[next](http://www.cs.cf.ac.uk/Dave/PERL/node80.html)[up](http://www.cs.cf.ac.uk/Dave/PERL/node78.html)[previous](http://www.cs.cf.ac.uk/Dave/PERL/node78.html)[contents](http://www.cs.cf.ac.uk/Dave/PERL/node1.html)  
**Next:** [Backtracking](http://www.cs.cf.ac.uk/Dave/PERL/node80.html) **Up:** [Using Regular Expressions](http://www.cs.cf.ac.uk/Dave/PERL/node78.html) **Previous:** [Using Regular Expressions](http://www.cs.cf.ac.uk/Dave/PERL/node78.html)

**Special pattern matching character operators**

In particular the following metacharacters have their standard egrep-ish meanings:

\ Quote the next metacharacter

^ Match the beginning of the line

. Match any character (except newline)

$ Match the end of the line (or before newline at the end)

| Alternation

() Grouping

[] Character class

The simplest and very common pattern matching character operators is the .

This simply allows for any single character to match where a . is placed in a regular expression.

For example /b.t/ can match to bat, bit, but or anything like bbt, bct ....

Square brackets ([..]) allow for any one of the letters listed inside the brackets to be matched at the specified position.

For example /b[aiu]t/ can only match to bat, bit or but.

You can specify a range inside[..]. For example (regex.pl):

[012345679] # any single digit

[0-9] # also any single digit

[a-z] # any single lower case letter

[a-zA-Z] # any single letter

[0-9\-] # 0-9 plus minus character

The *caret* (^) can be used to negate matches

For example (regex.pl):

[^0-9] # any single non-digit

[^aeiouAEIOU] # any single non-vowel

The control characters \d (digit), \s (space), \w (word character) can also be used. \D, \S, \W are the negations of \d\s\w (More on This Soon)

By default, the ^ character is guaranteed to match at only the beginning of the string, the $ character at only the end (or before the newline at the end) and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by ^ or $. You may, however, wish to treat a string as a multi-line buffer, such that the ^ will match after any newline within the string, and $ will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting $\*, but this practice is now deprecated.)

To facilitate multi-line substitutions, the . character never matches a newline unless you use the /s modifier, which in effect tells Perl to pretend the string is a single line-even if it isn't. The /s modifier also overrides the setting of $\*, in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

\* 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

{n,m} Match at least n but not more than m times

(If a curly bracket occurs in any other context, it is treated as a regular character.) The \* modifier is equivalent to {0,}, the + modifier to {1,}, and the ? modifier to {0,1}. n and m are limited to integral values less than 65536.

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

\*? Match 0 or more times

+? Match 1 or more times

?? Match 0 or 1 time

{n}? Match exactly n times

{n,}? Match at least n times

{n,m}? Match at least n but not more than m times

**Some Simple Examples**

fa\*t matches to ft, fat, faat, faaat *etc*

(.\*) can be used a *wild card* match for any number (zero or more) of any characters.

Thus f.\*k matches to fk, fak, fork, flunk, *etc.*

fa+t matches to fat, faat, faaat *etc*

.+ can be used to match to one or more of any character *i.e.* at least something must be there.

Thus f.+k matches to fak, fork, flunk, *etc.* **but not** fk.

? matches to zero or one character.

Thus ba?t matches to bt or bat.

b.?t matches to bt, bat, bbt, *etc.* **but not** bunt or higher than four-letter words.

ba{3}t} only matches to baaat.

ba{1,4} matches to bat, baat, baaat and baaaat

Because patterns are processed as double quoted strings, the following also work:

\t tab (HT, TAB)

\n newline (LF, NL)

\r return (CR)

\f form feed (FF)

\a alarm (bell) (BEL)

\e escape (think troff) (ESC)

\033 octal char (think of a PDP-11)

\x1B hex char

\c[ control char

\l lowercase next char (think vi)

\u uppercase next char (think vi)

\L lowercase till \E (think vi)

\U uppercase till \E (think vi)

\E end case modification (think vi)

\Q quote regular expression metacharacters till \E

If use locale is in effect, the case map used by \l, \L, \u and <\U> is taken from the current locale. See in the perllocale manpage.

The control characters \d (digit), \s (space), \w (word character) can also be used. \D, \S, \W are the negations of \d\s\w

Note that \w matches a single alphanumeric character, not a whole word. To match a word you'd need to say \w+. If use locale is in effect, the list of alphabetic characters generated by \w is taken from the current locale. See in the perllocale manpage. You may use \w, \W, \s, \S, \d, and \D within character classes (though not as either end of a range).

Perl defines the following zero-width assertions:

\b Match a word boundary

\B Match a non-(word boundary)

\A Match at only beginning of string

\Z Match at only end of string (or before newline at the end)

\G Match only where previous m//g left off (works only with /g)

A word boundary (\b) is defined as a spot between two characters that has a \w on one side of it and a \W on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a \W. (Within character classes \b represents backspace rather than a word boundary.) The \A and \Zare just like ^ and $ except that they won't match multiple times when the /m modifier is used, while ^ and $ will match at every internal line boundary. To match the actual end of the string, not ignoring newline, you can use \Z(?!\n). The \G assertion can be used to chain global matches (using m//g) -- *see later*.

**Parenthesis as Memory**

Parenthesis can be used to delimit special matches (enforce precedence)

For example:

(abc)\*

matches " ",abc, abcabc, abcabcabc,.....

and

(a|b)(c|d)

matches ac, ad,bc,bd

* The brackets () can be used to remember and

When the bracketing construct ( ... ) is used, \<digit>< code=""> matches the digit'th substring. Outside of the pattern, always use $ instead of \ in front of the digit. (While the \<digit>< code=""> notation can on rare occasion work outside the current pattern, this should not be relied upon.) </digit><></digit><>

So :

dave(.)marshall\1

will match something like

daveXmarshallX

**BUT NOT**

daveXmarshallY

You can have more than one memory:

For Example:

a(.)b(.)c\2d\1

would match axbycydx for example.

Multiple chars (inc. 0) can be remembered:

a(.\*)b\1c

matches to abc, aFREDbFREDc

**BUT NOT**

aXXbXXXc, for example.

**Read Only Variables**

After a successful match the variable $1, $2, $3, ... are set on the same values as \1,\2,\3, ....

The scope of $<digit> (and $`, $&, and $') extends to the end of the enclosing BLOCK or eval string, or to the next successful pattern match, whichever comes first. If you want to use parentheses to delimit a subpattern (e.g., a set of alternatives) without saving it as a subpattern, follow the ( with a ?:.

So you can use later in code.

$\_ = "One Two Three Four Once ....";

/(\w+)\W+(\w+)/; # match first two words

print "1st Word is " . $1" . "\n";

print "2nd Word is " . $2" . "\n";

You also rearrange the read-only variables.

Example:

s/^([^ ]\*) \*([^ ]\*)/$2 $1/; # swap first two words

if (/Time: (..):(..):(..)/) {

$hours = $1;

$minutes = $2;

$seconds = $3;

}

Once perl sees that you need one of $&, $` or $' anywhere in the program, it has to provide them on each and every pattern match. This can slow your program down. The same mechanism that handles these provides for the use of $1, $2, etc., so you pay the same price for each regular expression that contains capturing parentheses. But if you never use $&, etc., in your script, then regular expressions without capturing parentheses won't be penalized. So avoid $&, $', and $` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price.

You will note that all backslashed metacharacters in Perl are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This makes it simple to quote a string that you want to use for a pattern but that you are afraid might contain metacharacters. Quote simply all the non-alphanumeric characters:

$pattern =~ s/(\W)/\\$1/g;

You can also use the builtin quotemeta() function to do this. An even easier way to quote metacharacters right in the match operator is to say

/$unquoted\Q$quoted\E$unquoted/

Perl defines a consistent extension syntax for regular expressions. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses (this was a syntax error in older versions of Perl). The character after the question mark gives the function of the extension. Several extensions are already supported:

(?#text)

-- A comment. The text is ignored. If the /x switch is used to enable whitespace formatting, a simple # will suffice.

(?:regular\_expression)

This groups things like () but doesn't make backreferences like () does. So

split(/\b(?:a|b|c)\b/)

is like

split(/\b(a|b|c)\b/)

but doesn't spit out extra fields.

(?=regular\_expression)

A zero-width positive lookahead assertion. For example, /\w+(?=\t)/ matches a word followed by a tab, without including the tab in $&.

(?!regular\_expression)

A zero-width negative lookahead assertion. For example foo(?!bar)/ matches any occurrence of "foo" that isn't followed by "bar". Note however that lookahead and lookbehind are NOT the same thing. You cannot use this for lookbehind: /(?!foo)bar/ will not find an occurrence of "bar" that is preceded by something which is not "foo". That's because the (?!foo) is just saying that the next thing cannot be "foo" -- and it's not, it's a "bar", so "foobar" will match. You would have to do something like /(?!foo)...bar/ for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: (?:(?!foo)...|^..?)bar/. Sometimes it's still easier just to say:

if (/foo/ && $` =~ /bar$/)

(?imsx)

One or more embedded pattern-match modifiers. This is particularly useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. The case insensitive ones need to include merely (?i) at the front of the pattern. For example:

$pattern = "foobar";

if ( /$pattern/i )

# more flexible:

$pattern = "(?i)foobar";

if ( /$pattern/ )

The specific choice of question mark for this and the new minimal matching construct was because 1) question mark is pretty rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

[next](http://www.cs.cf.ac.uk/Dave/PERL/node80.html)[upprevious](http://www.cs.cf.ac.uk/Dave/PERL/node78.html)[contents](http://www.cs.cf.ac.uk/Dave/PERL/node1.html)  
**Next:** [Backtracking](http://www.cs.cf.ac.uk/Dave/PERL/node80.html) **Up:** [Using Regular Expressions](http://www.cs.cf.ac.uk/Dave/PERL/node78.html) **Previous:** [Using Regular Expressions](http://www.cs.cf.ac.uk/Dave/PERL/node78.html)

*dave@cs.cf.ac.uk*