only at the end of a string.
\Z	matches a character only at the end of a string or before newline at the end of a string.

Metacharacters and Replacement Strings

You can use the following metacharacters in both a regular expression and in replacement text, when you use a substitution regular expression:

- \l
- \u
- \L
- \E
- \U
- \Q

These metacharacters are useful in replacement text for controlling the case of capture buffers that are used within replacement text. For an example of how these metacharacters can be used, see Replacing Text

For a description of these metacharacters, see Basic Perl Metacharacters and Their Descriptions.

Other Quantifiers

The following table lists other qualifiers that you can use in Perl regular expressions. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Other Quantifiers

Metacharacter	Description
\	marks the next character as either a special character, a literal, a back reference, or an octal escape: • “\n” matches a newline character • “\\” matches “\” • “\(“ matches”(“
|	specifies the *or* condition when you compare alphanumeric strings. For example, the construct x|y matches either x or y: • "z|food" matches either "z" or "food" • “(z|f)ood” matches “zood” or “food”
^	matches the position at the beginning of the input string.
$	matches the position at the end of the input string.
period (.)	matches any single character except newline. To match any character including newline, use a pattern such as "[.\n]".
(pattern)	specifies grouping. Matches a pattern and creates a capture buffer for the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To retrieve the value of the capture buffer, use the PRXPOSN function. To match parentheses characters, use "\(" or "\)".

Greedy and Lazy Repetition Factors

Perl regular expressions support “greedy” repetition factors and “lazy” repetition factors. A repetition factor is considered greedy when the repetition factor matches a string as many times as it can when using a specific starting location. A repetition factor is considered lazy when it matches a string the minimum number of times that is needed to satisfy the match. To designate a repetition factor as lazy, add a ? to the end of the repetition factor. By default, repetition factors are considered greedy.

The following table lists the greedy repetition factors. The descriptions of the repetition factors in the table include examples of how they can be used.

Greedy Repetition Factors

Metacharacter	Description
*	matches the preceding subexpression zero or more times: • zo* matches "z" and "zoo" • * is equivalent to {0,}
+	matches the preceding subexpression one or more times: • "zo+" matches "zo" and "zoo" • "zo+" does not match "z" • + is equivalent to {1,}
?	matches the preceding subexpression zero or one time: • "do(es)?" matches the "do" in "do" or "does" • ? is equivalent to {0,1}
{n}	matches at least *n* times.
{n,}	matches a pattern at least *n* times.
{n,m}	m and n are nonnegative integers, where n<=m. They match at least *n* and at most *m* times: • "o{1,3}" matches the first three o's in "fooooood" • "o{0,1}" is equivalent to "o?" You cannot put a space between the comma and the numbers.

The following table lists the lazy repetition metacharacters.

Lazy Repetition Factors

Metacharacter	Description
*?	matches a pattern zero or more times.
+?	matches a pattern one or more times.
??	matches a pattern zero or one time.
{n}?	matches exactly *n* times.
{n,}?	matches a pattern at least *n* times.
{n,m}?	matches a pattern at least *n* times but not more than *m* times.

Class Groupings

The following table lists character class groupings. You specify these classes by enclosing characters inside brackets. These metacharacters share a set of common properties. To be successful, the character class must always match a character. The negated character class must always match a character that is not in the list of characters that are designated inside the brackets. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Character Class Groupings

Metacharacter	Description
[...]	specifies a character set that matches any one of the enclosed characters: • “[abc]” matches the “a” in “plain”
[^...]	specifies a negative character set that matches any character that is not enclosed: • “[^abc]” matches the “p” in “plain”
[a-z]	specifies a range of characters that matches any character in the range: • “[a-z]” matches any lowercase alphabetic character in the range “a” through “z”
[^a-z]	specifies a range of characters that does not match any character in the range: • "[^a-z]" matches any character that is not in the range "a" through "z"
[[:alpha:]]	matches an alphabetic character.
[[:^alpha:]]	matches a nonalphabetic character.
[[:alnum:]]	matches an alphanumeric character.
[[:^alnum:]]	matches a non-alphanumeric character.

[[:^alnum:]]	matches a non-alphanumeric character.
[[:ascii:]]	matches an ASCII character. Equivalent to [\0–\177].
[[:^ascii:]]	matches a non-ASCII character. Equivalent to [^\0–\177].
[[:blank:]]	matches a blank character.
[[:^blank:]]	matches a non-blank character.
[[:ctrl:]]	matches a control character.
[[:^ctrl:]]	matches a character that is not a control character.
[[:digit:]]	matches a digit. Equivalent to \d.
[[:^digit:]]	matches a non-digit character. Equivalent to \D.
[[:graph:]]	is a visible character, excluding the space character. Equivalent to [[:alnum:][:punct:]].
[[:^graph:]]	is not a visible character. Equivalent to [^[:alnum:][:punct:]].
[[:lower:]]	matches lowercase characters.
[[:^lower:]]	does not match lowercase characters.
[[:print:]]	prints a string of characters.
[[:^print:]]	does not print a string of characters.
[[:punct:]]	matches a punctuation character or a visible character that is not a space or alphanumeric.
[[:^punct:]]	does not match a punctuation character or a visible character that is not a space or alphanumeric.
[[:space:]]	matches a space. Equivalent to \s.
[[:^space:]]	does not match a space. Equivalent to \S.
[[:upper:]]	matches uppercase characters.
[[:^upper:]]	does not match uppercase characters.
[[:word:]]	matches a word. Equivalent to \w.
[[:^word:]]	does not match a word. Equivalent to \W.
[[:xdigit:]]	matches a hexadecimal character.
[[:^xdigit:]]	does not match a hexadecimal character.

Look-Ahead and Look-Behind Behavior

Look-ahead and look-behind are ways to look ahead or behind a match to see whether a particular text occurs. The text that is found with look-ahead or look-behind is not included in the match that is found. For example, if you want to find names that end with “Jr.”, but you do not want “Jr.” to be part of the match, you could use the regular expression /.*(?=Jr\.)/. For the value "John Wainright Jr.", the regular expression finds "John Wainright" as a match because it is followed by "Jr."

Look-Ahead and Look-Behind Behavior

Metacharacter	Description
(?=...)	specifies a zero-width, positive, look-ahead assertion. For example, in the expression *regex1 (?=regex2)*, a match is found if both *regex1* and *regex2* match. *regex2* is not included in the final match.
(?!...)	specifies a zero-width, negative, look-ahead assertion. For example, in the expression *regex1 (?!regex2)*, a match is found if *regex1* matches and *regex2* does not match. *regex2* is not included in the final match.
(?<=...)	specifies a zero-width, positive, look-behind assertion. For example, in the expression *(?<=regex1) regex2*, a match is found if both *regex1* and *regex2* match. *regex1* is not included in the final match. Works with fixed-width look-behind only.
(?<!...)	specifies a zero-width, negative, look-behind assertion. Works with fixed-width look-behind only.

Comments and Inline Modifiers

The metacharacters in this table contain a question mark as the first element inside the parentheses. The characters after the question mark indicate the extension.

Comments and Inline Modifiers

Metacharacter	Description
(?#text)	specifies a comment in which the text is ignored.
(?imsx)	specifies one or more embedded pattern-matching modifiers. If the pattern is case insensitive, you can use (?i) at the front of the pattern. An example is `$pattern="(?i)foobar";`. Letters that appear after a hyphen (-) turn the modifiers off.

Selecting the Best Condition By Using Combining Operators

The elementary regular expressions (for example, \a and \w) that are described in the preceding tables can match at most one substring at the given position in the input string. However, operators that perform combining in typical regular expressions combine elementary metacharacters to create more complex patterns. In an ambiguous situation, these operators can determine the best match or the worst match. The match that is the best is always chosen.

Best Match Using Combining Operators

Best Match Using Combining Operators

Metacharacter	Description
ST	in the following example, specifies that AB and A'B', and A and A' are substrings that can be matched by S, and that B and B' are substrings that can be matched by T: • If A is a better match for S than A', then AB is a better match than A'B'. • If A and A' coincide, then AB is a better match than AB' if B is a better match for T than B'.
S|T	specifies that when S can match, it is a better match than when only T can match. The ordering of two matches for S is the same as for S. Similarly, the ordering of two matches for T is the same as for T.
S{repeat-count}	matches as SSS . . . S (repeated as many times as necessary).
S{min,max}	matches as S{max}|S{max-1}| . . . |S{min+1}|S{min}.
S{min,max}?	matches as S{min}|S{min+1}| . . . |S{max-1}|S{max}.
S?, S*, S+	same as S{0,1}, S{0, big-number}, S{1,big-number}, respectively.
S??, S*?, S+	same as S{0,1}?, S{0, big-number}?, S{1,big-number}?, respectively.
(?=S), (?<=S)	considers the best match for S. (This is important only if S has capturing parentheses, and back references are used elsewhere in the whole regular expression.)
(?!S), (?<!S)	unnecessary to describe the ordering for this grouping operator because only whether S can match is important.

Previous Page | Next Page | Top of Page