A BASH SCRIPTING COURSE

Regular Expressions in Bash

12 chapters — grep · sed · awk · [[=~]] · PCRE

01 What Are Regular Expressions
02 Literals, the Dot, and Escaping
03 Anchors
04 Character Classes
05 Quantifiers
06 Groups, Alternation, and Back-references
07 grep and egrep in Depth
08 sed and Regex in Depth
09 Regex in awk
10 Bash [[=~]] and BASH_REMATCH
11 PCRE: Lookahead, Lookbehind, and Advanced Features
12 Your Regex Toolkit: Patterns, Testing, and Real Pipelines

BRE • ERE • PCRE • BASH_REMATCH • perl -ne

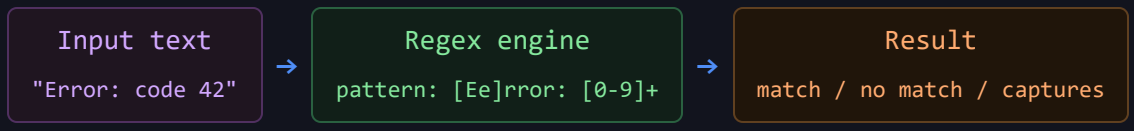
CHAPTER 1

What Are Regular Expressions

The Core Idea

A **regular expression** (regex) is a pattern that describes a set of strings. Instead of saying "find the exact text `error`", a regex lets you say "find any line that contains the word *error* followed by a colon and then one or more digits." It is a mini-language for describing text shapes.

Regexes appear constantly in Bash work — in `grep` for filtering, in `sed` for substitution, in `awk` for field processing, and directly inside Bash's own `[[ =~ ]]` conditional. Learning regex once unlocks all of them simultaneously.



The regex engine reads the pattern and the input text, then reports one of three outcomes: a **match** (the pattern was found), **no match**, or — when using capturing groups — the matched text and any captured sub-strings.

Where Bash Uses Regular Expressions

You will encounter regex in four main places when working at the Bash command line:

`grep`	`sed`	`awk`	`[[ =~ ]]`
BRE / ERE / PCRE	BRE / ERE	ERE	ERE
Filter lines from files or pipelines	Find and replace text in streams	Pattern matching and field processing	Regex conditionals inside Bash scripts

```
# grep: print lines containing a pattern
grep '[Ee]rror' app.log

# sed: substitute text matching a pattern
sed 's/[0-9]\+/NUM/g' data.txt
```

```
# awk: process lines where a field matches a pattern
awk '/^ERROR/ { print $0 }' app.log

# [[ =~ ]]: test a string against a regex in a script
if [[ "$input" =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2}$ ]]; then
    echo "Valid date format"
fi
```

The Three Flavours You Will Meet

Not all regex is the same. The Linux/Bash ecosystem has three distinct flavours, each with slightly different rules for which characters are "special". Understanding which tool uses which flavour is one of the most important things you can learn.

BRE — BASIC

- Default in `grep` and `sed`
- `(` `)` are literal — groups need `\(` `\)`
- `+` `?` are literal — need `\+` `\?`
- `{n}` needs `\{n\}`
- Oldest standard — most portable
- Use: `grep 'pat'`, `sed 's/pat/'`

ERE — EXTENDED

- `grep -E`, `sed -E`, `awk`, `[[ =~ ]]`
- `(` `)` group without backslash
- `+` `?` work directly
- `{n}` works without escaping
- Cleaner syntax — recommended default
- Use: `grep -E 'pat'`, `sed -E`

PCRE — PERL COMPATIBLE

- `grep -P` only (GNU grep)
- Lookahead `(?=...)`
- Lookbehind `(?<=...)`
- Non-greedy `.*?`
- Named groups `(?P<name>)`
- Most powerful — least portable

> **Practical rule:** use `-E` (ERE) by default for `grep` and `sed` — it is cleaner and available everywhere. Drop to BRE only when writing POSIX-portable scripts. Reach for `-P` (PCRE) only when you genuinely need lookahead/lookbehind or non-greedy matching.

How the Regex Engine Works

Understanding the engine's behaviour prevents a large class of surprises. The engine follows two simple rules:

1. **Left to right:** the engine tries to match the pattern starting at the leftmost character of the input. If it fails, it moves one character to the right and tries again.
2. **Greedy by default:** quantifiers like `*` and `+` consume as many characters as possible while still allowing the overall match to succeed.

```
# Leftmost match wins — engine finds the first position that works
echo "the cat sat on the mat" | grep -oE '[a-z]at'
cat
sat
mat   # -o prints each match on its own line

# Greedy: .* consumes as much as possible
echo "<b>bold</b> and <i>italic</i>" | grep -oE '<.*>'
<b>bold</b> and <i>italic</i>   # matched the whole thing!

# Fix: use a negated class to stop at the first >
echo "<b>bold</b> and <i>italic</i>" | grep -oE '<[^>]*>'
<b>
</b>
<i>
</i>
```

Your First Regex Patterns

Let's build familiarity with the most important regex building blocks. These work in all three flavours.

Literal characters

The simplest regex is just the text you want to find. Every letter and digit matches itself.

```
# Find all lines containing "error" (lowercase only)
grep 'error' app.log

# Find lines containing "404"
grep '404' access.log

# Most punctuation is also literal — but some are special (see below)
grep 'v1.0' changelog.txt   # . is special — matches any char here
grep 'v1\.0' changelog.txt  # \. matches a literal dot
```

Special (metacharacter) characters

These characters have special meaning in regex. To match them literally, escape with `\`:

Character	Special meaning	To match literally
`.`	Any single character (except newline)	`\.`
`*`	Zero or more of the preceding	`\*`
`+`	One or more (ERE/PCRE)	`\+`
`?`	Zero or one (ERE/PCRE)	`\?`
`^`	Start of line (anchor)	`\^`
`$`	End of line (anchor)	`\$`
`[`	Start of character class	`\[`
`(`	Start of group (ERE/PCRE)	`\(`
`|`	Alternation (ERE/PCRE)	`\\|`
`\`	Escape character	`\\`

A quick tasting menu

```
# . — match any single character
echo "cat bat hat" | grep -oE '.at'
```

```
cat
bat
hat

# ^ — match only at the start of the line
echo -e "apple\nbanana\napricot" | grep '^a'
apple
apricot

# $ — match only at the end of the line
echo -e "run\nfun\nbun\nsun" | grep 'un$'
run
fun
bun
sun

# [abc] — match one character from the set
echo "cat dog bat" | grep -oE '[cb]at'
cat
bat

# * — zero or more of the preceding
echo -e "ac abc abbc abbbc" | grep -oE 'ab*c'
ac
abc
abbc
abbbc
```

Testing Regex at the Command Line

The fastest way to learn regex is to experiment interactively. Here are the best testing patterns to use at a Bash prompt:

```
# Pattern 1: pipe echo into grep -E to test a pattern instantly
echo "test string here" | grep -E 'your_pattern'

# Pattern 2: use -o to show only the matched text (not the whole line)
echo "order #12345 total $99.50" | grep -oE '[0-9]+'
```

```
12345
99
50

# Pattern 3: use --color to highlight matches in context
grep --color=always -E '[0-9]+' data.txt

# Pattern 4: test [[ =~ ]] in the shell directly
str="hello123"
[[ "$str" =~ [0-9]+ ]] && echo "contains digits" || echo "no digits"
contains digits

# Pattern 5: use a here-string to test sed substitutions
sed -E 's/[0-9]+/NUM/g' <<< "There are 42 items and 7 errors"
There are NUM items and NUM errors
```

A Comparison: Shell Globbing vs Regex

New Bash users sometimes confuse **shell glob patterns** (used in filename expansion) with **regular expressions** (used in text tools). They look similar but mean completely different things:

Pattern	In shell glob	In regex
`*`	Zero or more of *any character*	Zero or more of the *preceding element*
`?`	Exactly *one* of any character	Zero or one of the preceding element
`[abc]`	One character from the set	One character from the set (same)
`.`	A literal dot	Any single character (not newline)
`**`	Recursive directory match (Bash 4+)	Not valid — just greedy `*` applied twice

```
# Shell glob: * means "any filename characters"
ls *.log          # lists all .log files — * is a glob wildcard
```

```
# Regex: * means "zero or more of the preceding element"
grep 'log*' file   # matches "lo", "log", "logg", "loggg" etc.
                   # NOT "anything ending in log"!

# To match "anything then .log" in a regex:
grep '.*\.log' file  # .* = any chars, \. = literal dot, log = literal
```

> **This is the single most common source of confusion for Bash beginners learning regex.** Shell globbing and regex look alike but behave completely differently. Globs live in the shell; regex lives inside text-processing tools. `ls *.log` and `grep '*.log'` mean entirely different things.

Quick Reference — Chapter 1

THE THREE FLAVOURS

`grep 'pat'`	BRE — Basic Regular Expressions (default)
`grep -E 'pat'` / `egrep`	ERE — Extended Regular Expressions
`grep -P 'pat'`	PCRE — Perl Compatible (GNU grep only)
`sed 's/pat/'` / `awk '/pat/'`	BRE and ERE respectively (sed -E for ERE)
`[[ str =~ pat ]]`	ERE — Bash built-in regex test

QUICK TESTING COMMANDS

`echo "str" | grep -E 'pat'`	Test if a pattern matches a string
`echo "str" | grep -oE 'pat'`	Extract and display only the matched text
`sed -E 's/pat/rep/' <<< "str"`	Test a substitution on a string
`[[ "str" =~ pat ]] && echo yes`	Test a pattern in a Bash conditional

> **What is coming next:** Chapter 2 dives into the building blocks — literal characters, the dot metacharacter, and the backslash escape. You will learn exactly which characters are "special" in a regex and how to match them literally when you need to.

CHAPTER 2

Literals, the Dot, and Escaping

Literal Characters

The simplest possible regex is a sequence of **literal characters** — characters that match themselves exactly. Every letter, digit, and most punctuation marks are literals. The engine looks for that exact sequence anywhere in the input line.

```
# A literal pattern matches anywhere on the line
echo -e "apple\napplesauce\npineapple\nbanana" | grep 'apple'
apple
applesauce
pineapple
# "banana" has no match — 'apple' must appear somewhere on the line

# Matching is case-sensitive by default
echo -e "Error\nerror\nERROR" | grep 'error'
error   # only the exact lowercase version matches

# Use -i for case-insensitive matching
echo -e "Error\nerror\nERROR" | grep -i 'error'
Error
error
ERROR
```

```
PATTERN: 'CAT' — WHERE DOES IT MATCH IN EACH LINE?

cat → match at position 0 concatenate → match at position 3 the cat sat →

match at position 4 dog → no match
```

Digits and spaces are literals too

```
# Digits match themselves
grep '404' access.log          # lines containing "404"
```

```
grep '2026' dates.txt            # lines containing "2026"

# A space is a literal space
grep 'hello world' file.txt      # the space must be there

# Multiple words: the whole string is one pattern
grep 'connection refused' syslog # all three words in sequence
```

The Dot — Matching Any Single Character

The dot `.` is the first metacharacter most people learn. It matches **any single character except a newline**. Think of it as a wildcard slot: put a `.` where any one character is acceptable.

```
# . matches exactly one character — any character
echo -e "cat\nbat\nhat\nsat\nat" | grep '.at'
cat
bat
hat
sat
# "at" alone has no match — . requires exactly one character before "at"

# Two dots = two characters (any)
echo -e "a1b\naxb\na b\nabc" | grep 'a.b'
a1b
axb
a b
# "abc" has no match — . requires exactly one char between a and b

# .* means "any number of any characters" (very common pattern)
echo -e "start-middle-end\nstart-end\nstart123end" | grep 'start.*end'
start-middle-end
start-end
start123end
```

```
PATTERN: C.T
grep 'c.t' <<< "word"
```

```
PATTERN: ....
grep -E '^....$' (exactly 4 chars)
```

```
cat ✓ (c + a + t) cut ✓ (c + u + t)
c9t ✓ (c + 9 + t) c t ✓ (c + space +
t) ct ✗ (nothing between c and t)
coat ✗ (two chars between c and t)
```

```
cats ✓ (4 characters) 1234 ✓ (4
characters) r!@# ✓ (4 characters) cat
✗ (3 characters) hello ✗ (5
characters) ^ and $ anchor to full line
```

The dot trap — matching when you mean a literal period

Because `.` matches any character, it will silently match things you didn't intend. This is the most common regex mistake in Bash scripts:

```
# Intended: match version "1.0" specifically
echo -e "v1.0\nv1x0\nv100" | grep '1.0'
v1.0
v1x0   # oops — . matched x

# Fix: escape the dot with \
echo -e "v1.0\nv1x0\nv100" | grep '1\.0'
v1.0   # only the real version matches now

# Real-world example: matching an IP address pattern safely
echo -e "192.168.1.1\n192X168X1X1" | grep '192\.168\.'
192.168.1.1   # only the real IP

# Matching a file extension: .txt must be a literal dot
ls | grep '\.txt$'   # ends with a literal .txt
```

The Backslash Escape

The backslash `\` removes the special meaning from the character that follows it. After a backslash, a metacharacter becomes a literal. This is called **escaping**.

You write	Matches	Example input that matches
`\.`	A literal dot	`file.txt`, `192.168.1.1`
`\*`	A literal asterisk	`5 * 3`, `footnote*`

`\+`	A literal plus sign (in BRE)	`a+b`, `C++`
`\?`	A literal question mark (in BRE)	`why?`
`\^`	A literal caret	`^start`, `a^b`
`\$`	A literal dollar sign	`$100`, `$VAR`
`\[`	A literal opening bracket	`[note]`
`\(`	A literal opening paren (in ERE)	`(optional)`
`\\|`	A literal pipe (in ERE)	`a|b`
`\\`	A literal backslash	`C:\Users\`
`\n`	Newline (in GNU tools)	Embedded newline in pattern space
`\t`	Tab character (in GNU tools)	Tab-separated data

```
# Escaping a dollar sign (common in shell scripts)
echo 'Price: $99.50' | grep '\$[0-9]'
Price: $99.50

# Matching a literal asterisk in text
echo -e "5 * 3 = 15\n5 x 3 = 15" | grep '\*'
5 * 3 = 15

# Matching a Windows-style path (literal backslashes)
echo 'C:\Users\emuba\file.txt' | grep 'Users\\emuba'
C:\Users\emuba\file.txt

# Matching a literal question mark
echo -e "Are you sure?\nAre you sure" | grep 'sure\?$'
Are you sure?
# In BRE: \? is a literal ?  — in ERE: \? means "zero or one"
```

The Double-Escape Problem — Shell Quoting and Regex

Here is where many Bash users get confused. When you write a regex on the command line, there are **two layers of interpretation**: first the shell processes the quoting, then the regex engine processes what remains.

```
TWO LAYERS: SHELL FIRST, THEN REGEX ENGINE

Command you type: grep '192\.168' file  Shell sees: single quotes — passes everything through unchanged  Regex engine gets: 192\.168 → matches 192.168

Command you type: grep "192\\.168" file  Shell sees: double quotes — \\ becomes \  Regex engine gets: 192\.168 → same result

Command you type: grep "192\.168" file  Shell sees: double quotes — \. stays as \. (\ not special before .)  Regex engine gets: 192\.168 → same result (this one usually works)
```

> **Use single quotes for regex patterns.** Single quotes prevent the shell from interpreting *anything* — no variable expansion, no backslash processing. The pattern reaches the regex engine exactly as written. Only switch to double quotes when you need to expand a shell variable inside the pattern.

```
# Single quotes: shell passes everything through literally (preferred)
grep '\$[0-9]\+' file.txt       # shell does nothing; engine sees \$[0-9]\+

# Double quotes: shell processes \\ → \ before the engine sees it
grep "\\$[0-9]\\+" file.txt     # shell: \\ → \, then \$ → $, then \+ → \+...

# Double quotes needed when expanding a variable in the pattern
TERM="error"
grep "$TERM" app.log            # $TERM expands to "error" — double quotes n

# Variable in pattern with special chars: escape the variable content
TERM="v1.0"
# BAD: . inside $TERM is treated as regex metachar
grep "$TERM" file.txt           # matches v1x0, v1.0, v100 etc.
# BETTER: use grep -F for literal string search
grep -F "$TERM" file.txt        # -F = fixed string, no regex
```

grep -F — When You Don't Want Regex At All

Sometimes you have a search term that contains special characters and you want to match it *literally* — no regex processing at all. The `-F` flag (fixed string) tells `grep` to treat the entire pattern as plain text.

```
# Searching for a URL — slashes, dots, question marks all literal
grep -F 'https://example.com/api?v=2' access.log

# Searching for a literal asterisk or plus sign
grep -F 'a+b=c' math.txt

# Searching for a pattern that came from a variable (safest approach)
SEARCH="$100.00"
grep -F "$SEARCH" invoice.txt    # $ and . treated as literals

# -F is also significantly faster on large files
# because no regex engine compilation is needed
grep -F "Connection refused" /var/log/syslog
```

Case Sensitivity

Regex is case-sensitive by default in all Bash tools. There are two ways to match regardless of case:

```
# -i flag: case-insensitive for grep
grep -i 'error' app.log          # matches Error, ERROR, error, ErRoR

# -i flag: case-insensitive for sed (GNU sed)
sed 's/error/FOUND/Ig' app.log   # I flag on s command

# In regex: use a character class to be explicit about what you allow
grep '[Ee]rror' app.log          # matches Error or error (not eRror)
grep '[Ee][Rr][Rr][Oo][Rr]' app.log  # matches any capitalisation

# In [[ =~ ]]: no -i flag — use character classes
```

```
word="ERROR"
[[ "$word" =~ [Ee][Rr][Rr][Oo][Rr] ]] && echo "matched"
```

Practical Patterns — Putting It Together

Matching version numbers

```
# Match "1.2.3" style version strings — escape all dots
grep '[0-9]\+\.[0-9]\+\.[0-9]\+' changelog.txt        # BRE
grep -E '[0-9]+\.[0-9]+\.[0-9]+' changelog.txt        # ERE

# Match a specific version literally
grep -F '2.14.1' changelog.txt
```

Matching file extensions

```
# Files ending in .log (literal dot, then log, then end of line)
ls | grep '\.log$'

# Files ending in .tar.gz (two literal dots)
ls | grep '\.tar\.gz$'

# Any hidden file (starts with a dot)
ls -a | grep '^\..'   # ^ then literal dot then any char (at least 2 char nam
```

Matching decimal numbers

```
# Match a decimal number like 3.14, 99.99, 0.5
grep -E '[0-9]+\.[0-9]+' data.txt

# Match in a sed substitution
sed -E 's/[0-9]+\.[0-9]+/PRICE/g' invoice.txt

# Test in [[ =~ ]] — is this string a decimal number?
```

```
val="3.14"
[[ "$val" =~ ^[0-9]+\.[0-9]+$ ]] && echo "is decimal"
is decimal
```

Common traps and fixes

Trap	Wrong pattern	Fixed pattern	Why
Unescaped dot in version	`grep '2.0'`	`grep '2\.0'`	`.` matches any char
Dot in domain name	`grep 'example.com'`	`grep 'example\.com'`	Matches `exampleXcom` too
Dollar sign in price	`grep '$99'`	`grep '\$99'` or `grep -F '$99'`	`$` is an anchor
Asterisk in maths expression	`grep '3*4'`	`grep '3\*4'` or `grep -F '3*4'`	`*` is a quantifier
Dot in file extension check	`grep '.txt$'`	`grep '\.txt$'`	Matches `atxt` `1txt` etc.

Quick Reference — Chapter 2

LITERALS AND THE DOT

```
a b c 0 9 space
```

All letters, digits, and spaces are literals — match themselves

`.`	Matches any single character except newline
`\.`	Matches a literal dot (escaped metacharacter)
`.*`	Matches zero or more of any character (greedy)

ESCAPING AND QUOTING

`\`

Escape — removes special meaning from the next character

`'pattern'`

Single quotes — shell passes pattern through unchanged (preferred)

`"$VAR"`

Double quotes — use when you need variable expansion in the pattern

`grep -F 'literal'`	Fixed string — no regex processing at all
`grep -i 'pattern'`	Case-insensitive match

> **What is coming next:** Chapter 3 covers anchors — `^` (start of line), `$` (end of line), and word boundaries (`\b` , `\<` , `\>`). Anchors let you pin your pattern to a specific position in the line rather than matching it anywhere.

CHAPTER 3

Anchors

What Are Anchors?

An **anchor** does not match a character — it matches a *position* in the text. The two most important anchors, `^` and `$`, pin your pattern to the start or end of a line. Without anchors, a pattern can match anywhere on the line; with anchors, you specify exactly where it must appear.

Think of anchors as invisible markers that say: "the match must begin here" or "the match must end here".

^

Start of line

Matches the position immediately before the first character of the line. The pattern that follows must appear at the very beginning.

`Works in: grep, sed, awk, [[ =~ ]]`

$

End of line

Matches the position immediately after the last character of the line (before the newline). The pattern that precedes must appear at the very end. `Works in: grep, sed, awk, [[ =~ ]]`

\b

Word boundary (GNU)

Matches the position between a word character (`\w`) and a non-word character (`\W`) — i.e. the edge of a word. Matches at both the start and end of words. `Works in: GNU grep, GNU sed, GNU awk`

\< \>

Word start / end (POSIX)

`\<` matches the position at the start of a word; `\>` matches the position at the end. Together they are the portable alternative to `\b`. `Works in: grep, sed (POSIX BRE/ERE)`

^ — The Start-of-Line Anchor

`^` asserts that what follows must appear at the very beginning of the line. Without `^`, the pattern can match anywhere; with `^`, it must be the first thing on the line.

```
PATTERN: ^ERROR — WHERE DOES IT MATCH?

^Error on line 1   ✓ match — "Error" is at the start

^2024 Error found  ✗ no match — line starts with "2024", not "Error"

^ Error            ✗ no match — line starts with a space
```

```bash
# Lines that start with "Error"
grep '^Error' app.log

# Lines that start with a digit
grep '^[0-9]' data.txt

# Lines that start with optional whitespace then a # (comment lines)
grep '^[[:space:]]*#' config.conf

# Delete all comment lines from a config file
sed '/^[[:space:]]*#/d' config.conf

# In awk: process lines that start with "WARN"
awk '/^WARN/ { print NR": "$0 }' app.log

# In [[ =~ ]]: does this string start with a digit?
[[ "$line" =~ ^[0-9] ]] && echo "starts with digit"
```

^ inside a character class means negation — not an anchor

```bash
# ^ at the START of the pattern = start-of-line anchor
grep '^cat'         # line must START with "cat"

# ^ as the FIRST char inside [...] = negation (match anything NOT in set)
grep '[^abc]'       # match any character that is NOT a, b, or c

# ^ anywhere else inside [...] = literal caret character
grep '[a^b]'        # matches a, ^, or b — ^ is literal here
```

> `^` **has two completely different meanings depending on context.** At the start of a pattern or after `|` it is a start-of-line anchor. As the first character inside `[...]` it negates the class. Everywhere else inside `[...]` it is just a literal caret. Memorise this — it trips up everyone.

$ — The End-of-Line Anchor

`$` asserts that what precedes must appear at the very end of the line. The regex engine matches `$` against the position just before the newline character (which SED and grep strip off before processing).

```
PATTERN: \.LOG$ — WHERE DOES IT MATCH?

a p p . l o g $   ✓ match — ends with .log

a p p . l o g . 1 $   ✗ no match — ends with .1 not .log

e r r o r . l o g · $   ✗ no match — trailing space after .log
```

```
# Lines that end with a semicolon
grep ';$' source.js

# List only .log files
ls | grep '\.log$'

# Lines that end with one or more digits
grep -E '[0-9]+$' data.txt

# Remove trailing whitespace (common pre-commit fix)
sed 's/[[:space:]]*$//' file.txt

# Delete lines that end with a backslash (continuation lines)
grep -v '\\$' makefile   # \\ matches literal \, then $ anchors to end
```

Combining ^ and $ Together

When you use `^` and `$` together, you anchor the pattern to the *entire line* — the match must cover the line from start to finish.

```
# Match blank lines (nothing between start and end)
grep '^$' file.txt             # exactly empty
grep '^[[:space:]]*$' file.txt  # blank or whitespace-only

# Delete blank lines
sed '/^$/d' file.txt

# Match lines that contain ONLY digits (nothing else)
grep -E '^[0-9]+$' data.txt

# Validate that a variable is a positive integer
[[ "$val" =~ ^[0-9]+$ ]] && echo "valid integer"

# Match lines that are exactly 8 characters long
grep -E '^.{8}$' file.txt

# Match lines containing ONLY uppercase letters and spaces
grep -E '^[A-Z ]+$' file.txt

# In sed: add text at the very start of every line
sed 's/^/    /' file.txt        # indent every line by 4 spaces

# In sed: add text at the very end of every line
sed 's/$/;/' file.txt           # append semicolon to every line
```

Word Boundaries

Line anchors pin you to the start or end of the entire line. But often you want to match a word only when it appears as a *complete word* — not as part of a longer word. Word boundary anchors solve this.

```
WORD BOUNDARIES IN: "THE CAT CONCATENATE SCAT"

|the| |cat| |concatenate| |scat|  | marks a word boundary position (between word-char and non-word-char) Pattern \bcat\b matches: cat (standalone word only)
```

```
Pattern \bcat\b does NOT match: concatenate, scat
```

\b — word boundary (GNU)

A word character is any letter, digit, or underscore (`[[:alnum:]_]`). A word boundary is the position between a word character and a non-word character (or between a word character and the start/end of the line).

```
# Match the word "cat" but not "concatenate" or "scat"
echo "the cat concatenate scat scatter" | grep -oE '\bcat\b'
cat

# Replace the word "error" but not "errors" or "no_error_code"
sed 's/\berror\b/ERROR/g' app.log

# Count occurrences of the exact word "the"
grep -oE '\bthe\b' essay.txt | wc -l

# \b also anchors to start/end of line
echo "error at line 5" | grep -E '\berror\b'
error at line 5   # matches — "error" is a standalone word at line start
```

\B — non-word boundary (GNU)

`\B` is the opposite of `\b` — it matches a position that is *not* at a word boundary, i.e. the middle of a word.

```
# Match "cat" only when it is part of a longer word
echo "cat concatenate scat" | grep -oE '\Bcat\B'
cat   # only the "cat" inside "con-cat-enate" matches

# Practical: find compound words containing "data"
grep -E '\Bdata\B' source.py   # matches metadata, database but not standalon
```

\< and \> — POSIX word edges

These are the POSIX-portable way to match word boundaries. `\<` matches the start of a word; `\>` matches the end. They work in GNU grep and sed in BRE mode without `-E`, making them the safest choice in portable scripts.

```
# Match whole word "cat" using POSIX word boundaries
grep '\<cat\>' file.txt

# Same as \bcat\b but more portable
echo "the cat concatenate scat" | grep -o '\<cat\>'
cat

# In sed: replace whole word only
sed 's/\<error\>/ERROR/g' app.log

# Only word-start boundary (match words beginning with "pre")
grep '\<pre' dictionary.txt   # preview, prepare, prefix — but not "impressiv

# Only word-end boundary (match words ending in "ing")
grep 'ing\>' text.txt          # running, jumping — but not "rings"
```

Anchor Portability — What Works Where

Anchor	grep (BRE)	grep -E (ERE)	grep -P (PCRE)	sed	awk	[[=~]]
`^`	✓	✓	✓	✓	✓	✓
`$`	✓	✓	✓	✓	✓	✓
`\b`	GNU only	GNU only	✓	GNU only	GNU awk	GNU bash
`\B`	GNU only	GNU only	✓	GNU only	GNU awk	GNU bash
`\< \>`	✓	✓	✗	✓	✗	✗

> **Portable scripts:** use `\<` and `\>` when writing scripts for POSIX environments or macOS (BSD tools). Use `\b` freely in interactive one-liners on Linux where GNU tools are guaranteed. In `[[ =~ ]]` and awk, use character class workarounds if portability matters.

Practical Anchor Patterns

Input validation in scripts

```
# Is input a valid positive integer?
is_integer() {
    [[ "$1" =~ ^[0-9]+$ ]]
}
is_integer "42"   && echo "ok"    # ok
is_integer "3.14" && echo "ok"  # no match — dot disqualifies it
is_integer "12x"  && echo "ok"  # no match — trailing x

# Is input a valid ISO date (YYYY-MM-DD)?
is_date() {
    [[ "$1" =~ ^[0-9]{4}-[0-9]{2}-[0-9]{2}$ ]]
}
is_date "2026-06-11"  && echo "valid"
is_date "06/11/2026"  && echo "valid"  # no match — wrong format

# Does the string start with http:// or https://?
[[ "$url" =~ ^https?:// ]] || echo "not a URL"
```

Config and log file tasks

```
# Print only non-blank, non-comment lines from a config file
grep -Ev '^[[:space:]]*(#|$)' sshd_config

# Find lines where a key appears at the start (key=value format)
grep '^MaxSessions' /etc/ssh/sshd_config
```

```
# Find duplicate words (same word at end and start of adjacent lines)
grep -E '\bthe\b.*\bthe\b' essay.txt   # "the" twice on same line

# Show only lines that are pure section headers [SectionName]
grep -E '^[[:space:]]*\[[[:alnum:]_]+\][[:space:]]*$' app.conf

# Add a line number prefix only to non-blank lines
grep -n '.' file.txt   # . matches any char = any non-empty line
```

Whole-word replacement in sed

```
# Replace standalone "cat" — not "cats", "tomcat", "concatenate"
sed 's/\bcat\b/dog/g' file.txt           # GNU sed
sed 's/\<cat\>/dog/g' file.txt          # POSIX portable

# Rename a function across a codebase (word boundary prevents partial matches)
sed -i 's/\bgetUser\b/fetchUser/g' *.js

# Replace a word at the start of a line only
sed 's/^TODO/DONE/' tasks.txt

# Append a comma to every line that doesn't already end with one
sed '/,$/! s/$/,/' list.txt   # /,$/! = lines NOT ending in comma
```

Common Anchor Mistakes

Mistake	Pattern used	Problem	Fix
Matching anywhere instead of start	`grep 'Error'`	Matches "Error" mid-line too	`grep '^Error'`
Trailing space breaks end anchor	`grep '\.log$'`	Fails if line has trailing space	`grep '\.log[[:space:]]*$'`

Partial word match	`sed 's/cat/dog/g'`	Changes "concatenate" → "condogenate"	`sed 's/\bcat\b/dog/g'`
`^` inside class negates instead of anchors	`grep '[^a-z]'`	Matches non-lowercase chars, not start of line	Use `^[a-z]` outside brackets for anchoring
Using `\b` on macOS/BSD	`grep '\bword\b'`	BSD grep doesn't support `\b`	`grep '\<word\>'` or `grep -E '\bword\b'` with GNU grep
Forgetting `$` in validation	`[[ "$v" =~ ^[0-9]+ ]]`	"123abc" passes — only checks the start	`[[ "$v" =~ ^[0-9]+$ ]]`

Quick Reference — Chapter 3

LINE ANCHORS

`^pattern`	Pattern must appear at the start of the line
`pattern$`	Pattern must appear at the end of the line
`^pattern$`	Pattern must match the entire line
`^$`	Match a completely empty line
`^[[:space:]]*$`	Match a blank or whitespace-only line

WORD BOUNDARIES

`\bword\b`	Match whole word — GNU tools (grep, sed, awk)
`\<word\>`	Match whole word — POSIX portable (grep, sed)
`\b`	Position between word-char and non-word-char
`\B`	Position inside a word (not at a boundary)
`\<`	Start of a word (POSIX)

`\>`	End of a word (POSIX)

> **What is coming next:** Chapter 4 covers character classes — `[abc]`, `[a-z]`, negated classes `[^...]`, and the full set of POSIX named classes like `[[:alpha:]]`, `[[:digit:]]`, and `[[:space:]]`. Character classes give you precise control over which characters are allowed at any position in your pattern.

CHAPTER 4

Character Classes

What Is a Character Class?

A **character class** is a set of characters enclosed in square brackets `[...]`. The class matches any single character from that set. Where the dot `.` matches *any* character, a character class matches *one specific character chosen from a defined set* — giving you precise control over what is acceptable at that position.

```
ANATOMY OF A CHARACTER CLASS

[aeiou]   → matches any one vowel: a, e, i, o, or u

[a-z]     → matches any one lowercase letter (a range)

[0-9]     → matches any one digit

[a-zA-Z0-9]   → matches any letter or digit (multiple ranges)

[^aeiou]  → negated class — matches any one character that is NOT a vowel

[abc-z]   → matches a, b, c, or any letter from c to z
```

A character class always matches **exactly one character**. To match multiple characters you combine it with a quantifier (covered in Chapter 5): `[0-9]+` matches one or more digits.

Simple Character Classes

```
# Match any vowel
echo "hello world" | grep -oE '[aeiou]'
e
o
o
```

```
# Match "grey" or "gray"
echo -e "grey\ngray\ngruy" | grep 'gr[ae]y'
grey
gray

# Match any single hex digit
grep -oE '[0-9A-Fa-f]' <<< "colour #ff3a2b"
f
f
3
a
2
b

# Match a bracket character (open or close)
grep '[()]' source.c     # any line with a parenthesis

# Match "colour" or "color" (British vs American spelling)
grep 'colo[u]r\|colour' file.txt
# simpler: u is optional so [u] works but see Chapter 5 for ? quantifier
```

Ranges Inside Character Classes

A hyphen `-` between two characters inside a class defines a range of characters based on their ASCII (or locale) order. Common ranges:

Range	Matches	ASCII positions
`[0-9]`	Any digit	48–57
`[a-z]`	Lowercase letters	97–122
`[A-Z]`	Uppercase letters	65–90
`[a-zA-Z]`	Any letter	65–90, 97–122
`[a-zA-Z0-9]`	Letter or digit	Combined

`[a-f]`	Hex lowercase letters a–f	97–102
`[!-~]`	All printable ASCII (excl. space)	33–126

```
# Match a word made of only lowercase letters
grep -E '^[a-z]+$' words.txt

# Match a 3-letter uppercase acronym
grep -E '\b[A-Z]{3}\b' text.txt

# Match a hexadecimal colour code like #3fa2c0
grep -E '#[0-9A-Fa-f]{6}' styles.css

# Match a line containing only digits and hyphens (like a date 2026-06-11)
grep -E '^[0-9-]+$' data.txt
```

> **Range behaviour depends on the locale.** In a UTF-8 locale, `[a-z]` may include accented characters between `a` and `z` in the collation order, giving unexpected results. For reliable ASCII-only matching use `LC_ALL=C` or switch to POSIX named classes like `[[:lower:]]` which always behave predictably regardless of locale.

Negated Classes — [^...]

When `^` is the first character inside a class, it **negates** the class — matching any single character that is *not* in the set. This is one of the most useful patterns in regex because it lets you match "everything up to a delimiter".

```
# Match any character that is NOT a digit
grep -oE '[^0-9]' <<< "abc123def"
a
b
c
d
e
f
```

```
# Extract text before the first colon on each line
grep -oE '^[^:]+' /etc/passwd   # matches from start up to (not including) fi

# The [^delimiter]* trick — match up to the first occurrence of a char
# (the lazy-quantifier workaround from Chapter 10 of the SED course)
echo "<b>bold</b> and <i>italic</i>" | grep -oE '<[^>]*>'
<b>
</b>
<i>
</i>

# Strip all non-alphanumeric characters
sed 's/[^a-zA-Z0-9]//g' <<< "hello, world! 2026"
helloworld2026

# Match a quoted string (no embedded quotes)
grep -oE '"[^"]+"' data.json   # " then one or more non-quote chars then "
```

```
PATTERN [^,]+ ON "ALICE,30,LONDON"
Alice ✓ (chars before first comma) 30 ✓ (chars between commas) London ✓ (chars after last comma) Each field is "one or more characters that are not a comma"
```

Special Characters Inside Classes

Most regex metacharacters lose their special meaning inside `[...]` and become literals. But a few characters need care:

Character	Inside `[...]`	How to include it literally
`]`	Closes the class	Put it **first**: `[]abc]`
`-`	Range operator between two chars	Put it **first or last**: `[-abc]` or `[abc-]`
`^`	Negation if it is the **first** char	Put it anywhere **except first**: `[a^b]`

`\`	Escape character (GNU tools)	`\\`
`.`	Literal dot — not any-char here	Just write `.` normally
`*`	Literal asterisk — not quantifier here	Just write `*` normally

```
# Include a literal ] by placing it first
grep '[]abc]' file.txt       # matches ], a, b, or c

# Include a literal - by placing it last
grep '[a-z0-9-]' file.txt    # matches lowercase, digit, or hyphen
# NOT [a-z0-9] followed by a range — - at the end is always literal

# Include a literal . and * (they are NOT special inside [])
grep '[.*]' file.txt          # matches . or * — both literal inside []

# Match a URL slug: lowercase, digits, hyphens only
grep -E '^[a-z0-9-]+$' slugs.txt

# Match a line containing [ or ] (the bracket chars themselves)
grep '[][]]' file.txt         # ] first (to be literal), then [ (normal liter
```

POSIX Named Character Classes

POSIX defines a set of named classes written as `[:name:]` *inside* a character class bracket: `[[:name:]]`. They are more portable than raw ranges like `[a-z]` because they correctly respect the current locale and character encoding.

`[[:alpha:]]`
≈ `[a-zA-Z]`
Any alphabetic letter (locale-aware)

`[[:digit:]]`
= `[0-9]`
Any decimal digit 0–9

`[[:alnum:]]`
≈ `[a-zA-Z0-9]`
Any letter or digit

`[[:upper:]]`
≈ `[A-Z]`

`[[:lower:]]`
≈ `[a-z]`

`[[:space:]]`
`[ \t\n\r\f\v]`

Uppercase letters	Lowercase letters	Any whitespace character

`[[:blank:]]`
`[ \t]`
Space or tab only (not newline)
	`[[:punct:]]`
	`(all punctuation)`
	Any punctuation character
		`[[:print:]]`
		`(printable + space)`
		Any printable character incl. space

`[[:graph:]]`
`(printable - space)`
Printable, excluding space
	`[[:cntrl:]]`
	`ASCII 0–31, 127`
	Control characters
		`[[:xdigit:]]`
		`[0-9A-Fa-f]`
		Hexadecimal digits

```
# Trim leading whitespace (any whitespace, not just spaces)
sed 's/^[[:space:]]*//' file.txt

# Remove all punctuation from a line
sed 's/[[:punct:]]//g' text.txt

# Keep only alphanumeric characters and spaces
sed 's/[^[:alnum:][:space:]]//g' text.txt

# Match lines that start with an uppercase letter
grep '^[[:upper:]]' sentences.txt

# Find lines containing control characters (e.g. stray \r from Windows)
grep -P '[[:cntrl:]]' file.txt

# Remove Windows carriage returns (\r) at end of lines
sed 's/[[:cntrl:]]//g' file.txt

# Match a hex colour: # followed by exactly 6 hex digits
grep -E '#[[:xdigit:]]{6}' styles.css

# Combining POSIX classes: letter OR digit OR underscore (like \w)
grep -E '[[:alnum:]_]+' identifiers.txt
```

POSIX classes can be combined and negated

```
# Match anything that is NOT a letter or digit
grep -oE '[^[:alnum:]]' <<< "hello, world!"
,

!

# Combine multiple POSIX classes in one set
# Match letter, digit, underscore, or hyphen (valid identifier + hyphen)
grep -E '^[[:alnum:]_-]+$' slugs.txt

# Match a line with at least one uppercase AND contains a digit
# (two separate greps piped — one class can't express AND logic)
grep '[[:upper:]]' file.txt | grep '[[:digit:]]'
```

GNU Shorthand Classes

GNU tools (grep, sed, awk on Linux) support shorthand escape sequences as convenient aliases for common POSIX classes. These are not available in POSIX BRE/ERE or on BSD/macOS tools.

\w \W Word char [[:alnum:]_]	\d \D Digit [[:digit:]] GNU sed 4.9+	\s \S Whitespace [[:space:]]	\h \H Horiz. space [[:blank:]] PCRE/grep -P

```
# \w — word character [a-zA-Z0-9_]  (GNU grep, GNU sed)
grep -oE '\w+' <<< "hello_world 2026"
hello_world
2026

# \W — non-word character
grep -oE '\W+' <<< "hello, world!"
,
!
```

```
# \s — whitespace (GNU grep -E)
grep -E '\s{2,}' file.txt     # lines with 2+ consecutive whitespace chars

# Collapse multiple spaces to one (GNU sed)
sed 's/\s\+/ /g' file.txt

# In [[ =~ ]] — \w works in Bash ERE
[[ "hello_123" =~ ^\w+$ ]] && echo "valid identifier"
```

Locale and Portability

Pattern	GNU/Linux (UTF-8)	C/POSIX locale	macOS (BSD)	Portable?
`[a-z]`	May include accented letters	a–z ASCII only	Locale-dependent	No
`[[:lower:]]`	Lowercase for locale	a–z ASCII	Correct	Yes
`\w`	GNU tools	GNU tools	Not in BSD grep/sed	No
`[[:alnum:]_]`	All tools	All tools	All tools	Yes
`[0-9]`	Always digits only	Always digits	Always digits	Yes (digits are safe)

```
# Force C locale for predictable ASCII range behaviour
LC_ALL=C grep '[a-z]' file.txt

# Portable alternative: POSIX class (no LC_ALL needed)
grep '[[:lower:]]' file.txt
```

Practical Recipes

Validate common input formats

```
# Is this a valid username? (letters, digits, underscore, 3-16 chars)
[[ "$username" =~ ^[[:alnum:]_]{3,16}$ ]] && echo "valid"

# Is this a valid hex colour? (#rgb or #rrggbb)
[[ "$col" =~ ^#([[:xdigit:]]{3}|[[:xdigit:]]{6})$ ]] && echo "valid colour"

# Is this a MAC address? (xx:xx:xx:xx:xx:xx)
[[ "$mac" =~ ^([[:xdigit:]]{2}:){5}[[:xdigit:]]{2}$ ]] && echo "valid MAC"

# Does the password contain at least one uppercase, one digit?
[[ "$pw" =~ [[:upper:]] ]] && [[ "$pw" =~ [[:digit:]] ]] && echo "strong enou
```

Text cleaning pipelines

```
# Remove all non-printable characters from a file
sed 's/[^[:print:]]//g' dirty.txt

# Strip all digits from a string
sed 's/[[:digit:]]//g' <<< "order123 total456"
order total

# Extract only the alphabetic words from a line
grep -oE '[[:alpha:]]+' <<< "error: code 42 at line 7"
error
code
at
line

# Convert to lowercase using tr (not regex, but pairs well)
echo "HELLO WORLD" | tr '[[:upper:]]' '[[:lower:]]'
hello world

# Find lines containing tabs (useful to spot mixed indentation)
grep -P '\t' source.py          # PCRE tab
grep '[[:blank:]]' source.py    # space OR tab — broader
```

Quick Reference — Chapter 4

CHARACTER CLASS SYNTAX

`[abc]`	Match one character: a, b, or c
`[a-z]`	Match one character in the range a to z
`[^abc]`	Match any one character that is NOT a, b, or c
`[a-zA-Z0-9]`	Multiple ranges combined in one class
`[]abc]`	Literal] — put it first inside the class
`[abc-]`	Literal - — put it first or last

ESSENTIAL POSIX CLASSES

`[[:alpha:]] [[:digit:]]`	Letters / digits — locale-aware and portable
`[[:alnum:]] [[:space:]]`	Letters+digits / any whitespace character
`[[:upper:]] [[:lower:]]`	Upper / lowercase letters
`[[:blank:]] [[:punct:]]`	Space+tab only / punctuation characters
`[[:xdigit:]] [[:print:]]`	Hex digits / all printable characters
`[^[:alpha:]]`	Negated POSIX class — not a letter

GNU SHORTHAND (LINUX ONLY)

`\w / \W`	Word char [[:alnum:]_] / non-word char
`\s / \S`	Whitespace [[:space:]] / non-whitespace
`\d / \D`	Digit [[:digit:]] / non-digit (GNU sed 4.9+, grep -P)

What is coming next: Chapter 5 covers quantifiers — how to specify how many times a character or class must appear: `*` (zero or more), `+` (one or more), `?` (optional), and `{n,m}` (exact counts). Combining quantifiers with character classes is where regex becomes truly powerful.

CHAPTER 5

Quantifiers

What Are Quantifiers?

A **quantifier** tells the regex engine how many times the preceding element — a literal character, a dot, or a character class — must appear for a match to succeed. Without a quantifier, every element matches exactly once. Quantifiers unlock patterns like "one or more digits" or "between 2 and 5 letters".

*

Zero or more

The preceding element may appear any number of times — including zero. Always matches (even empty string). `BRE: * (same)`

+

One or more

The preceding element must appear at least once. Fails if there is no match at all. `BRE: \+ (GNU) or [x][x]* workaround`

?

Zero or one (optional)

The preceding element is optional — it may appear once or not at all. Used for optional parts of a pattern. `BRE: \? (GNU) or (x|) workaround`

{n}

Exactly n times

The preceding element must appear exactly *n* times. No more, no less. `BRE: \{n\}`

{n,}

At least n times

The preceding element must appear *n* or more times. No upper limit. `BRE: \{n,\}`

{n,m}

Between n and m times

The preceding element must appear at least *n* and at most *m* times. Both bounds inclusive. `BRE: \{n,m\}`

Quantifiers apply to the single element immediately to their left. To quantify a group of characters, wrap them in parentheses (covered in Chapter 6): `(abc)+` matches one or more repetitions of *abc*.

* — Zero or More

`*` is the oldest and most common quantifier. It matches zero or more occurrences of the preceding element. Because it accepts zero occurrences, it *always* produces a match — even on an empty string or one where the element is absent.

```
# ab*c — 'b' appears zero or more times
echo -e "ac\nabc\nabbc\nabbbc\nxc" | grep 'ab*c'
ac      # zero b's
abc     # one b
abbc    # two b's
abbbc   # three b's
# "xc" has no match — 'a' is required (not quantified)

# .* — match any number of any characters (the wildcard pattern)
grep 'start.*end' file.txt      # "start" then anything then "end"

# [0-9]* — zero or more digits (matches empty string too!)
echo "abc" | grep -oE '[0-9]*'
           # matches empty string before 'a', before 'b', before 'c', after '
# This is the "zero or more matches empty" gotcha — see below
```

The zero-or-more trap. `[0-9]*` matches the empty string, so `grep -oE '[0-9]*'` on the string `abc` will match four empty strings — one between every character. This is almost never what you want. Use `+` (one or more) when you require at least one match.

+ — One or More

`+` requires at least one occurrence. It is the version of `*` you should reach for by default when you know the element must be present. In ERE (`grep -E`, `sed -E`, awk, `[[ =~ ]]`) write `+`; in BRE write `\+` (GNU extension).

```
# [0-9]+ — one or more digits (no empty-string problem)
echo "order 42 items, total 199 units" | grep -oE '[0-9]+'
42
199
```

```
# \w+ — one or more word characters (GNU grep)
grep -oE '\w+' <<< "hello_world 2026"
hello_world
2026

# [[:alpha:]]+ — one or more letters (POSIX, portable)
grep -oE '[[:alpha:]]+' <<< "error: code 42"
error
code

# BRE equivalent (GNU extension — note \+)
grep '[0-9]\+' data.txt

# Validate non-empty input in a script
[[ -z "$input" ]] && echo "empty!" || echo "has content"
# or with regex — must contain at least one non-space char:
[[ "$input" =~ [^[:space:]]+ ]] && echo "not blank"
```

? — Zero or One (Optional)

`?` makes the preceding element optional — it may appear once or not at all. It is perfect for handling optional characters like a sign, a delimiter, or a variant spelling. In BRE write `\?` (GNU extension).

```
# colou?r — the 'u' is optional
echo -e "colour\ncolor\ncoluur" | grep -E 'colou?r'
colour
color
# "coluur" has no match — only ONE optional u is allowed

# Match an optional leading minus sign on a number
grep -oE '-?[0-9]+' <<< "temps: -5 0 23 -18"
-5
0
23
-18
```

```
# https? — match http or https
grep -E 'https?://' urls.txt

# Match dates with optional separator: 20260611 or 2026-06-11 or 2026/06/11
grep -E '[0-9]{4}[-/]?[0-9]{2}[-/]?[0-9]{2}' dates.txt

# BRE version (GNU sed)
sed -n '/https\?:\/\//p' urls.txt   # \? in BRE
```

```
PATTERN: -?[0-9]+ APPLIED TO VARIOUS INPUTS
-42 ✓ optional minus present, digits follow 42 ✓ optional minus absent, digits follow 0 ✓ zero is a valid digit - ✗ minus present but no digits follow (+ requires at least one) abc ✗ no digits at all
```

{n}, {n,}, {n,m} — Exact and Bounded Counts

Curly-brace quantifiers give you precise control over repetition counts. They are indispensable for matching fixed-length formats like dates, postal codes, credit card numbers, and phone numbers.

```
# {n} — exactly n occurrences
grep -E '[0-9]{4}' data.txt        # exactly 4 consecutive digits
grep -E '[A-Z]{3}' text.txt        # exactly 3 uppercase letters
grep -E '^.{80}$' file.txt         # lines that are exactly 80 characters

# {n,} — n or more occurrences
grep -E '[0-9]{3,}' data.txt       # 3 or more consecutive digits
grep -E '[[:alpha:]]{8,}' words.txt # words with 8 or more letters

# {n,m} — between n and m occurrences (inclusive)
grep -E '[0-9]{2,4}' data.txt      # 2, 3, or 4 consecutive digits
grep -E '[[:alpha:]]{4,8}' words.txt# words between 4 and 8 letters long

# Practical: match a 4-digit year
grep -E '\b[0-9]{4}\b' text.txt

# Practical: match a UK postcode pattern (simplified: A9 9AA or A99 9AA)
```

```
grep -E '[A-Z]{1,2}[0-9]{1,2} [0-9][A-Z]{2}' addresses.txt

# Practical: match lines longer than 120 characters
grep -E '^.{121}' source.py          # 121+ chars means line exceeds 120
```

BRE requires backslashes around braces

BRE — DEFAULT GREP / SED

```
[0-9]\{4\} # exactly 4 [0-9]\{3,\} # 3 or more [0-9]\{2,4\} # 2 to 4 x\+ # one or more (GNU ext) x\? # zero or one (GNU ext) \(abc\) # group (see Ch 6)
```

ERE — GREP -E / SED -E / AWK

```
[0-9]{4} # exactly 4 [0-9]{3,} # 3 or more [0-9]{2,4} # 2 to 4 x+ # one or more x? # zero or one (abc) # group (see Ch 6)
```

Greedy Matching — The Default Behaviour

All quantifiers in BRE and ERE are **greedy** — they consume as many characters as possible while still allowing the overall pattern to succeed. The engine tries the longest possible match first, then backtracks if needed.

PATTERN: <.*> ON INPUT: <B>BOLD</B> AND <I>ITALIC</I>

Engine attempt: `<b>bold</b> and <i>italic</i>` .* consumed everything from <b> to the last > Result: ONE match — the entire string Pattern: <[^>]*> on same input (negated class workaround): Match 1: `<b>` — stops at first > Match 2: `</b>` Match 3: `<i>` Match 4: `</i>`

```
# Greedy: .* swallows as much as possible
echo '[one] and [two] and [three]' | grep -oE '\[.*\]'
[one] and [two] and [three]   # one giant match

# Fix with negated class — stop at the first ]
echo '[one] and [two] and [three]' | grep -oE '\[[^]]*\]'
[one]
[two]
```

```
[three]   # three separate matches

# Greedy with + on digits
echo "abc12345def" | grep -oE '[0-9]+'
12345   # + is greedy — matches all 5 digits as one match, not 5 single ones

# Greedy with {2,4}: takes 4 when it can
echo "123456" | grep -oE '[0-9]{2,4}'
1234   # takes 4 (the maximum) first, leaving "56" for the next match
56
```

No lazy quantifiers in BRE/ERE — the workaround

Most modern regex flavours have a *lazy* (non-greedy) quantifier: `*?`, `+?`, `??`. These match as *few* characters as possible. **BRE and ERE do not have lazy quantifiers.** The standard workaround is to replace `.*` with a negated character class that stops at the boundary:

Greedy (wrong)	Intended behaviour	BRE/ERE workaround
`<.*>`	Each HTML tag individually	`<[^>]*>`
`".*"`	Each quoted string individually	`"[^"]*"`
`\(.*\)`	Each parenthesised group	`\([^)]*\)`
`\[.*\]`	Each bracketed item	`\[[^]]*\]`
`start.*end`	Shortest start…end span	`start[^e]*end` (if safe)

> **grep -P has lazy quantifiers.** If you genuinely need non-greedy matching and negated classes won't work, `grep -P` (PCRE) supports `.*?` and `.+?`. This is covered fully in Chapter 11.

Quantifier Portability

Quantifier	ERE meaning	grep (BRE)	grep -E	sed (BRE)	sed -E	awk	[[=~]]
`*`	Zero or more	✓	✓	✓	✓	✓	✓
`+`	One or more	\+ GNU	✓	\+ GNU	✓	✓	✓
`?`	Zero or one	\? GNU	✓	\? GNU	✓	✓	✓
`{n,m}`	n to m times	\{n,m\}	✓	\{n,m\}	✓	✓	✓
`*?` `+?`	Lazy (non-greedy)	✗	✗	✗	✗	✗	✗

Practical Quantifier Recipes

Matching structured formats

```
# ISO date: YYYY-MM-DD
grep -E '\b[0-9]{4}-[0-9]{2}-[0-9]{2}\b' file.txt

# Time HH:MM or HH:MM:SS
grep -E '\b[0-9]{2}:[0-9]{2}(:[0-9]{2})?\b' log.txt

# IPv4 address (simplified — allows invalid ranges like 999.999.9.9)
grep -E '\b[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\b' file.txt

# Email address (simplified)
grep -E '[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}' contacts.txt

# Version string: 1 to 3 dot-separated number groups
grep -E '\b[0-9]+(\.[0-9]+){1,2}\b' changelog.txt
```

Using quantifiers in sed substitutions

```
# Replace any run of whitespace with a single space
sed -E 's/[[:space:]]+/ /g' file.txt

# Remove all standalone numbers (whole words that are digits)
sed -E 's/\b[0-9]+\b//g' text.txt

# Truncate lines longer than 80 characters (keep first 80)
sed -E 's/^(.{80}).*/\1/' file.txt   # capture 80 chars, drop the rest

# Normalise multiple blank lines to one blank line
cat -s file.txt   # cat -s squeezes blank lines (easiest way)
# or with sed:
sed '/./,/^$/!d' file.txt
```

Validation functions using quantifiers

```
# Is this a valid port number (1-65535)?
is_port() {
    [[ "$1" =~ ^[0-9]{1,5}$ ]] && [[ "$1" -ge 1 ]] && [[ "$1" -le 65535 ]]
}
# regex checks format; arithmetic checks range

# Is this a valid IPv4 octet (0-255)?
is_octet() {
    [[ "$1" =~ ^[0-9]{1,3}$ ]] && [[ "$1" -le 255 ]]
}

# Does this filename have a valid extension (2-4 alpha chars)?
[[ "$file" =~ \.[[:alpha:]]{2,4}$ ]] && echo "has extension"
```

Common Quantifier Mistakes

Mistake	Pattern	Problem	Fix

Using * when + is needed	`[0-9]*`	Matches empty string — passes validation even with no digits	`[0-9]+`
Greedy .* consuming too much	`<.*>`	Matches from first `<` to last `>`	`<[^>]*>`
Missing anchors with {n}	`[0-9]{4}`	Matches any 4 digits anywhere — including inside "123456"	`\b[0-9]{4}\b` or `^[0-9]{4}$`
BRE braces without backslash	`grep '[0-9]{4}'`	In BRE, `{4}` is literal — matches the characters `{`, `4`, `}`	`grep '[0-9]\{4\}'` or `grep -E '[0-9]{4}'`
+ or ? unescaped in BRE	`grep 'ab+c'`	In BRE, `+` is a literal plus sign — matches `ab+c`	`grep 'ab\+c'` or `grep -E 'ab+c'`
{n,m} with no comma for range	`[0-9]{3}`	Matches exactly 3 — if you want "3 to 5" you need a comma: `{3,5}`	`[0-9]{3,5}`

Quick Reference — Chapter 5

QUANTIFIERS — ERE SYNTAX (GREP -E, SED -E, AWK, [[=~]])

`x*`	Zero or more of x — always matches (even empty)
`x+`	One or more of x — requires at least one occurrence
`x?`	Zero or one of x — makes x optional
`x{n}`	Exactly n occurrences of x
`x{n,}`	At least n occurrences of x
`x{n,m}`	Between n and m occurrences of x (inclusive)

BRE EQUIVALENTS (DEFAULT GREP / SED)

`x\+ x\?`	One or more / zero or one (GNU extension)
`x\{n\} x\{n,m\}`	Exact and bounded counts (backslashes required)

GREEDY WORKAROUND

`[^delimiter]*`

Match up to (not including) the delimiter — the BRE/ERE lazy substitute

`grep -P '.*?'`

True lazy/non-greedy quantifier — PCRE only (Chapter 11)

What is coming next: Chapter 6 covers grouping and alternation — using `(...)` to treat multiple characters as a single unit for quantifiers, and `|` to match one of several alternatives. It also covers backreferences: capturing what a group matched and reusing it in the pattern or the replacement string.

CHAPTER 6

Groups, Alternation, and Back-references

Grouping with Parentheses

A **group** is a portion of a regex wrapped in parentheses `(...)`. Groups serve two purposes: they allow a quantifier to apply to more than one character at a time, and they *capture* the text matched by the group so it can be reused later.

```
WHAT PARENTHESES DO — THREE ROLES

(ab)+   → quantify: "ab" one or more times as a unit

(cat|dog)   → alternate: match "cat" OR "dog"

([0-9]+)-([0-9]+)   → capture: store matched text in \1 and \2
```

Groups as a unit for quantifiers

Without a group, a quantifier applies only to the immediately preceding character or class. With a group, it applies to everything inside the parentheses.

```
# Without grouping: + applies only to 'p'
echo -e "hap\nhapp\nhappp\nhappy" | grep -E 'hap+'
hap
happ
happp
happy   # all match — + only requires one or more 'p'

# With grouping: + applies to the whole "ha" unit
echo -e "ha\nhaha\nhahaha\nhello" | grep -E '(ha)+'
ha
haha
hahaha
# "hello" has no match — does not contain "ha"
```

```
# Repeated two-character pattern
grep -E '([0-9]{1,3}\.){3}[0-9]{1,3}' file.txt   # IPv4: three "NNN." groups

# Optional suffix as a group
grep -E 'colou(r|rs|red|ring)' text.txt
# cleaner: "colour" optionally followed by a suffix group
grep -E 'colour(s|ed|ing)?' text.txt
```

BRE requires backslashes around parentheses

```
# ERE: ( ) are grouping metacharacters
grep -E '(foo)+' file.txt

# BRE: ( ) are LITERALS — you need \( \) for grouping
grep '\(foo\)\+' file.txt

# BRE: a bare ( ) matches the literal characters ( and )
grep '(foo)' file.txt   # matches the string "(foo)" literally!
```

Alternation with |

The pipe character `|` means **OR** — the pattern matches if either the left side or the right side matches. It has the lowest precedence of all regex operators: everything to the left of `|` is one alternative, everything to the right is another. Use groups to limit the scope of alternation.

```
# Match "cat" or "dog"
grep -E 'cat|dog' file.txt

# Match "ERROR" or "FATAL" or "CRITICAL" log levels
grep -E 'ERROR|FATAL|CRITICAL' app.log

# Alternation without grouping: ^ applies only to left alternative
grep -E '^cat|dog' file.txt
# Matches: lines STARTING with "cat"  OR  lines containing "dog" anywhere
# NOT: lines starting with "cat" or "dog"
```

```
# Fix: use a group to limit the alternation scope
grep -E '^(cat|dog)' file.txt
# NOW: lines starting with "cat" OR starting with "dog"
```

```
^CAT|DOG VS ^(CAT|DOG) — SCOPE MATTERS
cat sat here → ^cat|dog ✓ (starts with cat) the dog barked → ^cat|dog ✓
(contains dog) cat sat here → ^(cat|dog) ✓ (starts with cat) the dog barked →
^(cat|dog) ✗ (starts with "the", not cat or dog) dog sat here → ^(cat|dog) ✓
(starts with dog)
```

```
# Alternation inside a larger pattern
grep -E '(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) [0-9]{1,2}' dates.

# Match http or https or ftp URLs
grep -E '(https?|ftp)://[^[:space:]]+' file.txt

# Match common image extensions
grep -E '\.(jpg|jpeg|png|gif|webp|svg)$' filelist.txt

# BRE alternation uses \| (GNU extension — not POSIX)
grep 'cat\|dog' file.txt
# On strictly POSIX systems without GNU grep, alternation requires -E
```

Capturing Groups and Backreferences

Every group enclosed in `(...)` — or `\(...\)` in BRE — is a **capturing group**. The regex engine stores the text matched by each group and makes it available as a numbered backreference: `\1` for the first group, `\2` for the second, and so on up to `\9`.

```
PATTERN: ([0-9]{4})-([0-9]{2})-([0-9]{2}) ON "2026-06-11"
Group \1 captures: 2026 ← matched by ([0-9]{4}) Group \2 captures: 06 ←
matched by ([0-9]{2}) Group \3 captures: 11 ← matched by ([0-9]{2})
Replacement \3/\2/\1 → 11/06/2026
```

Backreferences in sed replacements

```
# Reformat ISO date YYYY-MM-DD → DD/MM/YYYY
sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/' <<< "2026-06-11"
11/06/2026

# Swap first and last name (comma-separated)
sed -E 's/([^,]+), ([^,]+)/\2 \1/' <<< "Smith, John"
John Smith

# Wrap every number in brackets
sed -E 's/([0-9]+)/[\1]/g' <<< "port 8080 timeout 30"
port [8080] timeout [30]

# Extract just the filename from a full path
sed -E 's|.*/([^/]+)$|\1|' <<< "/home/user/documents/report.pdf"
report.pdf

# Surround a key=value pair: key → "key"
sed -E 's/^([^=]+)=/"\1"=/' <<< "username=alice"
"username"=alice

# BRE version of the date reformat (backslashes on groups)
sed 's/\([0-9]\{4\}\)-\([0-9]\{2\}\)-\([0-9]\{2\}\)/\3\/\2\/\1/' <<< "2026-06
11/06/2026
```

```
HOW BACKREFERENCES FLOW FROM PATTERN TO REPLACEMENT
Pattern: s/([A-Z]+)_([0-9]+)/\2_\1/ Input: CODE_42 Group 1 \1 captured: CODE
Group 2 \2 captured: 42 Output: 42_CODE
```

Backreferences in patterns — matching repeated text

Backreferences can also appear inside the *pattern itself* (not just the replacement). This forces the engine to match the same text a second time.

```
# Match a line where the same word appears twice consecutively
echo -e "the the cat\nthe cat\nhello hello world" | grep -E '\b(\w+) \1\b'
```

```
the the cat
hello hello world

# Find duplicate words and remove them
sed -E 's/\b(\w+) \1\b/\1/g' <<< "the the quick brown fox fox"
the quick brown fox

# Match a repeated character (e.g. doubled letters)
grep -E '([a-z])\1' words.txt   # words with a doubled letter: "book", "free"

# Match a string that starts and ends with the same character
grep -E '^(.).*\1$' words.txt   # e.g. "radar", "level", "noon"

# Match an HTML tag and its matching closing tag (simplified)
grep -E '<([a-z]+)>.*</\1>' index.html   # <b>...</b>, <em>...</em>
```

Backreferences in Bash [[=~]] with BASH_REMATCH

When you use `[[ =~ ]]` in Bash, the shell populates the `BASH_REMATCH` array with the captured groups. `BASH_REMATCH[0]` holds the entire match; `BASH_REMATCH[1]` holds group 1; `BASH_REMATCH[2]` holds group 2; and so on.

```
# Extract year, month, day from a date string
date_str="2026-06-11"
if [[ "$date_str" =~ ^([0-9]{4})-([0-9]{2})-([0-9]{2})$ ]]; then
    echo "Full match : ${BASH_REMATCH[0]}"
    echo "Year       : ${BASH_REMATCH[1]}"
    echo "Month      : ${BASH_REMATCH[2]}"
    echo "Day        : ${BASH_REMATCH[3]}"
fi
Full match : 2026-06-11
Year       : 2026
Month      : 06
Day        : 11

# Parse a key=value pair
line="max_connections=100"
[[ "$line" =~ ^([^=]+)=(.+)$ ]] && {
```

```
    echo "Key:   ${BASH_REMATCH[1]}"
    echo "Value: ${BASH_REMATCH[2]}"
}
Key:   max_connections
Value: 100

# Extract protocol and host from a URL
url="https://api.example.com/v2/users"
[[ "$url" =~ ^(https?)://([^/]+)(/.*)$ ]] && {
    echo "Protocol : ${BASH_REMATCH[1]}"
    echo "Host     : ${BASH_REMATCH[2]}"
    echo "Path     : ${BASH_REMATCH[3]}"
}
Protocol : https
Host     : api.example.com
Path     : /v2/users
```

BASH_REMATCH quoting rule. The regex pattern in `[[ =~ ]]` must *not* be quoted — quoting it forces a literal string comparison instead of regex matching. Store the pattern in a variable if it contains characters that the shell might misinterpret: `pat='^[0-9]+'` then `[[ "$str" =~ $pat ]]`.

Nested and Numbered Groups

Groups are numbered left to right by the position of their opening parenthesis. Nested groups count from the outermost in.

```
# Groups numbered by opening parenthesis, left to right
# Pattern: ((a)(b)) — three groups
#           ^  ^  ^
#           1  2  3
str="ab"
[[ "$str" =~ ((a)(b)) ]]
echo "${BASH_REMATCH[0]}"  # ab  — full match
echo "${BASH_REMATCH[1]}"  # ab  — group 1: outer (ab)
echo "${BASH_REMATCH[2]}"  # a   — group 2: inner (a)
echo "${BASH_REMATCH[3]}"  # b   — group 3: inner (b)
```

```
# Practical nested groups: match a version like "v1.2.3"
ver="v1.2.3"
[[ "$ver" =~ ^v([0-9]+)(\.([0-9]+))?(\.([0-9]+))?$ ]]
# Group 1: major (1)   Group 3: minor (2)   Group 5: patch (3)
echo "major=${BASH_REMATCH[1]} minor=${BASH_REMATCH[3]} patch=${BASH_REMATCH[
major=1 minor=2 patch=3
```

Non-Capturing Groups (?:...)

Sometimes you need a group for alternation or quantification but don't want it to capture — perhaps because you are already using `\1` – `\9` for other groups and don't want to waste a slot. Non-capturing groups use `(?:...)` syntax. They work in **ERE and PCRE** but are **not available in BRE**.

```
# Capturing group wastes \1 on the protocol — we only care about the host
sed -E 's|(https?://)([^/]+)|\2|' <<< "https://example.com/path"
example.com/path   # \2 is the host

# Non-capturing group: (?:https?://) groups but doesn't capture
# so the host becomes \1 instead of \2
sed -E 's|(?:https?://)([^/]+)|\1|' <<< "https://example.com/path"
example.com/path   # \1 is now the host — cleaner numbering

# Non-capturing group for alternation without wasting a capture slot
grep -E '(?:cat|dog) food' file.txt   # group needed for alternation, not cap

# In [[ =~ ]] — non-capturing groups work fine
[[ "$url" =~ ^(?:https?|ftp)://(.+)$ ]]
echo "rest of URL: ${BASH_REMATCH[1]}"   # \1 is the path, not the protocol
```

> **Non-capturing groups are NOT supported in BRE.** In default `grep` or `sed` (without `-E`), writing `(?:...)` will either error or match the literal characters `(?:`. Always use `-E` when using non-capturing groups.

The & Replacement Special

In sed replacements, `&` (ampersand) refers to the entire matched string — the full match, not just a captured group. It is a shorthand for "whatever was matched".

```
# Wrap the entire match in brackets
sed -E 's/[0-9]+/[&]/g' <<< "order 42 total 199"
order [42] total [199]

# Quote every word
sed -E 's/[[:alpha:]]+/"&"/g' <<< "hello world"
"hello" "world"

# Prepend a label to every matched IP address
sed -E 's/[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}/IP:&/g' log.txt

# & is equivalent to \0 (the whole match) in some tools
# In sed: & means the whole match; \1 means group 1
sed -E 's/([0-9]+)/num=\1 raw=&/' <<< "value 42"
value num=42 raw=42   # \1 and & are the same here (only one group)
```

BRE vs ERE — Grouping and Alternation Summary

Feature	ERE (grep -E, sed -E, awk)	BRE (grep, sed default)	PCRE (grep -P)
Capturing group	`(...)`	`\(...\)`	`(...)`
Non-capturing group	`(?:...)`	Not supported	`(?:...)`
Alternation	`|`	`|` (GNU only)	`|`
Backreference in pattern	`\1` – `\9`	`\1` – `\9`	`\1` – `\9`
Backreference in replacement	`\1` – `\9`	`\1` – `\9`	`\1` – `\9` or `$1`

Named groups	Not supported	Not supported	`(?P<name>...)`
Whole-match reference	`&` in sed replacement	`&` in sed replacement	`&` / `\0`

Practical Real-World Recipes

Log and data extraction

```
# Extract the HTTP status code from an Apache log line
sed -E 's/.*" ([0-9]{3}) .*/\1/' access.log

# Extract timestamp and log level from structured log
# Input: [2026-06-11 14:32:05] ERROR Something went wrong
[[ "$line" =~ ^\[([0-9-]+ [0-9:]+)\] ([A-Z]+) (.+)$ ]] && {
    echo "time=${BASH_REMATCH[1]} level=${BASH_REMATCH[2]}"
    echo "msg=${BASH_REMATCH[3]}"
}

# Reformat CSV date column: MM/DD/YYYY → YYYY-MM-DD
sed -E 's|([0-9]{2})/([0-9]{2})/([0-9]{4})|\3-\1-\2|g' report.csv
```

Code manipulation

```
# Rename a function: getUser → fetchUser (whole word, all files)
sed -i -E 's/\bgetUser\b/fetchUser/g' src/*.js

# Convert snake_case identifiers to camelCase
# Approach: loop sed until no more underscores-with-next-letter remain
echo "get_user_name" | sed -E 's/_([a-z])/\u\1/g'
getUserName   # \u uppercases the next char (GNU sed extension)

# Add quotes around all unquoted values in key=value format
sed -E 's/^([^=]+)=([^"].*)$/\1="\2"/' config.env
```

Parsing shell script output into variables

```
parse_df_line() {
    # Parse: /dev/sda1   50G   20G   30G  40%  /home
    local line="$1"
    if [[ "$line" =~ ^([^[:space:]]+)[[:space:]]+[^[:space:]]+[[:space:]]+[^[
        echo "device=${BASH_REMATCH[1]}"
        echo "use%=${BASH_REMATCH[2]}"
        echo "mount=${BASH_REMATCH[3]}"
    fi
}
# Alert if any filesystem is over 90% used
df -h | grep -E '([0-9]+)%' | while read -r line; do
    [[ "$line" =~ ([0-9]+)% ]] && [[ "${BASH_REMATCH[1]}" -gt 90 ]] &&
        echo "WARNING: $line"
done
```

Quick Reference — Chapter 6

GROUPING

`(abc)`

Capturing group — ERE, PCRE; quantify and capture

`\(abc\)` Capturing group — BRE (backslashes required)

`(?:abc)`

Non-capturing group — ERE/PCRE only; no capture slot used

`(a|b)+` Quantify a group — one or more of "a" or "b"

ALTERNATION

`cat|dog` Match "cat" OR "dog" — ERE/PCRE

`^(cat|dog)`

Line starts with "cat" or "dog" — group limits scope of |

`cat|dog`	BRE alternation — GNU extension only

BACKREFERENCES

`\1 \2 … \9`

Refer to captured group 1–9 (in pattern or sed replacement)

`&`	Entire match (in sed replacement)
`BASH_REMATCH[0]`	Full match from [[=~]] test
`BASH_REMATCH[n]`	Group n from [[=~]] test (n = 1, 2, 3 …)

What is coming next: Chapter 7 dives into `grep` and `egrep` in depth — practical flags (`-o` , `-n` , `-l` , `-c` , `-r` , `-v`), combining flags for real-world log analysis, the difference between `-E` , `-F` , and `-P` , and building multi-pattern pipelines for efficient text filtering.

CHAPTER 7

grep and egrep in Depth

grep at a Glance

`grep` — *Global Regular Expression Print* — is the primary regex tool at the Bash command line. It reads lines from files or stdin, tests each line against a pattern, and by default prints lines that match. It is fast, composable, and available on every Unix system.

Understanding grep's flags is what separates one-line lookups from powerful search pipelines. This chapter covers every flag you will reach for regularly, how to combine them, and when to switch between the three grep modes: BRE, ERE, and PCRE.

The Three grep Modes

`grep` / `grep -G`

BRE — Basic Regex

Default mode. `+` `?` `(` `|` are literals; need `\+` `\?` `\(` `\|`. Portable to all Unix systems.

`grep -E` / `egrep`

ERE — Extended Regex

Cleaner syntax. `+` `?` `(` `|` work without backslashes. Recommended for everyday use.

`grep -P`

PCRE — Perl Compatible

Full Perl regex: lookahead, lookbehind, lazy `.*?`, named groups. GNU grep only — not on macOS BSD grep.

```
# BRE: one-or-more needs \+
grep '[0-9]\+' file.txt

# ERE: cleaner — + works directly
grep -E '[0-9]+' file.txt
egrep '[0-9]+' file.txt   # egrep is an alias for grep -E

# PCRE: lazy quantifier — not available in BRE/ERE
grep -P '<.+?>' file.html
```

> `egrep` **is deprecated** in POSIX but still widely available as a symlink to `grep -E`. Prefer `grep -E` in scripts for clarity and portability.

Essential Output Flags

`-o`
Only matching
Print only the matched text, not the whole line. Each match on its own line.

`-n`
Line numbers
Prefix each matching line with its line number in the file.

`-c`
Count
Print a count of matching lines instead of the lines themselves.

`-l`
Files with matches
Print only the filename of files that contain at least one match.

`-L`
Files without matches
Print only filenames of files that contain NO matches — the inverse of `-l`.

`-v`
Invert match
Print lines that do NOT match the pattern — every non-matching line.

`-i`
Case insensitive
Match regardless of letter case. `ERROR`, `Error`, `error` all match `'error'`.

`-w`
Whole word
Match only whole words — equivalent to wrapping the pattern in `\b...\b`.

`-x`
Whole line
Match only lines where the entire line matches — equivalent to `^pattern$`.

```
INPUT: THREE LINES — "ERROR: PORT 8080", "WARNING: LOW MEMORY", "ERROR: DISK FULL"
grep 'error' → error: port 8080 / error: disk full (whole matching lines) grep -o 'error' → error / error (just the matched text) grep -c 'error' → 2 (count of matching lines) grep -n 'error' → 1:error: port 8080 / 3:error: disk full grep -v 'error' → warning: low memory (non-matching line)
```

```
# -o: extract only the matched portion — ideal for data extraction
echo "order #12345 placed on 2026-06-11" | grep -oE '[0-9]{4}-[0-9]{2}-[0-9]{
```

```
2026-06-11

# -o with multiple matches per line: each on its own line
echo "call 555-1234 or 555-5678" | grep -oE '[0-9]{3}-[0-9]{4}'
555-1234
555-5678

# -c: count matching lines (not individual matches)
grep -c 'ERROR' app.log
47

# -l: which files in a directory contain a pattern?
grep -rl 'TODO' src/

# -L: which files do NOT contain a shebang?
grep -rL '^#!' bin/

# -w: whole word — "error" won't match "errors" or "no_error_code"
grep -w 'error' app.log

# -x: only lines that are EXACTLY this string
grep -x 'done' status.txt   # "done" alone on a line — not "done." or " done"

# -i: case-insensitive (useful for logs with mixed conventions)
grep -i 'warning' app.log    # matches WARNING, Warning, warning
```

Context Flags — Seeing Surrounding Lines

Context flags show lines before and/or after each match — invaluable when debugging log files or reading code with grep.

```
GREP -B2 -A2 'ERROR' — 2 LINES BEFORE AND AFTER EACH MATCH
14- 2026-06-11 14:31:58 INFO Processing batch 7 15- 2026-06-11 14:31:59 INFO Connecting to database 16: 2026-06-11 14:32:00 ERROR Connection timeout after 30s 17- 2026-06-11 14:32:01 INFO Retrying connection (1/3) 18- 2026-06-11 14:32:04 INFO Retrying connection (2/3) -- (separator between match groups)
```

```
# -A n: show n lines After each match
grep -A3 'ERROR' app.log

# -B n: show n lines Before each match
grep -B2 'ERROR' app.log

# -C n: show n lines of Context (before AND after)
grep -C3 'ERROR' app.log

# Context with line numbers — very useful for finding the exact source locati
grep -n -C2 'def process' module.py

# Groups separated by "--" — suppress the separator
grep -A2 --no-group-separator 'ERROR' app.log
```

File and Directory Flags

```
# -r: recursive search through directories
grep -r 'api_key' /etc/

# -R: recursive, but also follows symlinks (GNU grep)
grep -R 'TODO' src/

# --include: only search files matching a glob
grep -r --include='*.py' 'import os' .
grep -r --include='*.{js,ts}' 'console\.log' src/

# --exclude: skip files matching a glob
grep -r --exclude='*.min.js' 'function' .

# --exclude-dir: skip entire directories
grep -r --exclude-dir='.git' --exclude-dir='node_modules' 'password' .

# Combine -rl with --include for scoped file search
grep -rl --include='*.conf' 'MaxConnections' /etc/
```

```
# Search a list of files from another command
find . -name '*.log' -newer deploy.txt | xargs grep -l 'ERROR'
```

Multiple Patterns — -e and -f

```
# -e: specify multiple patterns (OR logic — line matches if any pattern matche
grep -e 'ERROR' -e 'FATAL' -e 'CRITICAL' app.log
# Equivalent to: grep -E 'ERROR|FATAL|CRITICAL' app.log

# -f: read patterns from a file (one pattern per line)
cat patterns.txt
ERROR
FATAL
CRITICAL
grep -f patterns.txt app.log

# Combine -f with other flags
grep -if patterns.txt app.log   # -i = case insensitive
grep -cf patterns.txt app.log   # -c = count matching lines

# AND logic: pipe two greps (both patterns must match)
grep 'ERROR' app.log | grep 'database'  # lines with both ERROR and database

# NOT logic: grep -v to exclude a pattern
grep 'ERROR' app.log | grep -v 'connection'  # ERROR lines NOT about connecti
```

grep -F — Fixed String (No Regex)

`grep -F` (also `fgrep`) treats the pattern as a plain string — no regex processing at all. It is significantly faster on large files because it uses the Boyer-Moore-Horspool string search algorithm instead of a regex engine.

```
# -F: safe for patterns that contain regex metacharacters
grep -F '192.168.1.1' access.log   # dots are literals, not any-char
grep -F 'a+b=c' math.txt           # + is literal
```

```
grep -F '$100.00' invoice.txt      # $ is literal
grep -F '[::1]' access.log         # brackets are literal

# -F with a variable (the safest pattern when content is unknown)
SEARCH="$USER_INPUT"
grep -F "$SEARCH" file.txt

# -F is much faster on large files — no regex compilation overhead
grep -F "Connection refused" /var/log/syslog   # faster than grep -E

# -F with -f: search for many fixed strings at once (very efficient)
grep -Ff known_ips.txt access.log   # match any of 10,000 IPs
```

grep Exit Codes — Using grep in Scripts

grep's exit code is what makes it useful inside shell scripts — not just for output, but as a conditional test.

0	1	2
Match found	No match	Error
At least one line matched the pattern	Pattern was valid but no lines matched	Bad pattern syntax or file not found

```
# Use grep as a boolean test — -q suppresses all output
if grep -q 'ERROR' app.log; then
    echo "Errors found — alerting team"
fi

# -q (quiet): suppress output, just set exit code
# This is the standard idiom for "does this file contain X?"
grep -q 'pattern' file && echo "found" || echo "not found"

# Check if a process is running
if grep -q 'nginx' <(ps aux); then
    echo "nginx is running"
fi
```

```
# Validate that a config file has a required setting
if ! grep -qE '^MaxSessions[[:space:]]' /etc/ssh/sshd_config; then
    echo "WARNING: MaxSessions not configured" >&2
fi

# Count errors and fail if threshold exceeded
errors=$(grep -c 'ERROR' app.log)
((errors > 100)) && { echo "Too many errors: $errors"; exit 1; }
```

Practical Pipeline Patterns

Log analysis

```
# Frequency count of log levels
grep -oE '(ERROR|WARN|INFO|DEBUG)' app.log | sort | uniq -c | sort -rn
    523 INFO
     47 ERROR
     12 WARN
      3 DEBUG

# Top 10 most frequent IP addresses in an access log
grep -oE '^[0-9.]+' access.log | sort | uniq -c | sort -rn | head -10

# All unique HTTP status codes in today's log
grep -oE '" [0-9]{3} ' access.log | grep -oE '[0-9]{3}' | sort -u

# Extract all unique email addresses from a mailbox dump
grep -oE '[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}' mail.txt | sort

# Show only error lines WITH their timestamp (context aware)
grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2}.*ERROR' app.log
```

Code search

```
# Find all TODO/FIXME/HACK comments in a codebase
grep -rn --include='*.py' -E '#\s*(TODO|FIXME|HACK|XXX)' .

# Find function definitions in Python files
grep -rn --include='*.py' -E '^[[:space:]]*(def|async def) [[:alnum:]_]+' .

# Find hardcoded passwords or secrets (simple heuristic)
grep -rniE '(password|secret|api_key|token)\s*=\s*["\x27][^\s"]+["\x27]' . \
    --exclude-dir='.git' --exclude-dir='node_modules'

# Find all files that import a specific module
grep -rl --include='*.py' '^import os' .

# List all unique function names defined in a file
grep -oE '^def [[:alnum:]_]+' module.py | cut -d' ' -f2 | sort
```

Security and compliance checks

```
# Find files containing private key headers
grep -rl 'BEGIN.*PRIVATE KEY' /var/www/

# Check sshd_config for insecure settings
grep -E '^(PermitRootLogin yes|PasswordAuthentication yes)' /etc/ssh/sshd_con

# Audit: find world-writable files listed in a manifest
grep -oP '(?<=path=")[^"]+' manifest.xml | xargs ls -la | grep '^......rw'
```

grep -P Highlights — PCRE-Specific Power

When `-E` isn't enough, `grep -P` unlocks PCRE features. Full PCRE coverage is in Chapter 11; here are the patterns worth knowing now:

```
# Lazy quantifier: match each HTML tag individually
grep -oP '<.+?>' file.html
```

```
# Lookahead: match a word followed by a colon (but don't include the colon)
grep -oP '\w+(?=:)' file.txt

# Lookbehind: match digits that follow a $ sign
grep -oP '(?<=\$)[0-9]+' invoice.txt

# Named group: extract the value of a specific JSON field
grep -oP '"user":\s*"(?P<name>[^"]+)"' data.json

# Non-greedy: match content between the FIRST pair of matching delimiters
grep -oP '\[.+?\]' file.txt
```

> `grep -P` **is GNU grep only** — it is not available on macOS (BSD grep) or strictly POSIX environments. Scripts that must run on macOS should use `-E` with workarounds, or call `perl -ne` for PCRE patterns.

Flag Combinations Cheat Sheet

Combination	What it does	Typical use
`grep -q`	Silent test — exit code only	Script conditionals
`grep -c`	Count of matching lines	Metrics, thresholds
`grep -n`	Line numbers on output	Debugging, navigation
`grep -o`	Only the matched text	Data extraction
`grep -oE`	Extract matches with ERE	Parsing structured text
`grep -v`	Invert — non-matching lines	Filtering out noise
`grep -i`	Case insensitive	Mixed-case logs
`grep -w`	Whole word match	Avoid partial matches
`grep -rn`	Recursive with line numbers	Codebase search
`grep -rl`	Recursive, filenames only	Find which files contain X

`grep -rL`	Recursive, files without match	Find files missing a header
`grep -C3`	3 lines of context	Log debugging
`grep -inE`	Case-insensitive ERE with line numbers	General search
`grep -Fxq`	Exact line, fixed string, quiet	Check if list contains item
`grep -oP`	Extract with PCRE	Complex extraction

Quick Reference — Chapter 7

MODE FLAGS

`grep (no flag)` BRE — Basic Regular Expressions (default)

`grep -E / egrep` ERE — Extended Regular Expressions (recommended)

`grep -F / fgrep` Fixed string — no regex, fastest option

`grep -P` PCRE — lookahead, lookbehind, lazy (GNU only)

MOST-USED FLAG COMBINATIONS

`grep -q 'pat' file` Silent boolean test — use in if statements

`grep -oE 'pat' file` Extract matching text only (one match per line)

`grep -rn 'pat' dir/` Recursive search with line numbers

`grep -rl 'pat' dir/` List files containing a match (no content)

`grep -v 'noise' file` Exclude lines matching a pattern

`grep -C3 'pat' file` Match with 3 lines of surrounding context

`grep -c 'pat' file` Count matching lines

`grep -Fxq 'str' file` Test if file contains an exact line (fixed, whole-line, quiet)

> **What is coming next:** Chapter 8 covers `sed` and regex — the substitute command in depth, BRE vs ERE in sed, the `-E` flag, all substitution flags (`g` , `i` , `N` , `p` , `w`), replacement back-references, case-modifier sequences (`\u` `\U` `\l` `\L`), and address-targeted regex substitutions.

CHAPTER 8

sed and Regex in Depth

The sed Substitute Command

The `s` command is the heart of sed. It finds a regex match on a line and replaces it with a replacement string. Understanding every part of its syntax — address, pattern, replacement, and flags — is what makes sed a powerful text transformation engine.

```
3,/end/s/pattern/replacement/flags
address      regex pattern   replacement text   flags
```

- **Address** — which lines to operate on (optional; if omitted, all lines)
- **Pattern** — BRE by default; ERE with `sed -E`
- **Replacement** — literal text plus special tokens (`&` , `\1` – `\9` , case modifiers)
- **Flags** — modify how the substitution behaves (`g` , `i` , `p` , `w` , `Nth`)
- **Delimiter** — `/` is conventional but any character works: `s|/usr|/opt|g`

BRE vs ERE in sed

By default sed uses BRE. The `-E` flag (GNU sed; also `-r` on some systems) switches to ERE — the same extended syntax that `grep -E` uses.

```
# BRE (default) — grouping and + need backslashes
sed 's/\(foo\)\+/bar/g' file.txt

# ERE with -E — cleaner syntax, + and () work directly
sed -E 's/(foo)+/bar/g' file.txt

# BRE alternation needs \|
sed 's/cat\|dog/pet/g' file.txt
```

```
# ERE alternation — just |
sed -E 's/cat|dog/pet/g' file.txt
```

Feature	BRE (default)	ERE (`-E`)
Grouping	`\(...\)`	`(...)`
One or more	`\+`	`+`
Zero or one	`\?`	`?`
Alternation	`|`	`|`
Backreference (pattern)	`\1`	`\1`
Backreference (replacement)	`\1`	`\1`
`&` whole match	Yes	Yes

> **Use `sed -E` for new scripts.** The ERE syntax is less cluttered and more consistent with `grep -E` and `awk`. Reserve BRE-only syntax for scripts that must run on strictly POSIX systems without GNU sed.

Substitution Flags

`g`
Global

Replace ALL occurrences on the line, not just the first. The most commonly used flag.

`i / I`
Case insensitive

Match the pattern without regard to letter case. GNU sed extension.

`N (number)`
Nth occurrence

Replace only the Nth match on the line, e.g. `2` replaces the second match only.

`Ng`
Nth and beyond

Replace from the Nth occurrence onwards. `2g`

`p`
Print

Print the line if a substitution was made. Use with `sed -n` to print only changed lines.

`w file`
Write

Write the substituted line to a file. Creates or truncates the file at start of script.

replaces second, third, fourth… matches.

e

Execute

Execute the replacement as a shell command and replace the line with its output. GNU only.

m / M

Multi-line

Makes `^` and `$` match at the start/end of each embedded newline in the pattern space.

Combine

Mix freely

Flags combine: `gi` = global case-insensitive; `2p` = print only if 2nd occurrence replaced.

```
# g: replace all occurrences (default replaces only the first)
echo "aaa bbb aaa" | sed 's/aaa/xxx/'
xxx bbb aaa        # only first replaced
echo "aaa bbb aaa" | sed 's/aaa/xxx/g'
xxx bbb xxx        # all replaced

# i: case-insensitive replacement
echo "Error ERROR error" | sed 's/error/FAULT/gi'
FAULT FAULT FAULT

# 2: replace only the second occurrence
echo "cat and cat and cat" | sed 's/cat/dog/2'
cat and dog and cat

# 2g: replace from the second occurrence onwards
echo "cat and cat and cat" | sed 's/cat/dog/2g'
cat and dog and dog

# p with -n: print ONLY lines where a substitution was made
sed -n 's/ERROR/FAULT/p' app.log

# w: write substituted lines to a separate file
sed 's/ERROR/FAULT/gw /tmp/faults.txt' app.log

# e: replace line with output of a shell command (GNU sed only)
echo "date" | sed 's/.*/date/e'         # replaces "date" with the output of
```

Replacement Special Tokens

`&`

Whole match

Inserts the entire matched text. Equivalent to `\0` in some flavours.

`\1 … \9`

Capture group

Inserts the text captured by group N. Requires `(...)` in ERE or `\(...\)` in BRE.

`\\`

Literal backslash

A literal `\` in the output. Use `\\` in the replacement to insert a real backslash.

`\n`

Newline in output

Inserts a newline into the replacement text, splitting the line in two.

```
# &: wrap the entire match in brackets
echo "hello world" | sed 's/[a-z]*/[&]/g'
[hello] [world]

# \1 \2: reorder captured groups (date reformatting)
echo "2026-06-11" | sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3\/\2\/\1/'
11/06/2026

# Swap first and last name
echo "Smith, John" | sed -E 's/([A-Za-z]+), ([A-Za-z]+)/\2 \1/'
John Smith

# Quote every word
echo "one two three" | sed 's/[^ ]*/"\&"/g'
"one" "two" "three"

# Insert a newline: split "key=value" into two lines
echo "name=Alice" | sed 's/=/\n/'
name
Alice

# Escape & and \ when they must be literal in replacement
echo "foo" | sed 's/foo/a\&b/'     # output: a&b  (\& = literal &)
echo "foo" | sed 's/foo/a\\b/'     # output: a\b  (\\ = literal \)
```

Case Modifier Sequences (GNU sed)

GNU sed supports case-conversion sequences in the replacement string. These are not available in POSIX sed or BSD sed.

`\u` **Uppercase next** Capitalise the next character only.	`\l` **Lowercase next** Lowercase the next character only.	`\U` **UPPERCASE rest** Uppercase from here to end of replacement or `\E`.	`\L` **lowercase rest** Lowercase from here to end of replacement or `\E`.
`\E` **End modifier** Stop the `\U` or `\L` modifier — return to default case.			

CASE MODIFIER EXAMPLES

`s/\b\(.\)/\u\1/g` — title-case every word hello world → Hello World `s/.*/\U&/` — uppercase the entire line hello world → HELLO WORLD `s/.*/\L&/` — lowercase the entire line HELLO WORLD → hello world `s/\b\([a-z]\)/\u\1/g` — capitalise first letter of each word the quick brown fox → The Quick Brown Fox `s/\(ERROR\)/\L\1/g` — lowercase the match only, then \E stops it found ERROR in log → found error in log

```
# Title-case: capitalise the first letter of every word (BRE)
echo "the quick brown fox" | sed 's/\b\([a-z]\)/\u\1/g'
The Quick Brown Fox

# Title-case with ERE
echo "the quick brown fox" | sed -E 's/\b([a-z])/\u\1/g'
The Quick Brown Fox

# SCREAMING_SNAKE_CASE from lowercase
echo "my variable name" | sed 's/ /_/g; s/.*/\U&/'
```

```
MY_VARIABLE_NAME

# Normalise HTTP method names (mixed case → uppercase)
echo "method: Get" | sed -E 's/(GET|POST|PUT|DELETE|PATCH)/\U\1/gi'
method: GET

# camelCase to snake_case: insert _ before each uppercase letter then lowerca
echo "camelCaseVariable" | sed 's/\([A-Z]\)/_\L\1/g'
camel_case_variable
```

Address-Targeted Substitution

Addresses restrict which lines a sed command applies to. This is one of sed's most powerful features — you can combine regex addresses with substitution commands to perform context-aware replacements.

`N`

Line number

Apply only to line N. `1` = first line; `$` = last line.

`N,M`

Line range

Apply to lines N through M inclusive.

`/regex/`

Pattern address

Apply to every line that matches the regex.

`/start/,/end/`

Pattern range

Apply from the line matching `start` to the line matching `end`, inclusive.

`first~step`

Step address

Apply to every Nth line starting at `first`. `0~2` = even; `1~2` = odd. GNU only.

`addr!`

Negated address

Apply to every line that does NOT match the address. `/pattern/!s/x/y/`

```
# Line number addresses
sed '1s/^/# File header\n/' file.txt          # add comment before line 1
sed '$s/$/\n# End of file/' file.txt          # add comment after last line
```

```
sed '2,5s/^/    /' file.txt                    # indent lines 2–5

# Pattern address: substitute only on lines matching a condition
sed '/^Port/s/22/2222/' /etc/ssh/sshd_config  # change port only on Port line
sed '/ERROR/s/$/  <<< ALERT/' app.log          # append ALERT to error lines

# Pattern range: substitute within a block
sed '/\[section\]/,/\[end\]/s/^ *//' config.ini # strip leading spaces in sec
sed '/^---/,/^---/s/\bFoo\b/Bar/g' file.md    # replace only inside front-mat

# Negated address: substitute on every line EXCEPT those matching
sed '/^#/!s/foo/bar/g' script.sh              # skip comment lines
sed '/^$/!s/^/  /' file.txt                   # indent all non-empty lines

# Step address: number every other line (GNU sed)
sed '1~2s/^/> /' file.txt                    # prefix odd lines with "> "

# Combined: two addresses + substitution in a block
sed '/^BEGIN/,/^END/ { s/\bold\b/new/g; s/foo/bar/g }' file.txt
```

Alternate Delimiters

The delimiter does not have to be `/`. Any punctuation character can be used. This avoids the "leaning toothpick" problem when patterns or replacements contain slashes.

```
# Path substitution — using | as delimiter avoids escaping every /
sed 's|/usr/local|/opt|g' Makefile

# Compare: using / would require escaping all the slashes
sed 's/\/usr\/local/\/opt/g' Makefile   # harder to read

# URL rewriting with # as delimiter
sed 's#https://old.example.com#https://new.example.com#g' links.txt

# Pattern address with alternate delimiter — use \%..%
sed '\%/usr/local%s/local/share/' config.txt

# The rule: whatever character follows s is the delimiter
```

```
sed 's,foo,bar,g' file.txt     # comma as delimiter
sed 's@foo@bar@g' file.txt     # @ as delimiter
```

Practical sed Recipes

Config file management

```
# Set a key to a new value (handles both "key value" and "key=value" forms)
sed -E 's/^(MaxSessions\s*)=?\s*.*/\1= 20/' sshd_config

# Uncomment a setting: remove leading # from lines matching a key
sed '/^#.*Port /s/^#//' sshd_config

# Comment out a setting
sed 's/^PermitRootLogin/# PermitRootLogin/' sshd_config

# In-place edit with backup (GNU sed)
sed -i.bak 's/^Port 22/Port 2222/' /etc/ssh/sshd_config
```

Data reformatting

```
# ISO date (YYYY-MM-DD) to US format (MM/DD/YYYY)
sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\2\/\3\/\1/g' dates.txt

# CSV: quote unquoted fields (add quotes around each comma-separated value)
sed -E 's/([^,]+)/"\1"/g' data.csv

# Remove ANSI colour escape codes from terminal output
sed -E 's/\x1B\[[0-9;]*[mK]//g' coloured.log

# Normalise multiple spaces to one
sed 's/  */ /g' file.txt           # BRE: two or more spaces → one space
sed -E 's/ {2,}/ /g' file.txt      # ERE: cleaner syntax

# Strip leading and trailing whitespace
sed 's/^[[:space:]]*//; s/[[:space:]]*$//' file.txt
```

```
# Extract value from "key: value" format (print only the value)
sed -n '/^Name:/s/^Name:[[:space:]]*//'p record.txt
```

Log and source code manipulation

```
# Mask sensitive data: replace password values
sed -E 's/(password=)[^ &]*/\1[REDACTED]/gi' app.log

# Add line numbers as prefix
sed = file.txt | sed 'N; s/\n/\t/'

# Delete blank lines
sed '/^[[:space:]]*$/d' file.txt

# Delete comments (lines starting with #)
sed '/^[[:space:]]*#/d' config.txt

# Rename a variable across a source file
sed -i 's/\bconnection_timeout\b/connect_timeout/g' config.py
```

Portability — GNU sed vs BSD sed

Feature	GNU sed (Linux)	BSD sed (macOS)
`-E` flag	Yes	Yes (BSD also accepts `-E`)
`-r` flag (same as -E)	Yes	No
`-i` in-place	`sed -i` or `sed -i.bak`	`sed -i ''` or `sed -i .bak`
Case modifiers (`\u \U \l \L`)	Yes	No
`i` / `I` flag (case insensitive)	Yes	No

`Nth` occurrence flag	Yes	Yes
`e` flag (execute)	Yes	No
Step address (`first~step`)	Yes	No
`\w` shorthand in regex	Yes	No — use `[[:alnum:]_]`
`\b` word boundary in regex	Yes	No — use `\<\>`

> **macOS `-i` gotcha:** BSD sed requires a space between `-i` and the suffix argument: `sed -i .bak '...'` or `sed -i '' '...'` (empty string for no backup). GNU sed accepts `sed -i.bak` with no space. To write scripts that work on both, use: `sed -i.bak '...'` — the `.bak` immediately after `-i` works on both GNU and BSD.

Quick Reference — Chapter 8

SUBSTITUTION SYNTAX

`s/pat/rep/`	Replace first match on each line
`s/pat/rep/g`	Replace all matches on each line
`s/pat/rep/2`	Replace only the 2nd occurrence
`s/pat/rep/gi`	All matches, case-insensitive (GNU)
`s/pat/rep/p`	Print if substitution made (use with `-n`)

REPLACEMENT TOKENS

`&`	Entire matched text
`\1 … \9`	Capture group back-reference
`\u\1`	Capitalise first char of group 1
`\U&`	Uppercase the entire match
`\L&`	Lowercase the entire match
`\n`	Insert newline in replacement

COMMON ADDRESS PATTERNS

`1s/...`	First line only
`$s/...`	Last line only
`/regex/s/...`	Lines matching regex
`/start/,/end/s/...`	Lines in a block
`/regex/!s/...`	Lines NOT matching regex

What is coming next: Chapter 9 covers regex in `awk` — pattern-action rules, `/regex/ { action }`, field matching with `$0` – `$NF`, the `match()` function, `sub()` and `gsub()`, capturing groups in `gawk`, and practical awk recipes that combine regex with arithmetic and formatted output.

CHAPTER 9

Regex in awk

The awk Program Model

`awk` is a complete data-processing language built around a simple loop: for every line of input, test each *pattern* — if it matches, run its *action*. Regex is the most common pattern type, and awk provides richer regex integration than either grep or sed because it can combine pattern matching with arithmetic, string functions, and formatted output.

```
BEGIN    { setup before any input is read }
/regex/  { action for matching lines }  — regex pattern
expr     { action when expression is true }  — expression pattern
pat,pat  { action for a range of lines }  — range pattern
END      { cleanup / summary after all input }
```

```
# Minimal example: print lines containing "error"
awk '/error/' app.log            # pattern only — default action is print
awk '/error/ { print }' app.log  # explicit print — identical result

# Multiple rules — each is tested independently against every line
awk '/ERROR/ { errors++ }
     /WARN/  { warns++ }
     END     { print "Errors:", errors, "  Warnings:", warns }' app.log
```

Fields and $0

awk automatically splits each input line into fields separated by the field separator `FS` (default: any whitespace). Regex is most powerful in awk when combined with field access.

```
INPUT LINE: "2026-06-11 14:32:00 ERROR DATABASE CONNECTION FAILED"
```

$0 = "2026-06-11 14:32:00 ERROR database connection failed" (entire line) $1 = "2026-06-11" $2 = "14:32:00" $3 = "ERROR" $4 … $NF = "database connection failed" (NF = number of fields = 6)

```
# Print only the timestamp and message from error lines
awk '/ERROR/ { print $1, $2, $4, $5, $6 }' app.log

# Custom field separator: parse colon-delimited /etc/passwd
awk -F: '/\/bin\/bash$/ { print $1 }' /etc/passwd  # users with bash shell

# FS can itself be a regex (gawk/mawk/nawk)
awk -F'[,;|]' '{ print $2 }' data.txt   # split on comma, semicolon, or pipe

# Set FS inside BEGIN (equivalent to -F)
awk 'BEGIN { FS = ":" } /root/ { print $1, $6 }' /etc/passwd

# NF: last field regardless of how many fields a line has
awk '{ print $NF }' file.txt          # last field of every line
awk '{ print $(NF-1) }' file.txt      # second-to-last field
```

Regex Operators in awk

/regex/
Match $0

Pattern rule: true when the whole line matches the regex.

$N ~ /regex/
Field match

True when field N matches the regex. Most useful awk regex feature.

$N !~ /regex/
Field no-match

True when field N does NOT match. Invert the test.

~ in conditions
Dynamic regex

Used inside `if`, `while`, ternary expressions — not just as a top-level pattern.

/re1/,/re2/
Range pattern

Activates at a line matching re1, stays active until a line matches re2.

!/regex/
Negated pattern

True when $0 does NOT match. Equivalent to `grep -v`.

```
# ~ operator: test a specific field
awk -F, '$3 ~ /^[0-9]+$/ { print $1, $3 }' data.csv   # field 3 is all digits
awk -F: '$3 ~ /^[0-9]+$/ && $3+0 >= 1000 { print $1 }' /etc/passwd  # UID >= 

# !~ operator: exclude a field pattern
awk -F, '$2 !~ /^[A-Z]/ { print "Bad name:", $2 }' names.csv

# Negated line pattern
awk '!/^#/ && !/^$/' config.txt   # skip comment and blank lines

# Range pattern: print lines between markers (inclusive)
awk '/^START/,/^END/' file.txt

# Range pattern with action
awk '/^\[database\]/,/^\[/ { print }' config.ini  # print [database] section

# Dynamic regex: variable used as regex with ~
PATTERN="ERROR"
awk -v pat="$PATTERN" '$0 ~ pat { print }' app.log

# ~ inside an if statement
awk '{
    if ($1 ~ /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/)
        print "date:", $1
    else
        print "not a date:", $1
}' file.txt
```

The match() Function

`match(string, regex)` searches for the regex anywhere in *string*, sets two special variables, and returns the position of the match (or 0 if no match).

```
MATCH("ORDER #12345 PLACED", /[0-9]+/)
returns 8 (start position of match, 1-indexed) RSTART = 8 (same as return value)
RLENGTH = 5 (length of matched text = "12345") no match → returns 0, RSTART = 0,
```

```
RLENGTH = -1
```

```
# match() + substr() to extract the matched text
echo "order #12345 placed" | awk '{
    if (match($0, /[0-9]+/))
        print "Order number:", substr($0, RSTART, RLENGTH)
}'
Order number: 12345

# Extract version number from a string
echo "nginx/1.24.0" | awk '{ match($0, /[0-9]+\.[0-9.]+/); print substr($0, RS
1.24.0

# Use match() return value as a boolean
awk 'match($0, /ERROR: (.+)/) { print RSTART, RLENGTH }' app.log

# Loop to extract all matches (one per call — advance string manually)
echo "foo123 bar456 baz789" | awk '{
    s = $0
    while (match(s, /[0-9]+/)) {
        print substr(s, RSTART, RLENGTH)
        s = substr(s, RSTART + RLENGTH)
    }
}'
123
456
789
```

match() with Capture Arrays (gawk)

In `gawk`, `match()` accepts a third argument — an array that is populated with capture groups, giving awk true regex capture support.

```
# match(string, regex, array) — gawk only
# array[0] = full match, array[1] = group 1, array[2] = group 2, ...
echo "2026-06-11" | gawk '{
    if (match($0, /([0-9]{4})-([0-9]{2})-([0-9]{2})/, a))
```

```
        printf "year=%s month=%s day=%s\n", a[1], a[2], a[3]
}'
year=2026 month=06 day=11

# Parse a log line into components
echo "2026-06-11 14:32:00 ERROR database connection failed" | \
gawk '{
    if (match($0, /^([0-9-]+) ([0-9:]+) ([A-Z]+) (.+)/, a))
        printf "date=%s time=%s level=%s msg=%s\n", a[1], a[2], a[3], a[4]
}'
date=2026-06-11 time=14:32:00 level=ERROR msg=database connection failed

# Extract URL components
echo "https://api.example.com:8080/v2/users" | \
gawk '{
    match($0, /^(https?):\/\/([^:/]+)(:([0-9]+))?(\/.*)?$/, a)
    print "scheme:", a[1]
    print "host:  ", a[2]
    print "port:  ", (a[4] ? a[4] : "default")
    print "path:  ", a[5]
}'
```

sub() and gsub()

`sub(regex, replacement, target)` replaces the first match; `gsub(regex, replacement, target)` replaces all matches. If `target` is omitted, `$0` is used. Both return the number of replacements made.

```
# sub(): replace first match in $0
echo "aaa bbb aaa" | awk '{ sub(/aaa/, "xxx"); print }'
xxx bbb aaa

# gsub(): replace all matches in $0
echo "aaa bbb aaa" | awk '{ gsub(/aaa/, "xxx"); print }'
xxx bbb xxx

# target a specific field
echo "foo ERROR bar ERROR" | awk '-F " " { gsub(/ERROR/, "FAULT", $2); print
```

```
foo FAULT bar ERROR   # only field 2 was targeted

# & in replacement = entire match (same as sed)
echo "hello world" | awk '{ gsub(/[a-z]+/, "[&]"); print }'
[hello] [world]

# Escape & and \ in awk replacement strings
echo "foo" | awk '{ gsub(/foo/, "a\\&b"); print }'   # → a&b  (literal &)
echo "foo" | awk '{ gsub(/foo/, "a\\\\b"); print }'  # → a\b  (literal \)

# Use gsub return value: count replacements
awk '{
    n = gsub(/ERROR/, "FAULT")
    if (n > 0) print NR": replaced", n, "occurrence(s)"
}' app.log

# Strip HTML tags from a line
echo "<b>Hello</b> <i>World</i>" | awk '{ gsub(/<[^>]*>/, ""); print }'
Hello World
```

gensub() — Capture Groups in Replacement (gawk)

`gensub(regex, replacement, how, target)` is a gawk extension that supports `\1` – `\9` back-references in the replacement string. Unlike `sub()` / `gsub()`, it returns the modified string rather than modifying in-place.

```
# gensub() syntax: gensub(regex, replacement, how [, target])
# how = "g" for global, "1" for first, "2" for second, etc.

# Reformat date YYYY-MM-DD to DD/MM/YYYY using capture groups
echo "2026-06-11" | gawk '{
    print gensub(/([0-9]{4})-([0-9]{2})-([0-9]{2})/, "\\3/\\2/\\1", "g")
}'
11/06/2026

# Swap first and last name
echo "Smith, John" | gawk '{ print gensub(/([A-Za-z]+), ([A-Za-z]+)/, "\\2 \\1
John Smith
```

```
# gensub does not modify $0 — assign the result
awk '{
    new = gensub(/([0-9]+)/, "(\\1)", "g")  # wrap all numbers
    print new
}' file.txt

# Replace only the second occurrence (how = "2")
echo "cat and cat and cat" | gawk '{ print gensub(/cat/, "dog", 2) }'
cat and dog and cat
```

Practical awk Recipes

Log file analysis

```
# Count events by log level with formatted output
awk '{
    if ($3 ~ /^(ERROR|WARN|INFO|DEBUG)$/) counts[$3]++
}
END {
    for (level in counts)
        printf "%-8s %d\n", level, counts[level]
}' app.log

# Extract slow queries: lines where duration field exceeds a threshold
awk -F, '$4 ~ /ms$/ { dur = $4+0; if (dur > 500) print $2, dur"ms" }' queries

# Summarise HTTP status codes from access log
awk '{ match($0, /" [0-9]{3} /); code = substr($0, RSTART+2, 3); codes[code]+
END { for (c in codes) print c, codes[c] }' access.log
```

CSV and structured data

```
# Print rows where email field looks valid
awk -F, '$3 ~ /^[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}$/ { print $
```

```
# Extract key=value pairs from config lines
awk '/^[[:alpha:]]/ {
    match($0, /([^=]+)=(.*)/, a)
    printf "key=[%s] val=[%s]\n", a[1], a[2]
}' config.txt

# Reformat: turn "Last, First" CSV into "First Last" TSV
awk -F, 'NR>1 {                              # skip header
    name = gensub(/([^,]+), ([^,]+)/, "\\2 \\1", 1, $1)
    print name "\t" $2 "\t" $3
}' people.csv
```

Validation and reporting

```
# Validate that every line of a file is a valid IPv4 address
awk '{
    if ($0 !~ /^([0-9]{1,3}\.){3}[0-9]{1,3}$/)
        print NR": not a valid IP:", $0
}' ip_list.txt

# Report lines where a field value is outside a numeric range
awk -F, '$2 ~ /^[0-9]+$/ && ($2+0 < 1 || $2+0 > 100) {
    print "Out of range on line", NR":", $2
}' scores.csv

# Find duplicate values in field 1
awk -F, '{ seen[$1]++ }
END { for (k in seen) if (seen[k] > 1) print "Duplicate:", k, "("seen[k]" tim
```

Text transformation

```
# Wrap long lines at word boundary (simple fold at 72 chars)
awk '{ while (length($0) > 72) {
    print substr($0, 1, 72)
    $0 = substr($0, 73)
} print }' longlines.txt
```

```
# Number every non-blank line
awk '/[^[:space:]]/ { printf "%4d  %s\n", ++n, $0; next } { print }' file.txt

# Convert Markdown-style headers to uppercase plain text
awk '/^#/ { gsub(/^#+[[:space:]]*/, ""); print toupper($0); next } { print }'
```

awk Regex Flavour and Portability

Feature	POSIX awk	gawk	mawk
ERE syntax (`+` `?` `|`)	Yes	Yes	Yes
POSIX classes (`[:alpha:]`)	Yes	Yes	Yes
`\b` word boundary	No	Yes	No
`\w` shorthand	No	Yes	No
`match(s,r,array)` — capture array	No	Yes	No
`gensub(r,rep,how)`	No	Yes	No
`-F` regex separator	Yes	Yes	Yes
Interval expressions (`{n,m}`)	Varies	Yes (default since 4.0)	No
`patsplit()` function	No	Yes	No

> **awk uses ERE, not BRE.** Unlike default grep or sed, all three awk flavours use ERE. You do not need `\+` or `\(` — use `+` and `(` directly. Forgetting this is a common source of "why doesn't my pattern work?" frustration.

Quick Reference — Chapter 9

REGEX OPERATORS

`/regex/ { action }`	Run action on lines where $0 matches
`!/regex/ { action }`	Run action on lines where $0 does NOT match
`$N ~ /regex/`	Field N matches regex
`$N !~ /regex/`	Field N does not match regex
`/re1/,/re2/ { action }`	Range pattern: re1 activates, re2 deactivates

REGEX FUNCTIONS

`match(s, /r/)`

Find r in s; sets RSTART, RLENGTH; returns position or 0

`match(s, /r/, arr)`

As above; also fills arr[0..N] with captures (gawk only)

`sub(/r/, rep)`	Replace first match in $0 in-place
`gsub(/r/, rep)`	Replace all matches in $0 in-place; returns count
`sub(/r/, rep, field)`	Replace first match in a specific field
`gensub(/r/, rep, how)`	Return new string; supports \1–\9 in rep (gawk)
`split(s, arr, /r/)`	Split string s on regex r into array arr

USEFUL VARIABLES

`$0`	Entire current line
`$N`	Field N (1-indexed)
`$NF`	Last field
`NF`	Number of fields on current line
`NR`	Current line number (total across all files)
`FS`	Field separator (set in BEGIN or with -F)
`RSTART / RLENGTH`	Set by match(): position and length of match

> **What is coming next:** Chapter 10 covers regex in Bash itself — the `[[ =~ ]]` operator, `BASH_REMATCH`, common validation patterns (email, IP, date, URL), building a personal regex library of reusable validation functions, and performance considerations for shell-native pattern matching vs calling external tools.

CHAPTER 10

Bash [[=~]] and BASH_REMATCH

Native Regex in the Shell

Since Bash 3.2, the `[[ ]]` compound command supports a `=~` operator for testing a string against a regular expression — without spawning a subprocess. The result is stored in the special array `BASH_REMATCH`, giving you capture groups directly in shell variables. This is the right tool when you need to *validate* or *parse* a value inside a script without the overhead of calling `grep`, `sed`, or `awk`.

```
[[ string =~ regex ]]
      string              ERE pattern — unquoted
      to test
```

- Returns **exit code 0** (true) if the string contains a match; non-zero otherwise
- The regex is ERE — the same syntax as `grep -E` and `awk`
- **No subprocess** — handled entirely by the shell; faster than any external tool for single-value tests
- Populates `BASH_REMATCH` on a successful match

```
# Basic test: does the string contain a pattern?
input="hello world"
if [[ $input =~ world ]]; then
    echo "matched"
fi

# Anchored: entire string must be a number
if [[ $input =~ ^[0-9]+$ ]]; then
    echo "is an integer"
fi

# Negated: ! in front of [[ ]]
if ! [[ $input =~ ^[0-9]+$ ]]; then
```

```
    echo "not an integer"
fi

# ERE features work directly
if [[ $input =~ ^(foo|bar)$ ]]; then
    echo "is foo or bar"
fi
```

The Quoting Rule — Critical

Do NOT quote the regex

The regex must be **unquoted** (or stored in a variable). Quoting it with `'...'` or `"..."` turns it into a literal string comparison — no special characters are interpreted.

DO quote the string

The left-hand side (the value being tested) **must be quoted** if it can contain spaces or shell metacharacters: `[[ "$var" =~ regex ]]`. An unquoted variable with spaces will cause a syntax error.

Store complex regex in a variable

If your pattern contains spaces or shell-special characters, store it in a variable first and use the variable **unquoted** on the right-hand side: `re='foo bar'; [[ $x =~ $re ]]`

The Bash 3.1 exception

Bash 3.1 *required* the regex to be quoted. Bash 3.2+ changed this. For maximum compatibility write patterns in a variable — it works in all versions.

```
# WRONG: quoted regex becomes a literal string match
if [[ $date =~ "^[0-9]{4}-[0-9]{2}" ]]; then   # won't work as regex

# CORRECT: unquoted regex
if [[ $date =~ ^[0-9]{4}-[0-9]{2} ]]; then

# CORRECT: variable holds the regex — unquoted on RHS
re='^[0-9]{4}-[0-9]{2}'
if [[ $date =~ $re ]]; then

# CORRECT: quoted LHS protects value with spaces
```

```
input="hello world"
if [[ "$input" =~ ^hello ]]; then
```

BASH_REMATCH — Reading Capture Groups

After a successful `=~` match, Bash populates the read-only array `BASH_REMATCH`:

```
PATTERN: ^([0-9]{4})-([0-9]{2})-([0-9]{2})$ INPUT: "2026-06-11"
BASH_REMATCH[0]    = "2026-06-11"  entire match BASH_REMATCH[1]    = "2026"
 capture group 1: year BASH_REMATCH[2]    = "06"  capture group 2: month
BASH_REMATCH[3]    = "11"  capture group 3: day
```

```
# Extract date components
date="2026-06-11"
if [[ $date =~ ^([0-9]{4})-([0-9]{2})-([0-9]{2})$ ]]; then
    year="${BASH_REMATCH[1]}"
    month="${BASH_REMATCH[2]}"
    day="${BASH_REMATCH[3]}"
    echo "Day: $day  Month: $month  Year: $year"
fi
Day: 11  Month: 06  Year: 2026

# Parse a "key=value" string
line="timeout=30"
if [[ $line =~ ^([a-z_]+)=(.+)$ ]]; then
    key="${BASH_REMATCH[1]}"
    val="${BASH_REMATCH[2]}"
    echo "$key → $val"
fi

# Parse a URL: scheme, host, path
url="https://api.example.com/v2/users"
re='^(https?)://([^/]+)(/.*)$'
if [[ $url =~ $re ]]; then
    echo "scheme: ${BASH_REMATCH[1]}"
    echo "host:   ${BASH_REMATCH[2]}"
    echo "path:   ${BASH_REMATCH[3]}"
```

```
fi

# BASH_REMATCH is overwritten by every =~ test — save values immediately
if [[ $line =~ ^([0-9]+) ]]; then
    number="${BASH_REMATCH[1]}"   # save BEFORE the next [[ ]] test
fi
```

Validation Function Library

The most common use of `[[ =~ ]]` is input validation. Here are battle-tested patterns as reusable shell functions. Source this file in your scripts with `source validate.sh`.

is_integer

Whole number, optional leading minus

```
^-?[0-9]+$
```

is_positive_int

Positive integer, no sign, no zero-pad

```
^[1-9][0-9]*$
```

is_float

Decimal number, optional sign and fraction

```
^-?[0-9]+(\.[0-9]+)?$
```

is_ipv4

Four 1–3 digit octets separated by dots

```
^([0-9]{1,3}\.){3}[0-9]{1,3}$
```

is_email

Practical email format (not RFC-complete)

```
^[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}$
```

is_iso_date

ISO 8601 date YYYY-MM-DD

```
^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$
```

is_hostname

RFC 1123 hostname label rules

```
^[a-zA-Z0-9]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(\.[a-zA-Z0-9]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$
```

is_semver

Semantic version MAJOR.MINOR.PATCH

```
^[0-9]+\.[0-9]+\.[0-9]+$
```

```
#!/usr/bin/env bash
# validate.sh — reusable validation functions using [[ =~ ]]
```

```
is_integer() {
    [[ "$1" =~ ^-?[0-9]+$ ]]
}

is_positive_int() {
    [[ "$1" =~ ^[1-9][0-9]*$ ]]
}

is_float() {
    [[ "$1" =~ ^-?[0-9]+(\.[0-9]+)?$ ]]
}

is_ipv4() {
    local re='^([0-9]{1,3}\.){3}[0-9]{1,3}$'
    [[ "$1" =~ $re ]] || return 1
    # also verify each octet is 0-255
    local IFS='.'
    local octet
    for octet in $1; do
        (( octet >= 0 && octet <= 255 )) || return 1
    done
}

is_email() {
    local re='^[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}$'
    [[ "$1" =~ $re ]]
}

is_iso_date() {
    local re='^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$'
    [[ "$1" =~ $re ]]
}

is_semver() {
    [[ "$1" =~ ^[0-9]+\.[0-9]+\.[0-9]+$ ]]
}

is_hex_colour() {
    [[ "$1" =~ ^#([0-9a-fA-F]{3}|[0-9a-fA-F]{6})$ ]]
}
```

```
is_slug() {   # URL-safe slug: lowercase letters, digits, hyphens
    [[ "$1" =~ ^[a-z0-9]+(-[a-z0-9]+)*$ ]]
}
```

Using the Validation Library

```
# Source and use in a script
source validate.sh

# Validate script arguments
validate_args() {
    local port="$1" host="$2"
    if ! is_positive_int "$port" || ((port > 65535)); then
        echo "Error: '$port' is not a valid port number" >&2
        return 1
    fi
    if ! is_ipv4 "$host" && ! is_hostname "$host" 2>/dev/null; then
        echo "Error: '$host' is not a valid host" >&2
        return 1
    fi
}

# Validate user input in a loop
while true; do
    read -rp "Enter an IPv4 address: " ip
    is_ipv4 "$ip" && break
    echo "Invalid IP address, try again"
done

# Use in a case-like dispatch via if/elif
describe_input() {
    local val="$1"
    if   is_integer  "$val"; then echo "integer"
    elif is_float    "$val"; then echo "float"
    elif is_ipv4     "$val"; then echo "IPv4 address"
    elif is_email    "$val"; then echo "email address"
    elif is_iso_date "$val"; then echo "ISO date"
    else                          echo "unknown"
```

```
    fi
}
```

Parsing with BASH_REMATCH — Real Patterns

```
# Parse a log line into variables
line="2026-06-11 14:32:00 ERROR database connection failed"
re='^([0-9-]+)[[:space:]]+([0-9:]+)[[:space:]]+([A-Z]+)[[:space:]]+(.+)$'
if [[ $line =~ $re ]]; then
    log_date="${BASH_REMATCH[1]}"
    log_time="${BASH_REMATCH[2]}"
    log_level="${BASH_REMATCH[3]}"
    log_msg="${BASH_REMATCH[4]}"
fi

# Extract semantic version parts from a dependency line
dep="requests==2.31.0"
if [[ $dep =~ ^([a-z_-]+)==([0-9]+)\.([0-9]+)\.([0-9]+)$ ]]; then
    pkg="${BASH_REMATCH[1]}"
    major="${BASH_REMATCH[2]}"
    minor="${BASH_REMATCH[3]}"
    patch="${BASH_REMATCH[4]}"
    echo "$pkg v$major.$minor.$patch"
fi

# Check minimum version (compare captured integers)
if [[ $dep =~ ==([0-9]+)\.([0-9]+) ]]; then
    maj="${BASH_REMATCH[1]}"
    min="${BASH_REMATCH[2]}"
    if (( maj > 2 || ( maj == 2 && min >= 28 ) )); then
        echo "version OK"
    fi
fi

# Process lines from a file — parse each one
while IFS= read -r line; do
    if [[ $line =~ ^([a-z_]+)[[:space:]]*=[[:space:]]*(.+)$ ]]; then
        printf "key=%-20s value=%s\n" "${BASH_REMATCH[1]}" "${BASH_REMATCH[2]
```

```
    fi
done < config.ini
```

Performance — When to Use [[=~]] vs External Tools

CHOOSING THE RIGHT TOOL FOR REGEX WORK

Fast — use [[=~]] Validating a single variable or argument in a script Parsing one value into components (URL, version, date) Type-checking user input in a loop Conditional branching based on a variable's format Use grep/awk/sed Searching or filtering lines across a file or stream Extracting all matches from many lines (-o pattern) Transforming text in bulk (substitution across thousands of lines) Counting matches, aggregating statistics from a file Avoid [[=~]] inside loops over file lines Reading a 50,000-line file with while read + [[=~]] is slower than piping it through grep or awk — spawn the tool once

```
# SLOW: shell loop with [[ =~ ]] on a large file
while IFS= read -r line; do
    [[ $line =~ ERROR ]] && echo "$line"
done < bigfile.log

# FAST: let grep do all the line scanning in C
grep 'ERROR' bigfile.log

# GOOD use of [[ =~ ]]: single-value validation, no file I/O
deploy() {
    local version="$1"
    if ! [[ $version =~ ^[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
        echo "Error: version must be MAJOR.MINOR.PATCH" >&2
        return 1
    fi
    # ... deploy logic
}
```

Common Traps and Fixes

Trap	Example	Fix
Quoted regex becomes literal	`[[ $x =~ "^[0-9]+" ]]`	Remove quotes: `[[ $x =~ ^[0-9]+ ]]`
Pattern with spaces causes error	`[[ $x =~ foo bar ]]`	Store in variable: `re="foo bar"; [[ $x =~ $re ]]`
Unquoted LHS with spaces splits	`[[ $var =~ pat ]]` where var="a b"	Always quote LHS: `[[ "$var" =~ pat ]]`
BASH_REMATCH overwritten	Using BASH_REMATCH after another `[[ ]]`	Assign to named variables immediately after the test
Partial match surprises	`[[ "foobar" =~ foo ]]` is true	Anchor with `^` and `$` for full-string match
Using in `[ ]` instead of `[[ ]]`	`[ "$x" =~ pattern ]`	`=~` only works in `[[ ]]` — single brackets do not support it
ERE features missing in old Bash	`{n,m}` intervals on Bash < 3.2	Use Bash 3.2+ or rewrite without interval expressions

Quick Reference — Chapter 10

SYNTAX

`[[ "$var" =~ regex ]]`	Test — true (exit 0) if var contains a match
`! [[ "$var" =~ regex ]]`	Negated test — true if var does NOT match
`re='...'; [[ "$var" =~ $re ]]`	Store complex pattern in a variable (portable)
`${BASH_REMATCH[0]}`	Entire matched text
`${BASH_REMATCH[N]}`	Capture group N (1-indexed)

KEY RULES

`Regex — never quoted`

Quoting the RHS makes it a literal string comparison

`String — always quoted`

Quote the LHS to protect spaces and metacharacters

`ERE syntax`	Same flavour as grep -E and awk — not BRE
`Partial match by default`	

Use ^ and $ to anchor and force full-string matching

`Save BASH_REMATCH immediately`	It is overwritten by every subsequent [[ ]] test
`No subshell cost`	Faster than grep for validating a single value

> **What is coming next:** Chapter 11 covers PCRE — Perl-Compatible Regular Expressions in depth. Lookahead and lookbehind assertions, lazy quantifiers, named capture groups, non-capturing groups, atomic groups, and how to use PCRE features via `grep -P` and `perl -ne` inside shell pipelines.

CHAPTER 11

PCRE: Lookahead, Lookbehind, and Advanced Features

What Is PCRE?

PCRE — Perl-Compatible Regular Expressions — is a regex engine originally written to match Perl 5's regex behaviour. It extends ERE with features that are impossible in BRE or ERE: lookahead and lookbehind assertions, lazy quantifiers, named capture groups, non-capturing groups, and more.

In the Bash toolkit, PCRE is available via:

- `grep -P` — GNU grep with PCRE engine (Linux; not available in BSD/macOS grep)
- `perl -ne` — inline Perl one-liners, available everywhere Perl is installed
- `pcregrep` — standalone PCRE grep (installable on macOS via Homebrew)

> **`grep -P` is GNU grep only.** On macOS, `grep -P` will print an error. Use `perl -ne` or install `pcregrep` for portable PCRE across platforms.

Lookahead and Lookbehind Assertions

Assertions test what is *around* the match position without consuming characters — they are zero-width. The matched text does not include the assertion's content.

`(?=...)`

Positive lookahead

Match position where the pattern *ahead* succeeds. The ahead text is not included in the match.

`(?!...)`

Negative lookahead

Match position where the pattern *ahead* does NOT match. Useful for exclusions.

`(?<=...)`

Positive lookbehind

`(?<!...)`

Negative lookbehind

Match position where the pattern *behind* succeeds. The behind text is not included in the match.

Match position where the pattern *behind* does NOT match. Excludes by context.

POSITIVE LOOKAHEAD: \W+(?=:) — WORD FOLLOWED BY COLON, COLON NOT CAPTURED

Input: host: localhost port: 8080 timeout: 30 Match: `host` (colon stays in input, not consumed) Match: `port` Match: `timeout`

POSITIVE LOOKBEHIND: (?<=\$)[0-9]+ — DIGITS PRECEDED BY $, $ NOT CAPTURED

Input: price is $42.00 and $199.99 Match: `42` ($ was required but not included in match) Match: `199`

```
# Positive lookahead: extract keys (words before a colon)
echo "host: localhost  port: 8080" | grep -oP '\w+(?=:)'
host
port

# Positive lookbehind: extract values (digits after $)
echo "price $42 or $199" | grep -oP '(?<=\$)[0-9]+'
42
199

# Negative lookahead: match "foo" not followed by "bar"
echo "foobar foobaz fooqux" | grep -oP 'foo(?!bar)\w*'
foobaz
fooqux

# Negative lookbehind: match digits NOT preceded by a minus
echo "count=42 temp=-7 size=100" | grep -oP '(?<!-)[0-9]+'
42
100

# Lookahead + lookbehind together: extract value between delimiters
echo 'name="Alice"' | grep -oP '(?<=")[^"]+'
Alice

# Validate a password: must contain a digit AND an uppercase letter
pw="Secret42"
if echo "$pw" | grep -qP '^(?=.*[0-9])(?=.*[A-Z]).{8,}$'; then
```

```
    echo "password strong"
fi
```

Lazy Quantifiers

All ERE/BRE quantifiers are *greedy* — they match as much as possible. PCRE adds *lazy* (minimal) quantifiers by appending `?` to any quantifier. A lazy quantifier matches as little as possible while still allowing the overall pattern to succeed.

Greedy: .* .+ .{2,5}

Match as MUCH as possible, then back off only if needed. The engine consumes the whole string first, then retreats.

Lazy: .*? .+? .{2,5}?

Match as LITTLE as possible, then expand only if needed. The engine tries the shortest match first.

```
INPUT: "<B>BOLD</B> AND <I>ITALIC</I>"

Greedy: <.+> matches <b>bold</b> and <i>italic</i> (one huge match) Lazy: <.+?>
matches <b> then </b> then <i> then </i>
```

```
# Greedy: matches from first < to LAST > on the line
echo "<b>bold</b> and <i>italic</i>" | grep -oP '<.+>'
<b>bold</b> and <i>italic</i>   # one match — greedy consumed everything

# Lazy: matches each individual tag
echo "<b>bold</b> and <i>italic</i>" | grep -oP '<.+?>'
<b>
</b>
<i>
</i>

# Lazy: extract content of the FIRST quoted string
echo '"Alice" and "Bob"' | grep -oP '".*?"'
"Alice"
"Bob"

# Lazy: match shortest balanced {...} block
```

```
echo "{a} text {b} more {c}" | grep -oP '\{.+?\}'
{a}
{b}
{c}

# All lazy quantifier forms
# *?   zero or more, lazy
# +?   one or more, lazy
# ??   zero or one, lazy
# {n,m}?  between n and m, lazy
```

> **ERE workaround without PCRE:** `[^delimiter]*` is the ERE/BRE equivalent for lazy matching up to a delimiter. Use `[^"]*` instead of `.*?` to match inside double quotes. This works in all tools but only when the delimiter is a single character.

Named Capture Groups

PCRE allows capture groups to be named, making patterns self-documenting and the code that uses them easier to maintain. Names are referenced in the pattern and in replacement strings.

```
# Named group syntax: (?P<name>...) or (?<name>...) in PCRE

# grep -oP: extract named match (prints the whole match — use perl for the na
echo "2026-06-11" | grep -oP '(?P<year>[0-9]{4})-(?P<month>[0-9]{2})-(?P<day>
2026-06-11

# perl -ne: access named groups via %+ hash
echo "2026-06-11" | perl -ne '
    if (/(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})/) {
        print "year=$+{year} month=$+{month} day=$+{day}\n"
    }
'
year=2026 month=06 day=11

# perl: named group in replacement with \k<name>
echo "Smith, John" | perl -pe 's/(?P<last>\w+), (?P<first>\w+)/$+{first} $+{l
```

```
John Smith

# Parse a log line with named groups for readability
echo "2026-06-11 14:32:00 ERROR connection failed" | perl -ne '
    if (/(?P<date>\S+) (?P<time>\S+) (?P<level>\S+) (?P<msg>.+)/) {
        printf "LEVEL=%-8s MSG=%s\n", $+{level}, $+{msg}
    }
'
```

Group Types — Complete Reference

`(...)`

Capturing group

Captures the matched text. Numbered left-to-right by opening paren. Back-referenced as `\1`–`\9`.

`(?:...)`

Non-capturing group

Groups for alternation or quantification without consuming a capture slot. Keeps group numbers tidy.

`(?P<name>...)` `(?<name>...)`

Named capturing group

Captures and assigns a name. Also numbered. Referenced as `(?P=name)` or `\k<name>`.

`(?=...)` `(?!...)`

Lookahead assertion

Zero-width. Does not capture or consume. Tests what follows the current position.

`(?<=...)` `(?<!...)`

Lookbehind assertion

Zero-width. Tests what precedes the current position. Must be fixed-width in most PCRE implementations.

`(?>...)`

Atomic group

Once matched, the engine does not backtrack into this group. Prevents catastrophic backtracking.

`(?i:...)` `(?s:...)` `(?m:...)`

Inline modifier group

Apply flags to a portion of the pattern. `i`=case-insensitive, `s`=dot-all, `m`=multiline.

`(?|...)`

Branch reset group

Each branch of alternation uses the same group numbers. Useful for equivalent alternatives.

```
# Non-capturing group: group for alternation without a capture slot
echo "colour color" | grep -oP 'colo(?:u?r)'
colour
color

# Contrast: capturing group uses a slot even if you don't need it
# Non-capturing keeps group numbers clean when you only care about \1
echo "2026-06-11" | grep -oP '(?:[0-9]{4})-([0-9]{2})-([0-9]{2})'
# \1 = month (06), \2 = day (11) — year group not numbered

# Inline modifier: case-insensitive only within a group
echo "HTTP https FTP" | grep -oP '(?i:https?)'
HTTP
https

# Atomic group: prevent backtracking into a completed group
# Useful for performance; prevents runaway backtracking on long strings
echo "aaaaab" | grep -P '(?>a+)b'   # a+ consumed, no backtrack needed
```

perl -ne — PCRE in Shell Pipelines

`perl -ne` runs a Perl script line-by-line over stdin or a file. It is the most portable way to use full PCRE in shell pipelines, available on virtually every Unix system.

```
# -n: loop over lines, no auto-print; -e: inline script
# Basic filter: print matching lines (equivalent to grep -P)
perl -ne 'print if /ERROR/' app.log

# -p: loop and auto-print (like sed); substitution with s///
perl -pe 's/foo/bar/g' file.txt

# Extract only the matched portion (like grep -oP)
perl -ne 'print "$1\n" while /([0-9]{4}-[0-9]{2}-[0-9]{2})/g' log.txt

# Named groups + formatted output
perl -ne '
    if (/(?P<ip>\d{1,3}(?:\.\d{1,3}){3}).*?"(?P<method>GET|POST|PUT|DELETE)/)
        print "$+{ip}  $+{method}\n"
```

```
    }
' access.log

# Lazy match: extract each JSON string value
echo '{"name":"Alice","city":"Paris"}' | \
perl -ne 'print "$1\n" while /"([^"]+)":\s*"(.+?)"/g and print "key=$1 val=$2'

# Cleaner version for key:value JSON extraction
echo '{"name":"Alice","city":"Paris"}' | \
perl -ne 'while (/"(\w+)":\s*"([^"]+)"/g) { print "$1 = $2\n" }'
name = Alice
city = Paris

# Lookahead in perl: extract words before colons
echo "host: localhost  port: 8080" | \
perl -ne 'print "$1\n" while /(\w+)(?=:)/g'
host
port

# Multi-line mode: . matches newlines with /s, ^ and $ match line ends with /m
perl -0777 -ne 'print "$1\n" while /BEGIN(.*?)END/gs' file.txt
# -0777: slurp entire file into $_; /s: dot matches newline; /g: all matches
```

Practical PCRE Recipes

Extraction

```
# Extract all IPv4 addresses from a file
grep -oP '\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b' access.log | sort -u

# Extract all URLs (http and https)
grep -oP 'https?://[^\s">]+' page.html

# Extract email addresses
grep -oP '[\w.+-]+@[\w-]+\.[\w.]+' mail.txt | sort -u

# Extract value of a specific JSON field (not for production — use jq)
```

```
echo '{"user": "alice", "role": "admin"}' | \
grep -oP '(?<="user": ")[^"]+'
alice

# Extract all words after a specific keyword
echo "imported module requests as req" | grep -oP '(?<=as )\w+'
req
```

Validation and filtering

```
# Password policy: min 8 chars, must have digit + uppercase + special
grep -P '^(?=.*[0-9])(?=.*[A-Z])(?=.*[!@#$%^&*]).{8,}$' passwords.txt

# Match lines that do NOT start with a comment or blank line
grep -P '^(?!#|$)' config.txt

# Find duplicate consecutive words
grep -P '\b(\w+)\s+\1\b' document.txt

# Validate that a version string follows MAJOR.MINOR.PATCH-PRERELEASE format
echo "2.31.0-beta.1" | grep -P '^[0-9]+\.[0-9]+\.[0-9]+(?:-[\w.]+)?$'
```

Substitution with perl -pe

```
# Reformat date using named groups
echo "2026-06-11" | perl -pe 's/(?P<y>\d{4})-(?P<m>\d{2})-(?P<d>\d{2})/$+{d}\
11/06/2026

# Mask credit card numbers: keep last 4 digits
echo "card: 4111-1111-1111-1234" | \
perl -pe 's/\b\d{4}(?:-\d{4}){2}(?=-\d{4}\b)/****-****-****/g'
card: ****-****-****-1234

# Add a thousands separator to large numbers
echo "1234567" | perl -pe 's/(\d)(?=(\d{3})+(?!\d))/$1,/g'
1,234,567
```

```
# Remove ANSI colour codes from terminal output
cat coloured.log | perl -pe 's/\x1b\[[0-9;]*m//g'

# Normalise line endings: CRLF to LF
perl -pi -e 's/\r\n/\n/g' file.txt    # in-place edit
```

Catastrophic Backtracking — and How to Avoid It

Poorly written PCRE patterns on adversarial input can cause the regex engine to run for an exponentially long time — a *ReDoS* (Regular Expression Denial of Service) attack. Understanding the cause is essential for writing safe patterns.

```
# DANGEROUS: nested quantifiers on overlapping patterns
# Pattern (a+)+ on input "aaaaaaaaaaaab" causes exponential backtracking
# Each 'a' can be assigned to inner or outer group in 2^N ways
# grep -P '(a+)+b' <<< "aaaaaaaaaaaab"  — hangs!

# SAFE alternatives:
# 1. Atomic group — commit to match, no backtracking
grep -P '(?>a+)b'  # atomic: once a+ matched, don't re-try assignments

# 2. Possessive quantifiers (PCRE2 / newer Perl)
grep -P 'a++b'     # ++ = possessive: no backtrack

# 3. Rewrite to eliminate ambiguity
grep -P 'a+b'      # the nested group was unnecessary here
```

> **ReDoS rules of thumb:** avoid nesting quantifiers (`(a+)+`), avoid alternation where branches can match the same characters (`(a|a)+`), and always anchor patterns that validate full strings (`^...$`). Use `(?>...)` atomic groups to commit the engine when backtracking serves no purpose.

PCRE Feature Availability

Feature	grep -P (GNU)	grep -P (BSD/macOS)	perl -ne	pcregrep
Lookahead `(?=...)`	Yes	Error	Yes	Yes
Lookbehind `(?<=...)`	Yes	Error	Yes	Yes
Lazy quantifiers `.*?`	Yes	Error	Yes	Yes
Named groups `(?P<n>...)`	Yes	Error	Yes	Yes
Non-capturing `(?:...)`	Yes	Error	Yes	Yes
Atomic group `(?>...)`	Yes	Error	Yes	Yes
Inline flags `(?i:...)`	Yes	Error	Yes	Yes
Slurp mode `-0777`	N/A	N/A	Yes	pcregrep -M

Quick Reference — Chapter 11

ASSERTIONS

`(?=pat)` Positive lookahead — position followed by pat

`(?!pat)` Negative lookahead — position NOT followed by pat

`(?<=pat)` Positive lookbehind — position preceded by pat

`(?<!pat)`

Negative lookbehind — position NOT preceded by pat	

LAZY QUANTIFIERS

`*? +? ?? {n,m}?`	Match as little as possible (PCRE only)

GROUP TYPES

`(?:...)`	Non-capturing group
`(?P<name>...)`	Named capturing group
`(?>...)`	Atomic group — no backtracking
`(?i:...)`	Inline flag — case-insensitive within group

SHELL TOOLS

`grep -oP 'pat'`	Extract PCRE matches (GNU grep only)
`grep -P 'pat' file`	Filter lines with PCRE (GNU grep only)
`perl -ne 'print if /pat/'`	PCRE filter — portable alternative to grep -P
`perl -pe 's/pat/rep/'`	PCRE substitution — portable alternative to sed -E
`perl -0777 -ne '...'`	Slurp entire file; enables multi-line matching

What is coming next: Chapter 12 — the final chapter — covers building and maintaining a personal regex toolkit: a library of tested patterns, a test harness for validating your regex against edge cases, combining all the tools (grep, sed, awk, Bash, Perl) in real pipeline scripts, and a complete course reference card.

CHAPTER 12

Your Regex Toolkit: Patterns, Testing, and Real Pipelines

Building a Personal Pattern Library

The most productive regex habit is maintaining a library of tested, annotated patterns you reach for repeatedly. Instead of rewriting *"what was that IPv4 pattern again?"* from scratch, you keep one authoritative version that you know works correctly — including the edge cases that bit you before.

Below is a complete `regex_lib.sh` — source it in any script with `source regex_lib.sh`.

```
#!/usr/bin/env bash
# regex_lib.sh — tested, reusable regex patterns and validation functions
# Usage: source regex_lib.sh

# ── Anchored validation patterns ──────────────────────────────────────
RE_INTEGER='^-?[0-9]+$'
RE_POS_INT='^[1-9][0-9]*$'
RE_FLOAT='^-?[0-9]+(\.[0-9]+)?$'
RE_HEX='^(0[xX])?[0-9a-fA-F]+$'

RE_IPV4='^([0-9]{1,3}\.){3}[0-9]{1,3}$'
RE_IPV6='^([0-9a-fA-F]{0,4}:){2,7}[0-9a-fA-F]{0,4}$'
RE_CIDR='^([0-9]{1,3}\.){3}[0-9]{1,3}/([0-9]|[1-2][0-9]|3[0-2])$'
RE_MAC='^([0-9a-fA-F]{2}[:-]){5}[0-9a-fA-F]{2}$'

RE_EMAIL='^[[:alnum:]._%+-]+@[[:alnum:].-]+\.[[:alpha:]]{2,}$'
RE_URL='^https?://[[:alnum:]._~:/?#\[\]@!$&()*+,;=%-]+'
RE_HOSTNAME='^[a-zA-Z0-9]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(\.[a-zA-Z0-9]([a-zA

RE_ISO_DATE='^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$'
RE_TIME_24='^([01][0-9]|2[0-3]):[0-5][0-9](:[0-5][0-9])?$'
RE_SEMVER='^[0-9]+\.[0-9]+\.[0-9]+(-[[:alnum:].-]+)?(\+[[:alnum:].-]+)?$'

RE_SLUG='^[a-z0-9]+(-[a-z0-9]+)*$'
RE_HEX_COLOUR='^#([0-9a-fA-F]{3}|[0-9a-fA-F]{6})$'
RE_UUID='^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-f
```

```
RE_BASE64='^[A-Za-z0-9+/]+={0,2}$'

# ── Validation functions ──────────────────────────────────────
# Each returns 0 (true) if valid, 1 (false) if not

is_integer()    { [[ "$1" =~ $RE_INTEGER   ]]; }
is_pos_int()    { [[ "$1" =~ $RE_POS_INT   ]]; }
is_float()      { [[ "$1" =~ $RE_FLOAT     ]]; }
is_email()      { [[ "$1" =~ $RE_EMAIL     ]]; }
is_url()        { [[ "$1" =~ $RE_URL       ]]; }
is_hostname()   { [[ "$1" =~ $RE_HOSTNAME  ]]; }
is_iso_date()   { [[ "$1" =~ $RE_ISO_DATE ]]; }
is_semver()     { [[ "$1" =~ $RE_SEMVER    ]]; }
is_uuid()       { [[ "$1" =~ $RE_UUID      ]]; }
is_hex_colour() { [[ "$1" =~ $RE_HEX_COLOUR ]]; }
is_slug()       { [[ "$1" =~ $RE_SLUG      ]]; }

is_ipv4() {
    [[ "$1" =~ $RE_IPV4 ]] || return 1
    local IFS='.' octet
    for octet in $1; do
        (( octet >= 0 && octet <= 255 )) || return 1
    done
}
```

A Regex Test Harness

Every pattern in your library should have tests. This harness lets you declare positive cases (must match) and negative cases (must not match), then run them all and report failures — just like a unit test suite, but for regex.

```
#!/usr/bin/env bash
# test_regex.sh — regex unit test harness
source regex_lib.sh

PASS=0; FAIL=0

assert_match() {       # assert_match "label" pattern "value"
```

```
    local label="$1" re="$2" val="$3"
    if [[ "$val" =~ $re ]]; then
        printf "  \e[32mPASS\e[0m  %s: '%s'\n" "$label" "$val"
        (( PASS++ ))
    else
        printf "  \e[31mFAIL\e[0m  %s: '%s' should match /%s/\n" \
               "$label" "$val" "$re"
        (( FAIL++ ))
    fi
}

assert_no_match() {   # assert_no_match "label" pattern "value"
    local label="$1" re="$2" val="$3"
    if ! [[ "$val" =~ $re ]]; then
        printf "  \e[32mPASS\e[0m  %s: '%s' correctly rejected\n" "$label" "$
        (( PASS++ ))
    else
        printf "  \e[31mFAIL\e[0m  %s: '%s' should NOT match /%s/\n" \
               "$label" "$val" "$re"
        (( FAIL++ ))
    fi
}

# ── Tests ──────────────────────────────────────────────────────────
echo "=== RE_ISO_DATE ==="
assert_match    "valid date"         "$RE_ISO_DATE"  "2026-06-11"
assert_match    "leap day"           "$RE_ISO_DATE"  "2024-02-29"
assert_no_match "month 13"           "$RE_ISO_DATE"  "2026-13-01"
assert_no_match "day 00"             "$RE_ISO_DATE"  "2026-06-00"
assert_no_match "US format"          "$RE_ISO_DATE"  "06/11/2026"
assert_no_match "partial"            "$RE_ISO_DATE"  "2026-06"

echo "=== RE_EMAIL ==="
assert_match    "simple"             "$RE_EMAIL"     "user@example.com"
assert_match    "plus tag"           "$RE_EMAIL"     "user+tag@example.co.uk"
assert_no_match "no @"               "$RE_EMAIL"     "userexample.com"
assert_no_match "no TLD"             "$RE_EMAIL"     "user@example"
assert_no_match "spaces"             "$RE_EMAIL"     "user @example.com"

echo "=== is_ipv4 (with octet check) ==="
assert_match    "valid"              "$RE_IPV4"      "192.168.1.100"
```

```
assert_no_match "octet 256 (regex)" "$RE_IPV4"      "999.1.1.1"

# Summary
echo
printf "Results: \e[32m%d passed\e[0m  \e[31m%d failed\e[0m\n" "$PASS" "$FAIL
(( FAIL == 0 ))   # exit 0 if all passed, 1 if any failed
```

Sample output:

```
=== RE_ISO_DATE ===
  PASS  valid date: '2026-06-11'
  PASS  leap day: '2024-02-29'
  PASS  month 13 correctly rejected
  PASS  day 00 correctly rejected
  PASS  US format correctly rejected
  PASS  partial correctly rejected
=== RE_EMAIL ===
  PASS  simple: 'user@example.com'
  PASS  plus tag: 'user+tag@example.co.uk'
  PASS  no @ correctly rejected
  PASS  no TLD correctly rejected
  FAIL  spaces: 'user @example.com' should NOT match /^[[:alnum:]...

Results: 10 passed  1 failed
```

Tool Picker — Choosing the Right Tool

Do you need to match/filter lines from a file or stream?

→ yes, single pattern

```
grep 'pattern' file
```

→ yes, ERE syntax (|, +, ?)

```
grep -E 'pat1|pat2' file
```

→ yes, extract only the matched text

```
grep -oE 'pattern' file
```

→ yes, PCRE (lookahead, lazy, named groups)

```
      grep -P 'pattern' file — or — perl -ne 'print if /pat/'
```

Do you need to transform / substitute text?

```
   → simple find-and-replace on every line
      sed 's/old/new/g' file
   → ERE with capture groups, case modifiers
      sed -E 's/(group)/\U\1/g' file
   → PCRE substitution (lazy, lookahead in pattern)
      perl -pe 's/pattern/replacement/g' file
```

Do you need to process fields, aggregate, or do arithmetic?

```
   → split on delimiter, match per-field
      awk -F, '$3 ~ /pattern/ { print $1 }'
   → count, sum, group matches
      awk '/ERROR/ { count++ } END { print count }'
```

Are you validating a single variable inside a script?

```
   → no subprocess needed
      [[ "$var" =~ ^pattern$ ]]
```

Real Pipeline Scripts

Script 1 — Log report: errors by hour

```
#!/usr/bin/env bash
# hourly_errors.sh — count ERROR lines per hour from an app log
# Input format: "2026-06-11 14:32:00 ERROR message..."

awk '$3 == "ERROR" {
    match($2, /^([0-9]{2})/, t)
    hours[t[1]]++
}
END {
    print "Hour  Errors"
    print "----  ------"
    for (h = 0; h < 24; h++) {
        if (h in hours)
```

```
            printf "%02d:00  %d\n", h, hours[h]
    }
}' "${1:-app.log}"
```

Script 2 — Deploy guard: validate all inputs before proceeding

```
#!/usr/bin/env bash
# deploy.sh — validate arguments before running deployment
source regex_lib.sh

die() { echo "ERROR: $*" >&2; exit 1; }

VERSION="$1"
HOST="$2"
PORT="${3:-8080}"

is_semver  "$VERSION" || die "'$VERSION' is not a valid version (need MAJOR.M
is_hostname "$HOST"   || is_ipv4 "$HOST" || die "'$HOST' is not a valid host"
is_pos_int "$PORT"    || die "'$PORT' is not a valid port"
(( PORT <= 65535 ))   || die "Port $PORT exceeds 65535"

echo "Deploying v$VERSION to $HOST:$PORT ..."
# ... rest of deployment
```

Script 3 — Config auditor: scan for insecure settings

```
#!/usr/bin/env bash
# audit_config.sh — grep multiple files for security red flags

TARGET_DIR="${1:-/etc}"
ISSUES=0

check() {
    local label="$1" pattern="$2"
    shift 2
    local hits
    hits=$(grep -rnE --include='*.conf' --include='*.cfg' \
```

```
            "$pattern" "$@" 2>/dev/null)
    if [[ -n "$hits" ]]; then
        printf "\e[33mWARN\e[0m  %s\n%s\n\n" "$label" "$hits"
        (( ISSUES++ ))
    fi
}

check "PermitRootLogin enabled"     '^PermitRootLogin\s+yes'        "$TARGET_
check "PasswordAuthentication on"   '^PasswordAuthentication\s+yes' "$TARGET_
check "Plaintext password in config" '(?i)password\s*=\s*\S+'        "$TARGET_
check "Hardcoded API key"            '(?i)api.?key\s*=\s*["\x27]\S+'  "$TARGE

(( ISSUES == 0 )) && echo "No issues found."
exit $(( ISSUES > 0 ))
```

Script 4 — Access log analyser

```
#!/usr/bin/env bash
# access_summary.sh — parse nginx/apache combined log format
# Format: IP - - [date] "METHOD /path HTTP/x.x" STATUS bytes

awk '{
    # Extract status code (field 9) and request path
    status = $9
    match($7, /^\/[^?]*/, m)
    path = m[0]

    status_count[status]++
    if (status ~ /^[45]/) {
        errors[path]++
    }
    bytes += $10
}
END {
    print "=== Status codes ==="
    for (s in status_count)
        printf "  %s  %d\n", s, status_count[s]

    print "\n=== Top error paths ==="
```

```
    for (p in errors)
        printf "  %d  %s\n", errors[p], p
    | "sort -rn | head -10"

    printf "\n=== Total bytes: %.1f MB ===\n", bytes / 1024 / 1024
}' "${1:-/var/log/nginx/access.log}"
```

Complete Course Reference Card

ANCHORS

`^`	Start of line
`$`	End of line
`\b`	Word boundary (GNU)
`\< \>`	Word boundary (POSIX)
`^...$`	Whole-line / whole-string match

QUANTIFIERS

`* + ?`	0+, 1+, 0-or-1 (greedy)
`{n} {n,} {n,m}`	Exact, at-least, between
`*? +? ??`	Lazy (PCRE only)
`BRE: \+ \?`	ERE + and ? in BRE context

CHARACTER CLASSES

`[abc]`	Literal set
`[a-z]`	Range
`[^abc]`	Negated set
`[:alpha:]` `[:digit:]`	POSIX classes (inside [])
`\w \d \s`	GNU/PCRE shorthands
`.`	Any char except newline

GROUPS AND ALTERNATION

`(...)`	Capturing group — ERE/PCRE
`\(...\)`	Capturing group — BRE
`(?:...)`	Non-capturing group — PCRE
`(?P<n>...)`	Named group — PCRE
`| vs |`	Alternation ERE / BRE

PCRE ASSERTIONS

`(?=pat)`	Positive lookahead
`(?!pat)`	Negative lookahead
`(?<=pat)`	Positive lookbehind
`(?<!pat)`	Negative lookbehind
`(?>...)`	Atomic group — no backtrack

BACK-REFERENCES

`\1 … \9`	Group back-ref in pattern or replacement
`&`	Whole match in sed/awk replacement
`\u \U \l \L \E`	Case modifiers in GNU sed replacement
`BASH_REMATCH[N]`	Capture group N after [[=~]]

GREP FLAGS

`-E -F -P`	ERE / fixed-string / PCRE mode
`-o -n -c`	Only match / line nums / count
`-l -L -v`	Files with / without / invert
`-r -i -w`	Recursive / case-insensitive / word
`-q -A -B -C`	Quiet / after / before / context

SED / AWK / BASH

`sed -E 's/pat/rep/g'`	ERE substitution, all occurrences
`sed -i.bak '...'`	In-place edit with backup
`awk '$N ~ /pat/'`	Field N matches regex
`awk 'gsub(/p/,"r")'`	Replace all matches in $0
`[[ "$v" =~ ^pat$ ]]`	Native Bash ERE test, no subprocess

Flavour Comparison — The One Table

Feature	BRE	ERE	PCRE
One or more	`\+`	`+`	`+`
Zero or one	`\?`	`?`	`?`
Grouping	`\(...\)`	`(...)`	`(...)`
Alternation	`|`	`|`	`|`
Intervals `{n,m}`	`\{n,m\}`	`{n,m}`	`{n,m}`
Lazy quantifiers	No	No	`*? +? ??`
Non-capturing group	No	No	`(?:...)`
Lookahead / Lookbehind	No	No	`(?=)` `(?<=)`
Named groups	No	No	`(?P<n>...)`
Tools (typical)	grep, sed (default)	grep -E, sed -E, awk	grep -P, perl

Chapter Index

01 What Are Regular Expressions — engines, flavours, first patterns

02 Literals, the Dot, and Escaping — shell quoting, grep -F

03 Anchors — ^, $, \b, \<\>, whole-line matching

04 Character Classes — ranges, POSIX, \w \d \s shorthands

05 Quantifiers — *, +, ?, {n,m}, greedy rules, ERE vs BRE

06 Groups, Alternation, Back-references — BASH_REMATCH, &, \1–\9

07 grep and egrep in Depth — all flags, -o, -F, exit codes, pipelines

08 sed and Regex — s flags, case modifiers, address targeting

09 Regex in awk — ~, !~, match(), gsub(), gensub(), ERE

10 Bash [[=~]] — BASH_REMATCH, quoting rules, validation library

11 PCRE — lookahead, lookbehind, lazy, named groups, perl -ne

12 Your Toolkit — pattern library, test harness, real pipeline scripts

Regular Expressions in Bash — Complete

12 chapters · grep · sed · awk · [[=~]] · PCRE

BRE · ERE · PCRE · anchors · quantifiers · character classes · groups · back-references · lookahead · lookbehind · lazy matching · named groups · BASH_REMATCH · validation library · test harness