Regex Cheatsheet

By Dejan Panovski • Updated on Aug 28, 2026 •

Regex syntax and examples for Linux tools, including metacharacters, quantifiers, character classes, anchors, groups, BRE, ERE, and PCRE

A regular expression describes a text pattern instead of a fixed string. This cheatsheet shows how common patterns change between grep, sed, awk, Vim, Bash, and PCRE-based tools.

Metacharacters

Characters that mean something other than themselves.

`.`	Any single character except a newline
`*`	Zero or more of the item before it
`[...]`	Any one character from the set
`[^...]`	Any one character not in the set
`^`	Start of the line
`$`	End of the line
`\`	Escape a character or begin a special sequence, depending on the regex flavor

Most other characters match themselves. Line-based tools such as grep, sed, and awk read one line at a time, so `.` never matches the line ending.

Anchors and Word Boundaries

Tie a pattern to a position instead of a character.

`^error`	Line starts with `error`
`done$`	Line ends with `done`
`^exact$`	Match the whole line
`^$`	Match an empty line
`\<word`	Start of a word (GNU tools and Vim)
`word\>`	End of a word
`\bword\b`	Word boundary on both sides (GNU and PCRE)
`\B`	Any position that is not a word boundary

Anchors match a position of zero width, so they never consume a character. In awk, `\b` means a backspace rather than a word boundary; gawk provides `\y` instead.

Bracket Expressions

Match one character out of a set you define.

`[abc]`	One `a`, `b`, or `c`
`[^abc]`	Any character except `a`, `b`, or `c`
`[a-z]`	One lowercase letter
`[0-9a-fA-F]`	One hexadecimal digit
`[]abc]`	Include a literal `]` by putting it first
`[abc-]`	Include a literal `-` by putting it last
`[a^]`	`^` is literal when it is not first

Most metacharacters lose their meaning inside brackets, so `[.*]` matches a dot or an asterisk. Ranges follow the current locale, so use `LC_ALL=C` or a POSIX class when you need predictable results.

POSIX Character Classes

Portable, locale-aware sets that go inside a bracket expression.

`[[:digit:]]`	Digits
`[[:alpha:]]`	Letters
`[[:alnum:]]`	Letters and digits
`[[:lower:]] [[:upper:]]`	Lowercase and uppercase letters
`[[:space:]]`	Whitespace, including tabs and newlines
`[[:blank:]]`	Spaces and tabs only
`[[:punct:]]`	Punctuation characters
`[[:xdigit:]]`	Hexadecimal digits
`[[:print:]] [[:graph:]]`	Printable, and printable except space
`[[:cntrl:]]`	Control characters

The double brackets are not a typo. The class itself is `[:digit:]`, and the outer brackets are the bracket expression that holds it, so `[[:digit:]_]` matches a digit or an underscore.

Quantifiers

Say how many times the preceding item repeats.

`a*`	Zero or more `a`
`a\+ / a+`	One or more, BRE and ERE forms
`a\? / a?`	Zero or one
`a\{3\} / a{3}`	Exactly three
`a{3,}`	Three or more
`a{,3}`	Up to three (GNU extension for `{0,3}`)
`a{2,4}`	Between two and four
`.*`	Any run of characters, including none

Quantifiers are greedy and take the longest match available. In a basic regular expression, a `*` at the very start of the pattern is a literal asterisk because there is nothing for it to repeat.

Groups, Alternation, and Backreferences

Treat several characters as one unit and reuse what they matched.

`(ab)+`	ERE group repeated one or more times
`\(ab\)\+`	The same group in BRE
`cat|dog`	ERE alternation, either side matches
`cat|dog`	The same alternation in GNU BRE
`^(a|b)c$`	Alternation limited to the group
`\1`	Whatever group 1 matched
`\(.\)\1`	Any character repeated twice

Groups are numbered from left to right by their opening parenthesis. Backreferences are part of POSIX BRE, and GNU grep and GNU sed also accept them in extended patterns. POSIX ERE and awk do not support backreferences. In a `sed` replacement, `\1` inserts group 1 and `&` inserts the whole match.

Escaping Special Characters

Match a metacharacter as an ordinary character.

`\.`	A literal dot
`\\`	A literal backslash
`\* \[ \^ \$`	The literal symbol
`[.]`	A dot, escaped by a bracket expression instead
`grep -F 'a.b'`	Treat the whole pattern as a fixed string
`\Q...\E`	Quote a run of characters in PCRE

Quote patterns with single quotes so the shell passes the backslashes through untouched. Inside double quotes, the shell expands `$` and consumes some backslashes before the tool ever sees the pattern.

Shorthand Classes

Short names for common sets, provided as extensions rather than by POSIX.

`\w \W`	Word and non-word characters in GNU grep, GNU sed, gawk, PCRE, and Vim; the exact character set varies by engine and locale
`\s \S`	Whitespace and non-whitespace in GNU grep, GNU sed, gawk, PCRE, and Vim
`\d \D`	Digits and non-digits in PCRE and Vim; not defined by POSIX BRE or ERE
`\h \v`	Horizontal and vertical whitespace in PCRE; Vim assigns different meanings to both sequences

Do not use `\d` with grep's basic or extended syntax. GNU grep documents an escaped ordinary character such as `\d` as unspecified, and other grep implementations may interpret it differently. Use `[[:digit:]]`, `[0-9]`, or `grep -P` instead.

PCRE Extras

Available with `grep -P` and other Perl-compatible engines.

`.*? +? ??`	Lazy quantifiers that take the shortest match
`(?:...)`	Group without capturing it
`(?<name>...)`	Named capture group
`(?=...)`	Lookahead, text must follow
`(?!...)`	Negative lookahead
`(?<=...)`	Lookbehind, text must precede
`(?<!...)`	Negative lookbehind
`(?i)`	Case-insensitive from this point on
`\K`	Drop everything matched so far from the result

Lookarounds check their surroundings without adding them to the match, which pairs well with `grep -oP`. None of this works in sed or awk, and `grep -P` needs a build with PCRE support.

BRE, ERE, and PCRE

The same idea written three ways.

Grouping	BRE `\(ab\)`; ERE and PCRE `(ab)`
Alternation	GNU BRE `a|b`; ERE and PCRE `a|b`
One or more	GNU BRE `a\+`; ERE and PCRE `a+`
Optional	GNU BRE `a\?`; ERE and PCRE `a?`
Interval	BRE `a\{2,4\}`; ERE and PCRE `a{2,4}`
Backreference	BRE and PCRE `\1`; GNU grep and GNU sed also accept `\1` in ERE, but POSIX ERE and awk do not
\d and lookaround	PCRE supports both; BRE and ERE do not
Default in	BRE: `grep`, `sed`; ERE: `grep -E`, `sed -E`, awk; PCRE: `grep -P`

In GNU grep, BRE and ERE provide the same pattern-matching functionality with different notation. The characters `?`, `+`, `()`, `{}`, and `|` are special without backslashes in ERE and with backslashes in GNU BRE. This equivalence does not extend to POSIX ERE backreferences or PCRE-only features.

Regex in grep

Pick the flavor with a flag.

`grep 'pattern' file`	Basic regular expression, the default
`grep -E 'pattern' file`	Extended regular expression
`grep -P 'pattern' file`	Perl-compatible regular expression
`grep -F 'pattern' file`	No regex at all, fixed string
`grep -o 'pattern' file`	Print only the matched text
`grep -w 'pattern' file`	Require word boundaries around the match
`grep -i 'pattern' file`	Ignore case
`grep -v 'pattern' file`	Print the lines that do not match

Use `-o` while building a pattern to see exactly what it captures. GNU grep 3.8 and later print a warning for `egrep` and `fgrep`, so write `grep -E` and `grep -F` instead.

Regex in sed

Patterns select lines, and the same syntax drives substitutions.

`sed 's/old/new/' file`	Substitute using a basic regular expression
`sed -E 's/[0-9]+/N/g' file`	Extended syntax, with `-r` as a synonym
`sed -n '/error/p' file`	Print only matching lines
`sed '/^$/d' file`	Delete every empty line
`sed 's/\(a\)\(b\)/\2\1/' file`	Swap two captured groups
`sed 's/word/[&]/' file`	Wrap the whole match in brackets
`sed 's/error/ERROR/gI' file`	Replace every match, ignoring case
`sed -E 's/(\w+)/\U\1/' file`	Uppercase group 1 with a GNU escape

The delimiter does not have to be a slash. Writing `sed 's|/usr/bin|/usr/local/bin|'` avoids escaping every slash in a path.

Regex in awk

Patterns are extended regular expressions and sit between slashes.

`awk '/error/' file`	Print lines matching the pattern
`awk '$1 ~ /^web/' file`	Match a single field
`awk '$3 !~ /ok/' file`	Match fields that fail the pattern
`awk '/start/,/stop/' file`	Print an inclusive range of lines
`awk '{ gsub(/[0-9]+/, "N"); print }' file`	Replace every match on the line
`awk '{ sub(/^ +/, ""); print }' file`	Replace the first match only
`awk 'match($0, /[0-9]+/) { print substr($0, RSTART, RLENGTH) }' file`	Locate a match and extract it
`gawk '{ gsub(/\yroot\y/, "USER"); print }' file`	Word boundaries in gawk

Awk always uses extended syntax, so `+`, `?`, and `{n,m}` work without a backslash. When the pattern is a string rather than a `/.../` literal, every backslash needs doubling, so `gsub(/\./, "-")` becomes `gsub("\\.", "-")`.

Regex in Vim

Vim uses its own flavor, close to ERE once you turn on very magic mode.

`/pattern`	Search forward in the file
`:%s/old/new/g`	Replace every match in the file
`a\+`	One or more, since `+` needs a backslash by default
`\v`	Very magic mode, so `+`, `?`, `(`, and `|` work unescaped
`\v\d+`	Digits in very magic mode
`\<word\>`	Whole word match
`price: \zs\d\+`	Start the match after `\zs`, end it at `\ze`
`\cerror`	Ignore case for this pattern

Very magic mode is the shortcut worth remembering. Writing `:%s/\v(\w+), (\w+)/\2 \1/g` to swap two fields keeps a pattern readable instead of filling it with backslashes.

Regex in Bash

The `=~` operator inside `[[ ]]` takes an extended regular expression.

`[[ $ip =~ ^[0-9.]+$ ]]`	Test a string against a pattern
`[[ ! $name =~ ^[a-z] ]]`	Negate the test
`re='^v([0-9]+)\.([0-9]+)$'`	Keep the pattern in a variable
`[[ $tag =~ $re ]]`	Use the variable unquoted so it stays a regex
`${BASH_REMATCH[0]}`	The whole match
`${BASH_REMATCH[1]}`	The first capture group
`case $file in *.txt) ;; esac`	A reminder that `case` uses globs, not regex

Quoting the right-hand side turns the pattern into a literal string, so `[[ abc =~ "a.c" ]]` fails while `[[ abc =~ a.c ]]` succeeds. Store the pattern in a variable when it contains spaces or quotes.

Common Patterns

Working starting points to copy and adjust.

`grep -E '\b([0-9]{1,3}\.){3}[0-9]{1,3}\b'`	Anything shaped like an IPv4 address
`grep -P '\b((25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]?[0-9])\.){3}(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]?[0-9])\b'`	An IPv4 candidate with each octet under 256
`grep -E '[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}'`	An email-shaped address
`grep -E '[0-9]{4}-[0-9]{2}-[0-9]{2}'`	An ISO date such as `2026-08-28`
`grep -E '#[0-9a-fA-F]{6}\b'`	A six-digit hex color
`grep -nE '[[:space:]]+$'`	Trailing whitespace, with line numbers
`grep -cE '^[[:space:]]*$'`	Count blank lines
`grep -E '^[[:space:]]*#'`	Commented-out configuration lines
`grep -vE '^[[:space:]]*(#|$)'`	Everything except comments and blank lines
`grep -E '\b([[:alpha:]]+) \1\b'`	A word accidentally repeated twice
`grep -oE '"[^"]*"'`	A double-quoted string, quotes included
`grep -oP 'user=\K\S+'`	The value after a `user=` key, key excluded

The address patterns describe a shape rather than validate it. Use them to pull candidates out of logs, and check the results with a real parser when correctness matters.

Troubleshooting

Common regex problems and what to check first.

`\d fails or behaves unexpectedly`	BRE and ERE do not define `\d`; use `[[:digit:]]`, `[0-9]`, or `grep -P`
`+ or ? matched literally`	The pattern is basic syntax; escape them as `\+` and `\?`, or switch to `grep -E`
`Alternation found nothing`	ERE takes a bare pipe, BRE takes an escaped one; `grep -E` is the simpler fix
`The shell changed the pattern`	Wrap the pattern in single quotes so backslashes and `$` survive
`[a-z] also matched uppercase`	Range order follows the locale; use `LC_ALL=C` or `[[:lower:]]`
`Lookahead or lookbehind rejected`	BRE and ERE do not support them; use `grep -P`, or the engine's own syntax such as Vim's `\@=`
`grep -P is not supported`	The build lacks PCRE; use `pcre2grep`, `perl -ne`, or rewrite the pattern
`\b did nothing in awk`	Awk reads `\b` as a backspace; use `\y` in gawk or `\<` and `\>`
`The match ran too far`	Quantifiers are greedy; use a negated class such as `[^"]*`, or a lazy `.*?` with `-P`
`Intervals matched literally in Vim`	Vim needs `\{2,4}` in magic mode, or `\v` first

Related Guides

Longer walkthroughs and per-tool references.

Regular Expressions Basics	Learn the building blocks from the ground up
Regular Expressions in Grep	Every grep flavor with worked examples
grep Cheatsheet	Search options, recursion, and context output
sed Cheatsheet	Substitution, addresses, and in-place editing
awk Cheatsheet	Fields, patterns, actions, and built-in variables
Vim Cheatsheet	Motions, editing, search, and replace