perlrequick (source, CPAN)

# CONTENTS



# NAME

perlrequick - Perl regular expressions quick start

# DESCRIPTION

This page covers the very basics of understanding, creating and using regular expressions ('regexes') in Perl.

# The Guide

This page assumes you already know things, like what a "pattern" is, and the basic syntax of using them. If you don't, see perlretut.

## Simple word matching

The simplest regex is simply a word, or more generally, a string of characters. A regex consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/;  # matches
```

In this statement, `World` is a regex and the `//` enclosing `/World/` tells Perl to search a string for a match. The operator `=~` associates the string with the regex match and produces a true value if the regex matched, or false if the regex did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. This idea has several variations.

Expressions like this are useful in conditionals:

```
print "It matches\n" if "Hello World" =~ /World/;
```

The sense of the match can be reversed by using `!~` operator:

```
print "It doesn't match\n" if "Hello World" !~ /World/;
```

The literal string in the regex can be replaced by a variable:

```
$greeting = "World";
print "It matches\n" if "Hello World" =~ /$greeting/;
```

If you're matching against `$_`, the `$_ =~` part can be omitted:

```
$_ = "Hello World";
print "It matches\n" if /World/;
```

Finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```
"Hello World" =~ m!World!;   # matches, delimited by '!'
"Hello World" =~ m{World};   # matches, note the matching '{}'
"/usr/bin/perl" =~ m"/perl"; # matches after '/usr/bin',
                             # '/' becomes an ordinary char
```

Regexes must match a part of the string *exactly* in order for the statement to be true:

```
"Hello World" =~ /world/;  # doesn't match, case sensitive
"Hello World" =~ /o W/;    # matches, ' ' is an ordinary char
"Hello World" =~ /World /; # doesn't match, no ' ' at end
```

Perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;       # matches 'o' in 'Hello'
"That hat is red" =~ /hat/; # matches 'hat' in 'That'
```

Not all characters can be used 'as is' in a match. Some characters, called **metacharacters**, are considered special, and reserved for use in regex notation. The metacharacters are

```
{}[]()^$.|*+?\
```

A metacharacter can be matched literally by putting a backslash before it:

```
"2+2=4" =~ /2+2/;    # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;   # matches, \+ is treated like an ordinary +
'C:\WIN32' =~ /C:\\WIN/;                    # matches
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;  # matches
```

In the last regex, the forward slash `'/'` is also backslashed, because it is used to delimit the regex.

Most of the metacharacters aren't always special, and other characters (such as the ones delimitting the pattern) become special under various circumstances. This can be confusing and lead to unexpected results. `use re 'strict'` can notify you of potential pitfalls.

Non-printable ASCII characters are represented by **escape sequences**. Common examples are `\t` for a tab, `\n` for a newline, and `\r` for a carriage return. Arbitrary bytes are represented by octal escape sequences, e.g., `\033`, or hexadecimal escape sequences, e.g., `\x1B`:

```
"1000\t2000" =~ m(0\t2)  # matches
"cat" =~ /\143\x61\x74/  # matches in ASCII, but
                         # a weird way to spell cat
```

Regexes are treated mostly as double-quoted strings, so variable substitution works:

```
$foo = 'house';
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

With all of the regexes above, if the regex matched anywhere in the string, it was considered a match. To specify *where* it should match, we would use the **anchor** metacharacters `^` and `$`. The anchor `^` means match at the beginning of the string and the anchor `$` means match at the end of the string, or before a newline at the end of the string. Some examples:

```
"housekeeper" =~ /keeper/;         # matches
"housekeeper" =~ /^keeper/;        # doesn't match
"housekeeper" =~ /keeper$/;        # matches
"housekeeper\n" =~ /keeper$/;      # matches
"housekeeper" =~ /^housekeeper$/;  # matches
```

## Using character classes

A **character class** allows a set of possible characters, rather than just a single character, to match at a particular point in a regex. There are a number of different types of character classes, but usually when people use this term, they are referring to the type described in this section, which are technically called "Bracketed character classes", because they are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. But we'll drop the "bracketed" below to correspond with common usage. Here are some examples of (bracketed) character classes:

```
/cat/;            # matches 'cat'
/[bcr]at/;        # matches 'bat', 'cat', or 'rat'
"abc" =~ /[cab]/; # matches 'a'
```

In the last statement, even though `'c'` is the first character in the class, the earliest point at which the regex can match is `'a'`.

```
/[yY][eE][sS]/; # match 'yes' in a case-insensitive way
                # 'yes', 'Yes', 'YES', etc.
/yes/i;         # also match 'yes' in a case-insensitive way
```

The last example shows a match with an `'i'` **modifier**, which makes the match case-insensitive.

Character classes also have ordinary and special characters, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are `-]\^$` and are matched using an escape:

```
/[\]c]def/; # matches ']def' or 'cdef'
$x = 'bcr';
/[$x]at/;   # matches 'bat, 'cat', or 'rat'
/[\$x]at/;  # matches '$at' or 'xat'
/[\\$x]at/; # matches '\at', 'bat, 'cat', or 'rat'
```

The special character `'-'` acts as a range operator within character classes, so that the unwieldy `[0123456789]` and `[abc...xyz]` become the svelte `[0-9]` and `[a-z]`:

```
/item[0-9]/;  # matches 'item0' or ... or 'item9'
/[0-9a-fA-F]/;  # matches a hexadecimal digit
```

If `'-'` is the first or last character in a character class, it is treated as an ordinary character.

The special character `^` in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both `[...]` and `[^...]` must match a character, or the match fails. Then

```
/[^a]at/;  # doesn't match 'aat' or 'at', but matches
           # all other 'bat', 'cat, '0at', '%at', etc.
/[^0-9]/;  # matches a non-numeric character
/[a^]at/;  # matches 'aat' or '^at'; here '^' is ordinary
```

Perl has several abbreviations for common character classes. (These definitions are those that Perl uses in ASCII-safe mode with the `/a` modifier. Otherwise they could match many more non-ASCII Unicode characters as well. See "Backslash sequences" in perlrecharclass for details.)

- \d is a digit and represents

  ```
  [0-9]
  ```

- \s is a whitespace character and represents

  ```
  [\ \t\r\n\f]
  ```

- \w is a word character (alphanumeric or _) and represents

```
[0-9a-zA-Z_]
```

- \D is a negated \d; it represents any character but a digit

```
[^0-9]
```

- \S is a negated \s; it represents any non-whitespace character

```
[^\s]
```

- \W is a negated \w; it represents any non-word character

```
[^\w]
```

- The period '.' matches any character but "\n"

The `\d\s\w\D\S\W` abbreviations can be used both inside and outside of character classes. Here are some in use:

```
/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;         # matches any digit or whitespace character
/\w\W\w/;         # matches a word char, followed by a
                  # non-word char, followed by a word char
/..rt/;           # matches any two chars, followed by 'rt'
/end\./;          # matches 'end.'
/end[.]/;         # same thing, matches 'end.'
```

The **word anchor** `\b` matches a boundary between a word character and a non-word character `\w\W` or `\W\w`:

```
$x = "Housecat catenates house and cat";
$x =~ /\bcat/;  # matches cat in 'catenates'
$x =~ /cat\b/;  # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

In the last example, the end of the string is considered a word boundary.

For natural language processing (so that, for example, apostrophes are included in words), use instead `\b{wb}`

```
"don't" =~ / .+? \b{wb} /x;  # matches the whole string
```

## Matching this or that

We can match different character strings with the **alternation** metacharacter `'|'`. To match `dog` or `cat`, we form the regex `dog|cat`. As before, Perl will try to match the regex at the earliest possible point in the string. At each character position, Perl will first try to match the first alternative, `dog`. If `dog` doesn't match, Perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and Perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/;  # matches "cat"
"cats and dogs" =~ /dog|cat|bird/;  # matches "cat"
```

Even though `dog` is the first alternative in the second regex, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/; # matches "cats"
```

At a given character position, the first alternative that allows the regex match to succeed will be the one that matches. Here, all the alternatives match at the first string position, so the first matches.

## Grouping things and hierarchical matching

The **grouping** metacharacters `()` allow a part of a regex to be treated as a single unit. Parts of a regex are grouped by enclosing them in parentheses. The regex `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more examples are

```
/(a|b)b/;    # matches 'ab' or 'bb'
/(^a|b)c/;   # matches 'ac' at start of string or 'bc' anywhere

/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/;  # matches either 'housecats' or 'housecat' or
                    # 'house'.  Note groups can be nested.

"20" =~ /(19|20|)\d\d/;  # matches the null alternative '()\d\d',
                         # because '20\d\d' can't match
```

## Extracting matches

The grouping metacharacters `()` also allow the extraction of the parts of a string that matched. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
# extract hours, minutes, seconds
$time =~ /(\d\d):(\d\d):(\d\d)/;  # match hh:mm:ss format
$hours = $1;
$minutes = $2;
$seconds = $3;
```

In list context, a match `/regex/` with groupings will return the list of matched values `($1,$2,...)`. So we could rewrite it as

```
($hours, $minutes, $second) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regex are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regex and the matching variables indicated below it:

```
/(ab(cd|ef)((gi)|j))/;
 1  2      34
```

Associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\g1`, `\g2`, ... Backreferences are matching variables that can be used *inside* a regex:

```
/(\w\w\w)\s\g1/; # find sequences like 'the the' in string
```

`$1`, `$2`, ... should only be used outside of a regex, and `\g1`, `\g2`, ... only inside a regex.

## Matching repetitions

The **quantifier** metacharacters `?`, `*`, `+`, and `{}` allow us to determine the number of repeats of a portion of a regex we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- `a?` = match 'a' 1 or 0 times
- `a*` = match 'a' 0 or more times, i.e., any number of times
- `a+` = match 'a' 1 or more times, i.e., at least once
- `a{n,m}` = match at least `n` times, but not more than `m` times.
- `a{n,}` = match at least `n` or more times
- `a{n}` = match exactly `n` times

Here are some examples:

```
/[a-z]+\s+\d*/;  # match a lowercase word, at least some space, and
                 # any number of digits
/(\w+)\s+\g1/;    # match doubled words of arbitrary length
$year =~ /^\d{2,4}$/;  # make sure year is at least 2 but not more
                       # than 4 digits
$year =~ /^\d{4}$|^\d{2}$/; # better match; throw out 3 digit dates
```

These quantifiers will try to match as much of the string as possible, while still allowing the regex to match. So we have

```
$x = 'the cat in the hat';
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

The first quantifier `.*` grabs as much of the string as possible while still having the regex match. The second quantifier `.*` has no string left to it, so it matches 0 times.

## More matching

There are a few more things you might want to know about matching operators. The global modifier `/g` allows the matching operator to match within a string as many times as possible. In scalar context, successive matches against a string will have `/g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function. For example,

```
$x = "cat dog house"; # 3 words
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the `/c`, as in `/regex/gc`.

In list context, `/g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regex. So

```
@words = ($x =~ /(\w+)/g);  # matches,
                            # $word[0] = 'cat'
                            # $word[1] = 'dog'
                            # $word[2] = 'house'
```

## Search and replace

Search and replace is performed using `s/regex/replacement/modifiers`. The `replacement` is a Perl double-quoted string that replaces in the string whatever is matched with the `regex`. The operator `=~` is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made; otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;   # $x contains "Time to feed the hacker!"
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;  # strip single quotes,
                       # $y contains "quoted words"
```

With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression. With the global modifier, `s///g` will search and replace all occurrences of the regex in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;  # $x contains "I batted four for four"
```

The non-destructive modifier `s///r` causes the result of the substitution to be returned instead of modifying `$_` (or whatever variable the substitute was bound to with `=~`):

```
$x = "I like dogs.";
$y = $x =~ s/dogs/cats/r;
print "$x $y\n"; # prints "I like dogs. I like cats."

$x = "Cats are great.";
print $x =~ s/Cats/Dogs/r =~ s/Dogs/Frogs/r =~
    s/Frogs/Hedgehogs/r, "\n";
# prints "Hedgehogs are great."

@foo = map { s/[a-z]/X/r } qw(a b c 1 2 3);
# @foo is now qw(X X X 1 2 3)
```

The evaluation modifier `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. Some examples:

```
# reverse all the words in a string
$x = "the cat in the hat";
$x =~ s/(\w+)/reverse $1/ge;   # $x contains "eht tac ni eht tah"

# convert percentage to decimal
$x = "A 39% hit rate";
$x =~ s!(\d+)%!$1/100!e;       # $x contains "A 0.39 hit rate"
```

The last example shows that `s///` can use other delimiters, such as `s!!!` and `s{}{}`, and even `s{}//`. If single quotes are used `s'''`, then the regex and replacement are treated as single-quoted strings.

## The split operator

`split /regex/, string` splits `string` into a list of substrings and returns that list. The regex determines the character sequence that `string` is split with respect to. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@word = split /\s+/, $x;  # $word[0] = 'Calvin'
                          # $word[1] = 'and'
                          # $word[2] = 'Hobbes'
```

To extract a comma-delimited list of numbers, use

```
$x = "1.618,2.718,   3.142";
@const = split /,\s*/, $x;  # $const[0] = '1.618'
                            # $const[1] = '2.718'
                            # $const[2] = '3.142'
```

If the empty regex `//` is used, the string is split into individual characters. If the regex has groupings, then the list produced contains the matched substrings from the groupings as well:

```
$x = "/usr/bin";
@parts = split m!(/)!, $x;  # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
```

Since the first character of $x matched the regex, `split` prepended an empty initial element to the list.

## `use re 'strict'`

New in v5.22, this applies stricter rules than otherwise when compiling regular expression patterns. It can find things that, while legal, may not be what you intended.

See 'strict' in re.