



PROJECT COURSE

grep: Searching Text

Finding text with grep: files, whole projects, basic, extended and Perl patterns, context and counting, pipelines and scripts, real-world jobs, and the faster tools that sit beside it. Every example was run, on two versions of grep.

12 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	First Searches	2
Chapter 2	Searching Folders	13
Chapter 3	Basic and Extended Regex	26
Chapter 4	Context and Output Control	42
Chapter 5	Perl Regex with -P	55
Chapter 6	grep in Pipelines and Scripts	68
Chapter 7	Real-World Searching	83
Chapter 8	Faster Alternatives	98

grep: Searching Text

Chapter 1 · First Searches

You have a file, and you want the lines that mention something. That is all `grep` does: it reads text a line at a time and prints the lines that match a pattern. It never changes the file, which is why it is safe to run on anything, and why it ends up in nearly every shell one-liner you will ever write. This chapter covers the everyday core: a plain search, case, inverting, whole words, fixed strings, several patterns, quoting, and the exit status that scripts depend on.

EVERY EXAMPLE HERE WAS RUN

The commands and their output below come from running them on GNU grep 3.0 (the one in Git Bash) and again on GNU grep 3.11 (in WSL): the output was identical, line for line, against the practice files in the next section. If your output differs, check that you typed the files exactly. For a one-page reminder of the options, see the *grep Cheat Sheet* in the Linux part of the Sidebar.

The Shape of a grep Command

```
grep [options] PATTERN [file ...]
```

The first thing that is not an option is the **pattern**; everything after it is a **file** to search. The name comes from an old editor command, `g/re/p`: *globally search for a regular expression and print*. A pattern is a regular expression by default, which is powerful and has a catch we will meet in a moment.

Practice Files

Make a new empty folder, change into it, and paste this once. It creates six small files you will use for the whole chapter. (The `'EOF'` in quotes stops the shell from changing anything inside.)

```
cat > server.log <<'EOF'
2026-10-08 09:14:02 INFO  server started on port 8080
2026-10-08 09:14:05 INFO  loaded 42 routes
2026-10-08 09:15:11 WARN  slow query: 1250 ms
```

```
2026-10-08 09:15:48 ERROR database connection lost
2026-10-08 09:15:49 INFO  retrying database connection
2026-10-08 09:15:52 INFO  database connection restored
2026-10-08 09:21:30 error: unexpected token in config.yml
2026-10-08 09:22:03 INFO  user bob logged in
2026-10-08 09:22:41 ERROR timeout while calling payments.example.com
2026-10-08 09:30:00 INFO  backup finished
EOF
```

```
cat > words.txt <<'EOF'
cat
Cat
CAT
concatenate
the cat sat
scatter
category
cat-flap
cat_food
EOF
```

```
cat > versions.txt <<'EOF'
version 1.2
version 1x2
version 10.2
price: $5.00
price: 5000
EOF
```

```
cat > names.txt <<'EOF'
bob
bobby
bob smith
alice
Bob
EOF
```

```
cat > notes.txt <<'EOF'
-v is the invert flag
remember to rotate the log
the cat sat on the mat
EOF
```

```
printf 'ERROR\nWARN\n' > patterns.txt
```

Your First Search

Print every line of `server.log` that contains `ERROR`:

```
grep ERROR server.log
```

```
2026-10-08 09:15:48 ERROR database connection lost
2026-10-08 09:22:41 ERROR timeout while calling payments.example.com
```

Two lines. Case matters: the lowercase `error:` on line 7 was not printed. Add `-n` to see the line number of each match, which is the quickest way to find your place in a long file:

```
grep -n ERROR server.log
```

```
4:2026-10-08 09:15:48 ERROR database connection lost
9:2026-10-08 09:22:41 ERROR timeout while calling payments.example.com
```

More than one file, and no file at all

Name several files and `grep` puts the **file name in front** of each line, so you can tell where a match came from:

```
grep cat words.txt notes.txt
```

```
words.txt:cat
words.txt:concatenate
words.txt:the cat sat
words.txt:scatter
words.txt:category
words.txt:cat-flap
words.txt:cat_food
notes.txt:the cat sat on the mat
```

Give it **no file** and it reads what is piped into it. This is how `grep` filters the output of other commands:

```
ls | grep txt
```

```
names.txt
notes.txt
patterns.txt
versions.txt
words.txt
```

You will often see `cat file | grep pattern`. It works, but `grep pattern file` does the same with one process fewer, and it keeps the file-name prefix when you give several files.

Case and Inversion

`-i` ignores case. Now the lowercase `error:` line appears too:

```
grep -i error server.log
```

```
2026-10-08 09:15:48 ERROR database connection lost
2026-10-08 09:21:30 error: unexpected token in config.yml
2026-10-08 09:22:41 ERROR timeout while calling payments.example.com
```

`-v` **inverts** the match: print the lines that do *not* match. Useful for removing noise:

```
grep -v INFO server.log
```

```
2026-10-08 09:15:11 WARN  slow query: 1250 ms
2026-10-08 09:15:48 ERROR database connection lost
2026-10-08 09:21:30 error: unexpected token in config.yml
2026-10-08 09:22:41 ERROR timeout while calling payments.example.com
```

Whole Words: -w and -x

A plain search finds the pattern *anywhere*: searching for `cat` would match `concatenate`, `scatter` and `category`. Two options narrow it down:

OPTION	THE MATCH MUST BE...
--------	----------------------

<code>-w</code>	a whole word : not touching a letter, digit or underscore on either side
-----------------	---

OPTION	THE MATCH MUST BE...
--------	----------------------

<code>-x</code>	the whole line , from start to end
-----------------	---

```
grep -w cat words.txt
```

```
cat
the cat sat
cat-flap
```

Look carefully at what `-w` did. `cat-flap` matched, because a hyphen is not part of a word. `cat_food` did *not*, because an underscore is. `Cat` and `CAT` did not, because we did not ask for `-i`.

```
grep -x bob names.txt
```

```
bob
```

Compare `-w` on the names file, where `bob smith` counts because `bob` is a whole word in it:

```
grep -w bob names.txt
```

```
bob
bob smith
```

WHAT COUNTS AS A "WORD"

For `-w`, a word is a run of letters, digits and underscores. Anything else (a space, a hyphen, a dot, the start or end of the line) is a boundary. Do not assume it means "separated by spaces".

Fixed Strings: -F

The pattern is a **regular expression**, and in a regular expression a dot means "any one character". So looking for `1.2` finds more than you meant:

```
grep '1.2' versions.txt
```

```
version 1.2  
version 1x2
```

`1x2` matched because the dot matched the `x`. When you mean the characters exactly as typed, with no special meaning, use `-F` (fixed string):

```
grep -F '1.2' versions.txt
```

```
version 1.2
```

The same trap with a price. Without `-F`, the dot matches the zero in `5000`:

```
grep '5.00' versions.txt
```

```
price: $5.00  
price: 5000
```

```
grep -F '5.00' versions.txt
```

```
price: $5.00
```

Use `-F` whenever you are searching for a file name, an address, a version number or anything else typed by a person or pasted from somewhere. Regular expressions themselves get their own chapters: *grep: Searching Text 3* for grep's flavours, and *Learning Regular Expressions* for the language in general.

Always Quote the Pattern

The shell reads your command line *before* grep does. Without quotes, a pattern with a space becomes two words, and grep takes the second as a file name:

```
grep unexpected token server.log
```

```
grep: token: No such file or directory
server.log:2026-10-08 09:21:30 error: unexpected token in config.yml
```

Here `unexpected` was the pattern and `token` a file that does not exist. Put the pattern in **single quotes**:

```
grep 'unexpected token' server.log
```

```
2026-10-08 09:21:30 error: unexpected token in config.yml
```

Single quotes pass everything inside exactly as typed. Double quotes still let the shell change `$`, backticks and backslashes, which is rarely what you want. Make single quotes your default and switch to double quotes only when you deliberately want a shell variable inside the pattern.

More Than One Pattern: -e and -f

Repeat `-e` to search for several patterns at once; a line matches if it matches *any* of them:

```
grep -e ERROR -e WARN server.log
```

```
2026-10-08 09:15:11 WARN    slow query: 1250 ms
2026-10-08 09:15:48 ERROR  database connection lost
2026-10-08 09:22:41 ERROR  timeout while calling payments.example.com
```

If the list is long or reused, put one pattern per line in a file and use `-f`. Our `patterns.txt` holds `ERROR` and `WARN`:

```
grep -f patterns.txt server.log
```

```
2026-10-08 09:15:11 WARN    slow query: 1250 ms
2026-10-08 09:15:48 ERROR  database connection lost
2026-10-08 09:22:41 ERROR  timeout while calling payments.example.com
```

`-e` has a second job. A pattern that starts with a dash looks like an option. Without `-e` (or a `--` to say “options end here”), `grep` would try to read `-v` as the invert flag. Both of these find the line that starts with `-v`:

```
grep -e '-v' notes.txt
```

```
-v is the invert flag
```

```
grep -- -v notes.txt
```

```
-v is the invert flag
```

Exit Status: How a Script Hears the Answer

Whatever you see on screen, `grep` also hands back a number when it finishes. In the shell it is `$?`. Scripts depend on it:

STATUS	MEANING
0	At least one line matched
1	No line matched. This is not an error : <code>grep</code> worked, it just found nothing
2	A real error: a missing file, a bad pattern, a bad option

`-q` (quiet) prints nothing at all and only sets the status, which is exactly what a script wants:

```
grep -q ERROR server.log; echo $?
```

```
0
```

```
grep -q CRITICAL server.log; echo $?
```

```
1
```

```
grep ERROR nosuchfile; echo $?
```

```
grep: nosuchfile: No such file or directory
2
```

The third one printed an error message from grep, then status `2`. Because the status is what `if` and `&&` look at, you can use grep as a test:

```
grep -q ERROR server.log && echo 'problems found'
```

```
problems found
```

A STATUS OF 2 BEATS A MATCH

Search a file that has matches *and* a file that does not exist, and grep still reports 2:

```
grep -c ERROR server.log nosuchfile; echo $?
```

```
server.log:2
grep: nosuchfile: No such file or directory
2
```

`server.log:2` is the number of matching lines (`-c`, which Chapter 4 covers; with several files each count carries its file name), the `grep:` line is the complaint about the missing file, and the last line is the status. A script that only checks for “0 or not 0” will take this as a failure even though grep found matches. Chapter 6 comes back to this when we put grep into scripts.

Hands-On Exercises

EXERCISE 1

Make an `inventory.txt` that mixes `Apple juice`, `apple pie`, `pineapple`, `green apple`, `APPLE sauce` and `crab apples`. Print the lines that contain the word apple in any case but not as part of another word, then those of them that are not pies.

 [View solution](#)

EXERCISE 2

A file `releases.txt` holds `3.1.4`, `3x1x4`, `3.1.4-beta`, `v3.1.4` and `31.14`. Find version 3.1.4 three ways (plain, fixed string, whole line) and explain exactly why each prints what it does.

 [View solution](#)

EXERCISE 3

Write `check.sh WORD FILE` that prints *found*, *missing* or an error depending on `grep`'s exit status, and exits with 2 on an error. It must cope with a word that starts with a dash. Test it on a match, a non-match, a missing file and the word `-v`.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- `grep [options] PATTERN [file ...]`: print the lines that match; never changes the file; several files get a file-name prefix; no file means read the pipe
- `-i` ignore case, `-v` invert, `-n` line numbers
- `-w` whole word (letters, digits and underscore are word characters); `-x` whole line
- `-F` fixed string: no regex, so a dot is just a dot; use it for names, addresses and version numbers
- Quote the pattern in single quotes: otherwise the shell splits it at spaces and expands `$` and `*` first
- `-e A -e B` several patterns; `-f file` patterns from a file; `-e` or `--` for a pattern that starts with a dash
- Exit status: `0` match, `1` no match (not an error), `2` error; `-q` prints nothing and only sets it; a 2 wins even when there were matches

COMING NEXT

grep: Searching Text 2 moves from one file to whole folders: `-r`, choosing which files to search, skipping `.git` and binary files, and handling file names with spaces.

grep: Searching Text

Chapter 2 · Searching Folders

Chapter 1 searched named files. Real work is usually “somewhere in this project”: dozens of folders, files you did not write, build output you do not want, hidden folders, and a few files that are not text at all. This chapter is about telling `grep` *where* to look and *what to skip*.

RUN ON TWO VERSIONS OF GREP

Every example was run on GNU `grep` 3.0 (Git Bash) and again on GNU `grep` 3.11 (WSL), on the same practice folder. There was **one** difference, the wording of the message for a binary file, and it is shown where it comes up. Two things in the output depend on *your* computer: the **order** of results (`grep` lists files in the order the folder holds them, which is not sorted and differs between file systems; add `| sort` when order matters) and the symlink example, which needs a Linux file system.

A Practice Project

Make an empty folder, change into it and paste this. It builds a small `project` folder with source files, a build folder, a hidden `.git` folder, a hidden `.env` file, a file name with spaces, and one binary file with the word `TODO` hidden inside it. You finish inside `project`.

```
mkdir -p project/src project/notes project/build project/logs project/data project/.git
cd project

printf '# Demo project\nTODO write the install guide\nRun app.py to start\n' > README.md
printf 'import os\n# TODO handle errors\ndef main():\n    print("hello")\n' > app.py
printf 'def helper():\n    # TODO remove this\n    return 1\n' > src/util.py
printf '// TODO rename\nconsole.log("hi");\n' > src/main.js
printf 'buy milk\nTODO call the plumber\n' > notes/todo.txt
printf '/* generated */ var a = 1; // TODO generated noise\n' > build/bundle.js
printf 'INFO started\nERROR something broke\n' > logs/app.log
printf 'data\0TODO inside a binary file\n' > data/image.bin
printf 'API_KEY=abc123\npassword=hunter2\n' > .env
printf '[core]\nrepositoryformatversion = 0\n# TODO placeholder inside .git\n' > .git/
printf 'TODO book the room\npassword reminder: change it\n' > "my notes/meeting notes.tx"
```

grep Does Not Enter Folders by Itself

```
grep TODO src
```

```
grep: src: Is a directory
```

Name a folder and plain grep refuses. The option that makes it go in is `-r` (*recursive*): search every file in the folder, every folder inside that, and so on.

Searching a Whole Tree: -r

```
grep -r TODO .
```

```
./git/config:# TODO placeholder inside .git
./app.py:# TODO handle errors
./build/bundle.js:* generated */ var a = 1; // TODO generated noise
Binary file ./data/image.bin matches
./my notes/meeting notes.txt:TODO book the room
./notes/todo.txt:TODO call the plumber
./README.md:TODO write the install guide
./src/main.js:// TODO rename
./src/util.py:    # TODO remove this
```

That is every file that contains `TODO`, with the path in front of each line. Notice what was searched without being asked: the **hidden** `.git` folder, the generated `build` folder, and `data/image.bin`, which is not text. Most of this chapter is about getting rid of that noise. (The binary file's line, *Binary file ... matches*, is grep 3.0's wording; grep 3.11 prints `grep: ./data/image.bin: binary file matches` instead, and prints it on standard error. More on that below.)

With **no folder** at all, GNU grep searches the current folder; the only difference is that the paths lose their leading `./`:

```
grep -r TODO
```

```
.git/config:# TODO placeholder inside .git
app.py:# TODO handle errors
build/bundle.js:/* generated */ var a = 1; // TODO generated noise
Binary file data/image.bin matches
my notes/meeting notes.txt:TODO book the room
notes/todo.txt:TODO call the plumber
README.md:TODO write the install guide
src/main.js:// TODO rename
src/util.py:    # TODO remove this
```

DOT OR STAR: THEY ARE NOT THE SAME

`grep -r PATTERN .` asks `grep` to walk the folder, so it sees hidden files. `grep -r PATTERN *` lets the *shell* expand `*` first, and the shell does not include names that start with a dot. Compare, looking for the word `password`:

```
grep -rl password *
```

```
my notes/meeting notes.txt
```

```
grep -rl password .
```

```
./.env
./my notes/meeting notes.txt
```

The star missed `.env`, which is exactly the sort of file that holds a password. If a search seems to be missing things, check whether you used `*`.

Just the File Names: `-l` and `-L`

Often you do not want the lines, only *which files*. `-l` lists the files that contain a match, each once:

```
grep -rl TODO .
```

```
./.git/config
./app.py
./build/bundle.js
./data/image.bin
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

`-L` is the opposite: the files that do **not** contain it.

```
grep -rL TODO .
```

```
./.env
./logs/app.log
```

-L AND THE EXIT STATUS

The exit status of `-L` does not mean “I listed something”. It still means “a line matched somewhere”. So when `-L` lists nothing (every file matched) the status is `0`, and when it lists a file (nothing matched in it) the status is `1`. Both versions behaved this way:

```
grep -L TODO app.py README.md src/util.py; echo status $?
```

```
status 0
```

```
grep -L NOSUCHWORD app.py; echo status $?
```

```
app.py
status 1
```

Do not use `if grep -L ...` as a test for “some file lacks the word”; look at its output instead.

Choosing Files: `--include` and `--exclude`

Search only files whose **name** matches a pattern with `--include`. Quote the pattern so the shell does not expand it:

```
grep -r1 TODO --include='*.py' .
```

```
./app.py
./src/util.py
```

Repeat the option to allow more than one kind:

```
grep -r1 TODO --include='*.py' --include='*.js' .
```

```
./app.py
./build/bundle.js
./src/main.js
./src/util.py
```

`--exclude` does the opposite: skip files whose name matches.

```
grep -r1 TODO --exclude='*.js' .
```

```
./.git/config
./app.py
./data/image.bin
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/util.py
```

The name is matched against the file name only, not the folder it is in. So `build/bundle.js` and `src/main.js` are both removed by `--exclude='*.js'`.

If you give both, **the last option that matches a file decides**. I tried both orders on the same pattern; the first keeps the JavaScript files, the second finds nothing at all:

```
grep -r1 TODO --exclude='*.js' --include='*.js' .
```

```
./.git/config
./app.py
./build/bundle.js
./data/image.bin
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

```
grep -r1 TODO --include='*.js' --exclude='*.js' .
```

(no output)

Skipping Folders: --exclude-dir

`--exclude-dir` takes a folder name and skips that folder wherever it appears in the tree. The two you will use most are `.git` and a build or dependency folder such as `node_modules` or `build`. Repeat the option for each:

```
grep -r1 TODO --exclude-dir=.git --exclude-dir=build .
```

```
./app.py
./data/image.bin
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

In bash, braces do the repeating for you (this is bash, not plain `sh`), with the same result:

```
grep -r1 TODO --exclude-dir={.git,build} .
```

```
./app.py
./data/image.bin
./my notes/meeting notes.txt
```

```
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

Binary Files

A file that contains a zero byte is treated as **binary**. If it matches, grep does not print the line (which would fill your screen with junk); it only says so:

```
grep TODO data/image.bin
```

```
Binary file data/image.bin matches
```

That is grep 3.0. In grep 3.11 the same search prints `grep: data/image.bin: binary file matches`, and it goes to **standard error**, not standard output. So a script that filters output with `| grep -v '^Binary'` works on one and not the other, and `2>/dev/null` hides the message on the newer one. Three ways to deal with binary files:

OPTION	WHAT IT DOES
<code>-I</code>	Skip binary files completely, as if they had no match
<code>-a</code>	Treat a binary file as text and print its matching lines
<code>--binary-files=TYPE</code>	The long form: <code>binary</code> (default), <code>text</code> (same as <code>-a</code>) or <code>without-match</code> (same as <code>-I</code>)

```
grep -I TODO data/image.bin; echo status $?
```

```
status 1
```

`-I` printed nothing and gave status `1`: “no match”, as far as it is concerned. With `-a` you get the text back (I used `-o`, which prints only the matching part, so the zero byte before it does not reach your screen):

```
grep -a -o 'TODO.*' data/image.bin
```

TODO inside a binary file

`-a` is for the occasional file you know is mostly text. Do not use it on a whole tree: it will dump raw binary into your terminal. For recursive searches of source code, `-I` is nearly always right:

```
grep -rIl TODO --exclude-dir=.git .
```

```
./app.py
./build/bundle.js
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

Note that `-l` without `-I` did list `data/image.bin` earlier: a binary file that matches is still a match.

Symbolic Links: -r and -R

A symbolic link is a name that points at another file or folder. `-r` does **not** follow links it finds *inside* the tree, so it does not search the same folder twice (or go round in a circle); `-R` follows them all. A link you **name yourself** on the command line is followed by both. This needs a real Linux file system, so I ran it in WSL's home folder (grep 3.11 only; Git Bash cannot make real links without special settings and would copy the folder instead, which proves nothing):

```
mkdir -p ~/g2link/real && cd ~/g2link
printf 'TODO in the real folder\n' > real/a.txt
ln -s real link
ls -l
```

```
link@
real/
```

```
grep -r TODO .
```

```
./real/a.txt:TODO in the real folder
```

```
grep -R TODO .
```

```
./link/a.txt:TODO in the real folder  
./real/a.txt:TODO in the real folder
```

```
grep -r TODO link
```

```
link/a.txt:TODO in the real folder
```

Use `-R` only when you know the tree is meant to be walked through its links, and be aware that a link pointing back up the tree can make it loop.

File Names with Spaces: `-Z` and `xargs -0`

Our tree has `my notes/meeting notes.txt`. A common pattern is to feed `grep`'s list of files to another command with `xargs`. Here it just prints one argument per line, so you can see how it split the names:

```
grep -rl TODO . | xargs -n1 echo
```

```
./.git/config  
./app.py  
./build/bundle.js  
./data/image.bin  
./my  
notes/meeting  
notes.txt  
./notes/todo.txt  
./README.md  
./src/main.js  
./src/util.py
```

`xargs` splits on spaces, so one file became three arguments, and any real command would complain that `./my` does not exist. The cure is to separate the names with a **zero byte**, which cannot appear in a file name. `-Z` makes `grep` print them that way, and `xargs -0` reads them that way:

```
grep -rLZ TODO . | xargs -0 -n1 echo
```

```
./.git/config
./app.py
./build/bundle.js
./data/image.bin
./my notes/meeting notes.txt
./notes/todo.txt
./README.md
./src/main.js
./src/util.py
```

Make `-LZ` / `xargs -0` a habit whenever the next command does something, and always whenever it changes or deletes files.

grep or find?

`grep -r` chooses files by *name* (`--include`, `--exclude`) and folder name. `find` can choose by far more: type, size, age, depth, the whole path, permissions. When the file selection is the hard part, let `find` pick the files and hand them to `grep` with `-exec ... {} +`. Add `-H` so a single file still shows its name:

```
find . -name '*.py' -exec grep -H TODO {} +
```

```
./app.py:# TODO handle errors
./src/util.py:    # TODO remove this
```

```
find . -name '*.py' -path './src/*' -exec grep -H TODO {} +
```

```
./src/util.py:    # TODO remove this
```

The second one used `-path`, which matches the whole path, something `--include` cannot do. For everyday work, `grep -r` with `--include` and `--exclude-dir` is shorter. (Chapter 8 looks at tools such as `ripgrep` that make the common case shorter still.)

A Good Default for Searching Code

```
grep -rIn --exclude-dir=.git PATTERN .
```

Recursive, skip binary files, show line numbers, skip the `.git` folder. Add more `--exclude-dir` options for the folders your own projects generate. If the search finds nothing and you expected something, quiet failures are the likely cause: `-s` hides “no such file” errors, and the exit status is `2` either way:

```
grep -r TODO nosuchdir; echo status $?
```

```
grep: nosuchdir: No such file or directory
status 2
```

```
grep -s TODO nosuchdir; echo status $?
```

```
status 2
```

Hands-On Exercises

EXERCISE 1

In the practice project, list, sorted, the Python and JavaScript files that contain `TODO`, ignoring the `build` and `.git` folders and binary files.

 [View solution](#)

EXERCISE 2

Show exactly which file `-I` hides from a `TODO` search, using two searches and `comm`, then list the files that contain no `TODO` at all and show what adding `-I` does to that list.

 [View solution](#)

EXERCISE 3

Find every file that mentions `password`, including hidden ones, and count the lines in each with `wc -l`, without breaking on `my notes/meeting notes.txt`. Show the version that breaks first.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- Plain `grep` refuses a folder (*Is a directory*); `-r` recurses; with no folder, GNU `grep` searches the current folder
- `grep -r PAT .` sees hidden files; `grep -r PAT *` does not (the shell skips names that start with a dot)
- `-l` files with a match, `-L` files without; the exit status of `-L` still means “a line matched”, not “I listed something”
- `--include='*.py'` / `--exclude='*.js'` match the file name only; the last matching option wins; quote them
- `--exclude-dir=.git` skips a folder anywhere in the tree; repeat it, or `--exclude-dir={.git,build}` in bash
- Binary files: default says they matched (3.0 on stdout: *Binary file X matches*; 3.11 on stderr: *grep: X: binary file matches*); `-I` skips them; `-a` prints them as text
- `-r` does not follow links inside the tree, `-R` does; a link named on the command line is followed by both
- File names with spaces: `grep -lZ ... | xargs -0 cmd`
- `find ... -exec grep -H PAT {} +` when the file choice is more than a name; results come in folder order, so `| sort` when order matters
- A good default: `grep -rIn --exclude-dir=.git PATTERN .`

COMING NEXT

grep: Searching Text 3 turns to the pattern itself: basic against extended regular expressions, anchors, bracket expressions and the character classes, and why `+` and `|` behave differently without `-E`.

grep: Searching Text

Chapter 3 · Basic and Extended Regex

So far the patterns were plain words. A pattern can also *describe* text: "a number", "a line that starts with a date", "the same word twice". That description language is the **regular expression**, and grep has two spellings of it. This chapter covers grep's own two spellings and where they trip people up. For regular expressions as a subject in their own right (and other tools that use them), see *Learning Regular Expressions*.

RUN ON TWO VERSIONS OF GREP

Every example was run on GNU grep 3.0 (Git Bash) and GNU grep 3.11 (WSL), on the same practice file. Output was identical except for three examples, which come up where they matter: how a Windows line ending (CRLF) interacts with `$`, and a warning about `*` at the start of a pattern. Anything marked *GNU* works in GNU grep (Linux, WSL, Git Bash) and may not in macOS/BSD grep, which I did not try.

One Language, Two Spellings

By default grep reads the pattern as a **basic** regular expression (BRE). With `-E` it reads an **extended** one (ERE). They describe the same things; they differ in which punctuation characters are operators and which are plain text. Learn the shared core first, then the difference.

Practice File

Paste this into an empty folder. It makes `lines.txt`, 20 lines chosen to trip up each idea in this chapter.

```
cat > lines.txt <<'EOF'
cat
concatenate
The cat sat on the mat
cot cut cat c.t
color colour
a+b=c
aaa
```

```
abab abab
price: $19.99
call 555-0142 now
2026-10-08 ERROR disk full
2026-10-08 WARN low memory
the the end
[note] keep this
path C:\temp\file
hello world
    indented line

file.txt and fileXtxt
end.
EOF
```

Where: Anchors

`^` matches the start of a line and `$` the end. They match a *position*, not a character.

```
grep '^the' lines.txt
```

```
the the end
```

Only the line that *starts* with a lowercase `the`: "The cat sat" has a capital T, and other lines have `the` in the middle.

```
grep 'end\.$' lines.txt
```

```
end.
```

Together, `^$` is an empty line, the classic way to count or remove blank lines:

```
grep -c '^$' lines.txt
```

```
1
```

What: One Character and Repeats

`.` matches any one character. `-o` prints just what matched, which is the best way to see what a pattern really did:

```
grep -o 'c.t' lines.txt
```

```
cat
cat
cat
cot
cut
cat
c.t
```

The last match is a literal `c.t` (the dot matched a dot). Put `*` after something to mean “zero or more of it”:

```
grep -E '*a' lines.txt | head -2; echo status $?
```

```
grep: warning: * at start of expression
cat
concatenate
status 0
```

Two traps with `*`. First, it is **greedy**: it takes as much as it can. `.*` (“anything”) between two letters swallows everything up to the *last* match on the line:

```
grep -o 'c.*t' lines.txt
```

```
cat
concatenat
cat sat on the mat
cot cut cat c.t
```

Look at the second line of output: it starts at the first `c` in `concatenate` and runs to the last `t` in the word. Second, “zero or more” includes **zero**, so a pattern made only of optional things

matches every line, even ones with no `x` at all:

```
grep -c 'x*' lines.txt; wc -l < lines.txt
```

```
20
```

```
20
```

All 20 lines matched `x*`. If you mean “one or more”, say so (`xx*`, or `x\+` / `x+` below).

Sets: Bracket Expressions

Square brackets match **one character from a set**. A `^` straight after the `[` reverses it, and a dash makes a range:

```
grep -o 'c[aou]t' lines.txt
```

```
cat
```

```
cat
```

```
cat
```

```
cot
```

```
cut
```

```
cat
```

```
grep -o 'c[^aou]t' lines.txt
```

```
c.t
```

```
grep -o '[0-9][0-9]*' lines.txt
```

```
19
```

```
99
```

```
555
```

```
0142
```

```
2026
```

```
10
```

```
08
```

```
2026
10
08
```

Inside brackets almost everything stops being special, so to match a literal dot you can write `[.]` instead of `\.`. A `]` must come **first** in the set, and a backslash is just a backslash:

```
grep '[.]' lines.txt
```

```
cot cut cat c.t
price: $19.99
file.txt and fileXtxt
end.
```

```
grep '[]' lines.txt
```

```
[note] keep this
```

```
grep '[\]' lines.txt
```

```
path C:\temp\file
```

Named classes

Ready-made sets, written with **double** brackets, which are defined by the locale (the language settings), so they follow that language's idea of a letter or a digit; a range such as `[a-z]` depends on how the locale orders letters:

CLASS	MATCHES
<code>[:digit:]</code>	0-9
<code>[:alpha:]</code> / <code>[:alnum:]</code>	letters / letters and digits
<code>[:upper:]</code> / <code>[:lower:]</code>	upper / lower case letters
<code>[:space:]</code>	space, tab, newline and similar
<code>[:punct:]</code>	punctuation

```
grep -o '[:digit:]]\+' lines.txt
```

```
19
99
555
0142
2026
10
08
2026
10
08
```

The class name sits *inside* a bracket expression, so the brackets are doubled. Writing it with one pair is a very common slip, and grep says so:

```
grep '[:digit:]' lines.txt; echo status $?
```

```
grep: character class syntax is [[:space:]], not [:space:]
status 2
```

The Difference: Basic vs Extended

Here is the whole difference. In a **basic** expression the characters `+ ? | ( ) { }` are ordinary text, and you put a **backslash** in front of them to make them operators. In an **extended** expression (`-E`) it is the other way round: they are operators, and the backslash makes them ordinary text.

YOU WANT	BASIC (DEFAULT)	EXTENDED (<code>-E</code>)
one or more	<code>a\+</code>	<code>a+</code>
optional	<code>a\?</code>	<code>a?</code>
either / or	<code>a\b</code>	<code>a b</code>
group	<code>\(ab\)</code>	<code>(ab)</code>
between 2 and 4	<code>a\{2,4\}</code>	<code>a{2,4}</code>

YOU WANT	BASIC (DEFAULT)	EXTENDED (<code>-E</code>)
a literal plus sign	<code>a+</code>	<code>a\+</code>

(The backslash forms in the basic column are GNU extensions: they work in GNU grep but not in every grep. Standard basic expressions have no `+ ? |` at all, which is a reason to prefer `-E` in scripts that may run elsewhere.) Every one of these has a version that *silently does the wrong thing*. “Optional u”, first as basic written like extended, which looks for a real question mark:

```
grep 'colou?r' lines.txt; echo status $?
```

```
status 1
```

```
grep 'colou\?r' lines.txt
```

```
color colour
```

```
grep -E 'colou?r' lines.txt
```

```
color colour
```

Status `1`, no error, no output: grep was happily looking for the text `colou?r`. And the mirror image, an extended pattern with a backslash, which is also a literal:

```
grep -E 'colou\?r' lines.txt; echo status $?
```

```
status 1
```

The plus sign is the clearest case, because one line of our file really contains `a+b`:

```
grep 'a+b' lines.txt
```

```
a+b=c
```

```
grep -E 'a+b' lines.txt
```

```
abab abab
```

```
grep -E 'a\+b' lines.txt
```

```
a+b=c
```

The first found the literal text `a+b`. The second found the line with `abab` in it, because `a+b` in extended means “one or more a, then b”. The third asked for a literal plus again. The same trap with either/or:

```
grep 'ERROR|WARN' lines.txt; echo status $?
```

```
status 1
```

```
grep 'ERROR\|WARN' lines.txt
```

```
2026-10-08 ERROR disk full
2026-10-08 WARN low memory
```

```
grep -E 'ERROR|WARN' lines.txt
```

```
2026-10-08 ERROR disk full
2026-10-08 WARN low memory
```

THE RULE I WOULD GIVE A NEW USER

Use `-E` whenever the pattern needs `+ ? | ( ) { }`, and then write them *without* backslashes. Reading one set of rules is easier than two. Reserve the basic form for simple patterns with no operators.

Groups, Repeats of Groups and Counts

Brackets repeat one character. A **group** repeats a whole piece, and `{n}` or `{n,m}` says how many times:

```
grep -E '(ab){2}' lines.txt
```

```
abab abab
```

```
grep '\(ab\)\'{2\}' lines.txt
```

```
abab abab
```

```
grep -E '[0-9]{3}-[0-9]{4}' lines.txt
```

```
call 555-0142 now
```

```
grep '[0-9]\{3\}-[0-9]\{4\}' lines.txt
```

```
call 555-0142 now
```

Those are the same searches written both ways. The extended ones are shorter and easier to read.

Back-references: Matching Something Again

A group remembers what it matched. `\1` means “the same text as group 1 matched”, `\2` group 2, and so on. It works in both forms (in `-E` it is a GNU extension; standard extended expressions do not have it):

```
grep -E '(ab)\1' lines.txt
```

```
abab abab
```

```
grep '\(ab\)\\1' lines.txt
```

```
abab abab
```

That found `abab`: the group matched `ab` and `\1` demanded `ab` again. The classic use is finding a repeated word:

```
grep -oE '\b([a-z]+) \1\b' lines.txt
```

```
abab abab  
the the
```

`\b` is a word boundary (below), `[a-z]+` is a word, then a space, then `\1`, the same word again. It found `the the`, and also `abab abab`, whose first `abab` is a word repeated.

Word Edges and Shorthands (GNU)

WRITE	MEANING
<code>\b</code>	a word boundary: between a word character and a non-word character, or the line edge
<code>\<</code> and <code>\></code>	the start and the end of a word
<code>\B</code>	NOT a word boundary (inside a word)
<code>\w</code> / <code>\W</code>	a word character (letter, digit, underscore) / anything else
<code>\s</code> / <code>\S</code>	white space / anything else

```
grep -o '\bcat\b' lines.txt
```

```
cat  
cat  
cat
```

```
grep -o '\<cat\>' lines.txt
```

```
cat
cat
cat
```

Both found the whole word `cat` three times (not the one inside `concatenate`). `\B` is the reverse, the one inside a word:

```
grep -o '\Bcat' lines.txt
```

```
cat
```

`\w+` is a neater way to say “a word” than `[a-zA-Z0-9_]+`:

```
grep -o '\w\+' lines.txt | head -3
```

```
cat
concatenate
The
```

Note that `\d` is **not** “a digit” in grep's basic or extended expressions. It is only a digit in Perl mode (`-P`, Chapter 5). In basic and extended it quietly means the letter `d`; use `[0-9]` or `[:digit:]`.

Matching a Special Character Literally

Put a backslash in front of a character that has a meaning to make it plain, or use `-F` (Chapter 1) when the whole pattern is plain text. Unescaped, the dot in a file name matches anything:

```
grep -o 'file.txt' lines.txt
```

```
file.txt
fileXtxt
```

```
grep -o 'file\.txt' lines.txt
```

```
file.txt
```

The dollar sign needs particular care because `$` is an **anchor** in extended expressions wherever it appears, but in a *basic* expression it is an anchor only at the very end of the pattern, and plain text anywhere else:

```
grep '$19' lines.txt; echo status $?
```

```
price: $19.99
status 0
```

```
grep -E '$19' lines.txt; echo status $?
```

```
status 1
```

```
grep -E '\$19' lines.txt
```

```
price: $19.99
```

The basic one found the price by accident of that rule; the extended one looked for “the end of a line, then 19”, which can never match. Escape it (`\$`) and both agree.

Which Match Wins: Longest

When a pattern could match in more than one way, grep takes the **longest** match at the leftmost position, whatever order you listed the alternatives in:

```
echo foobar | grep -oE 'foo|foobar'
```

```
foobar
```

```
echo foobar | grep -oE 'foobar|foo'
```

```
foobar
```

Perl-style engines (Chapter 5) take the *first* alternative that works instead, so the same patterns give different answers there. It matters whenever you use `-o` or edit with the match.

Windows Line Endings

A file written on Windows ends each line with two characters, carriage return and line feed (CRLF). On Linux, `grep` sees the carriage return as *part of the line*, so `$` (the end of the line) comes after it, and `done$` does not match `done`:

```
printf 'done\r\n' | grep 'done$'; echo status $?
```

```
status 1
```

```
printf 'done\r\n' | grep $'done\r$' | cat -A
```

```
done^M$
```

That is WSL's `grep` 3.11, behaving as it would on any Linux server. **Git Bash's `grep` 3.0 behaves differently:** it is a Windows build and evidently drops the carriage return, so the same first command printed `done` and exited with status `0` there, and the second printed nothing. So a pattern that works in Git Bash can fail when the same file reaches a Linux machine. If a `$` pattern mysteriously fails on a file from Windows, look for the carriage returns (`cat -A file` shows them as `^M`), and either match them (as the second command did, using bash's `$'...\r'` quoting) or convert the file first.

When `grep` Complains

A pattern that is not valid gives a message and status `2`:

```
grep -E '(' lines.txt; echo status $?
```

```
grep: Unmatched ( or \(  
status 2
```

```
grep '[' lines.txt; echo status $?
```

```
grep: Invalid regular expression  
status 2
```

In a *basic* expression an unmatched `(` is only a bracket character, so there is nothing to complain about (status `1`, nothing found):

```
grep '(' lines.txt; echo status $?
```

```
status 1
```

Newer GNU grep also warns about patterns that are legal but usually a mistake. A `*` at the start of an extended pattern has nothing in front of it to repeat. In 3.11 (shown here) grep says so; the 3.0 run printed no warning, and both treated the pattern the same way:

```
grep -E '*a' lines.txt | head -2; echo status $?
```

```
grep: warning: * at start of expression  
cat  
concatenate  
status 0
```

Putting It Together

Lines that start with a date and are an error or a warning:

```
grep -E '^[0-9]{4}-[0-9]{2}-[0-9]{2} (ERROR|WARN)' lines.txt
```

```
2026-10-08 ERROR disk full  
2026-10-08 WARN low memory
```

Read it left to right: start of line, four digits, a dash, two digits, a dash, two digits, a space, then either word. (`egrep` and `fgrep` are old names for `grep -E` and `grep -F`; both ran silently on the two versions here; I understand some newer GNU releases warn that they are obsolete, which I did not see, so write the option instead.)

Hands-On Exercises

EXERCISE 1

A log has well-formed lines, lines with a badly written date, lowercase levels, indented lines and a `WARNING` that is not a `WARN`. With one extended expression print only the lines that start with a full date and time, then `ERROR` or `WARN` as a whole word, then a space.

 [View solution](#)

EXERCISE 2

In a paragraph, print each repeated word (ignoring case) with `-o`, then print every word that has a doubled letter in it, using a back-reference.

 [View solution](#)

EXERCISE 3

Write a small shell function that takes a basic pattern and an extended pattern, runs both on `lines.txt` with `-o`, and prints whether they gave the same output. Use it to check four correct basic-to-extended translations and to catch two translations that look right and are not.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- Default = basic (BRE), `-E` = extended (ERE); same ideas, different punctuation. Perl (`-P`) is Chapter 5
- Position: `^` start, `$` end (`^$` = empty line). In basic, `$` is an anchor only at the end of the pattern; in extended it is one everywhere

- `.` any one character; `*` zero or more (greedy: `.*` runs to the last match; `x*` matches every line)
- Sets: `[abc]`, `^[abc]`, `[a-z]`; `]` first, specials lose their meaning inside; classes need double brackets: `[[:digit:]]`
- Extended: `+ ? | ( ) {n,m}` are operators and the backslash makes them literal; basic is the reverse (GNU: `\+ \? \| \(\ \) \{ \}`); a wrong choice is silent
- `\1`...`\9` repeat what a group matched; `\b \< \> \B \w \W \s \S` are GNU; `\d` is NOT a digit here
- Longest leftmost match wins, whatever the order of alternatives
- CRLF files: on Linux the carriage return is part of the line and `$` fails; Git Bash's grep hides it, so the same pattern can behave differently
- Invalid pattern: message and status 2; in basic an unmatched `(` is just text

COMING NEXT

grep: Searching Text 4 controls what grep prints around the match: context lines with `-A`, `-B` and `-C`, and the output options `-n -c -o -b -H -h -m -q -s`, with the difference between counting lines and counting matches.

grep: Searching Text

Chapter 4 · Context and Output Control

A match on its own is often not enough. You want the lines around it, which line it was, which file it came from, how many there were, or only the matching part. This chapter is the set of options that shape what grep prints, and the ones that make it quiet.

RUN ON TWO VERSIONS OF GREP

Every example was run on GNU grep 3.0 (Git Bash) and GNU grep 3.11 (WSL), on the same two practice files. The output was identical on both for every example in this chapter.

Two Practice Files

Paste this into an empty folder. `app.log` has 16 lines: five `ERROR` matches on four lines (line 14 has the word twice), two `WARN` lines and the rest `INFO`. `access.log` is a small web server log.

```
cat > app.log <<'EOF'
09:00:01 INFO    service starting
09:00:02 INFO    loading config
09:00:03 WARN    config key 'timeout' missing, using default
09:00:04 INFO    connecting to database
09:00:09 ERROR   database connection refused
09:00:10 INFO    retrying in 5s
09:00:15 ERROR   database connection refused
09:00:16 ERROR   giving up after 2 attempts
09:00:17 INFO    switching to read-only mode
09:00:20 INFO    listening on port 8080
09:05:00 INFO    health check ok
09:10:00 WARN    slow response: 2300 ms
09:10:30 INFO    health check ok
09:15:00 ERROR   disk usage 95%, ERROR threshold 90%
09:15:01 INFO    alert sent
09:20:00 INFO    health check ok
EOF
```

```
cat > access.log <<'EOF'
10.0.0.5 GET /index.html 200
10.0.0.7 GET /about.html 200
```

```
10.0.0.5 POST /login 302
10.0.0.9 GET /missing 404
10.0.0.5 GET /index.html 200
10.0.0.7 GET /index.html 200
10.0.0.5 GET /dashboard 200
10.0.0.9 GET /missing 404
10.0.0.7 GET /logout 302
EOF
```

Lines Around the Match: -A, -B and -C

An error is rarely understood on its own line. `-A N` prints N lines **A**fter each match, `-B N` N lines **B**efore, and `-C N` N lines of **C**ontext on both sides:

```
grep -A1 'giving up' app.log
```

```
09:00:16 ERROR giving up after 2 attempts
09:00:17 INFO  switching to read-only mode
```

```
grep -B1 'giving up' app.log
```

```
09:00:15 ERROR database connection refused
09:00:16 ERROR giving up after 2 attempts
```

```
grep -B1 -A2 'giving up' app.log
```

```
09:00:15 ERROR database connection refused
09:00:16 ERROR giving up after 2 attempts
09:00:17 INFO  switching to read-only mode
09:00:20 INFO  listening on port 8080
```

`-NUM` is a shorthand for `-C NUM`: `-2` means two lines either side.

```
grep -2 'giving up' app.log
```

```
09:00:10 INFO   retrying in 5s
09:00:15 ERROR  database connection refused
09:00:16 ERROR  giving up after 2 attempts
09:00:17 INFO   switching to read-only mode
09:00:20 INFO   listening on port 8080
```

Groups, separators and overlap

When two matches are close, their context overlaps and `grep` prints the lines *once*, as one group. When there is a gap, it prints a line containing `--` between the groups so you can see where one ended and the next began. Here, the three `ERROR` lines at 5, 7 and 8 became one group, and line 14 is a second:

```
grep -C1 ERROR app.log
```

```
09:00:04 INFO   connecting to database
09:00:09 ERROR  database connection refused
09:00:10 INFO   retrying in 5s
09:00:15 ERROR  database connection refused
09:00:16 ERROR  giving up after 2 attempts
09:00:17 INFO   switching to read-only mode
--
09:10:30 INFO   health check ok
09:15:00 ERROR  disk usage 95%, ERROR threshold 90%
09:15:01 INFO   alert sent
```

With `-n` (line numbers) the output shows which lines are matches and which are context. A **colon** follows the number of a match, a **dash** follows the number of a context line:

```
grep -n -C1 ERROR app.log
```

```
4-09:00:04 INFO   connecting to database
5-09:00:09 ERROR  database connection refused
6-09:00:10 INFO   retrying in 5s
7-09:00:15 ERROR  database connection refused
8-09:00:16 ERROR  giving up after 2 attempts
9-09:00:17 INFO   switching to read-only mode
--
13-09:10:30 INFO  health check ok
```

```
14:09:15:00 ERROR disk usage 95%, ERROR threshold 90%
15-09:15:01 INFO alert sent
```

The same rule applies when the file name is shown, so you can tell matches from context in a long listing. It also means a file name that itself contains a dash can be ambiguous to read:

```
grep -H -n -C1 'giving up' app.log
```

```
app.log-7-09:00:15 ERROR database connection refused
app.log:8-09:00:16 ERROR giving up after 2 attempts
app.log-9-09:00:17 INFO switching to read-only mode
```

You can change the separator, or remove it:

```
grep -C1 WARN app.log
```

```
09:00:02 INFO loading config
09:00:03 WARN config key 'timeout' missing, using default
09:00:04 INFO connecting to database
--
09:05:00 INFO health check ok
09:10:00 WARN slow response: 2300 ms
09:10:30 INFO health check ok
```

```
grep -C1 --no-group-separator WARN app.log
```

```
09:00:02 INFO loading config
09:00:03 WARN config key 'timeout' missing, using default
09:00:04 INFO connecting to database
09:05:00 INFO health check ok
09:10:00 WARN slow response: 2300 ms
09:10:30 INFO health check ok
```

```
grep -C1 --group-separator='====' WARN app.log
```

```
09:00:02 INFO loading config
09:00:03 WARN config key 'timeout' missing, using default
```

```
09:00:04 INFO    connecting to database
=====
09:05:00 INFO    health check ok
09:10:00 WARN    slow response: 2300 ms
09:10:30 INFO    health check ok
```

CONTEXT OPTIONS AND OTHER OPTIONS

`-o` and context do not mix well: `-o` prints only the matching text, so there is no context to show, but grep still prints the `--` separators between groups. And `-c` ignores context completely (the count is of matching lines only):

```
grep -o -C1 WARN app.log
```

```
WARN
--
WARN
```

```
grep -c -C1 ERROR app.log
```

```
4
```

Which Line, Which File, How Far In

`-n` adds the line number:

```
grep -n ERROR app.log
```

```
5:09:00:09 ERROR database connection refused
7:09:00:15 ERROR database connection refused
8:09:00:16 ERROR giving up after 2 attempts
14:09:15:00 ERROR disk usage 95%, ERROR threshold 90%
```

`-b` adds the **byte offset**, how many bytes into the file the line starts. It is rarely useful for people, but it is how you would jump to a spot with other tools. Combine with `-n` and you get line number, then offset:

```
grep -b 'giving up' app.log
```

```
275:09:00:16 ERROR giving up after 2 attempts
```

```
grep -nb 'giving up' app.log
```

```
8:275:09:00:16 ERROR giving up after 2 attempts
```

`-H` always shows the file name, and `-h` never does. By default `grep` shows it only when there is more than one file to search (Chapter 1). `-H` is what you want when you search one file but another tool will read the output, and `-h` when you search many files but want only the lines:

```
grep -H 'giving up' app.log
```

```
app.log:09:00:16 ERROR giving up after 2 attempts
```

```
grep -h GET access.log | head -2
```

```
10.0.0.5 GET /index.html 200  
10.0.0.7 GET /about.html 200
```

When the input is a pipe there is no file name at all, so `-H` shows `(standard input)`. `--label=NAME` lets you say what to call it:

```
cat app.log | grep -H 'giving up'
```

```
(standard input):09:00:16 ERROR giving up after 2 attempts
```

```
cat app.log | grep -H --label=app.log 'giving up'
```

```
app.log:09:00:16 ERROR giving up after 2 attempts
```

Only the Match: -o

`-o` prints just the part of the line that matched, **one match per line of output**, even when one input line has several. Line 14 contains `ERROR` twice, so it appears twice:

```
grep -on ERROR app.log
```

```
5:ERROR
7:ERROR
8:ERROR
14:ERROR
14:ERROR
```

The byte offset with `-o` is the position of *the match itself*, not of the line, which is how `-ob` tells two matches on the same line apart:

```
grep -ob ERROR app.log
```

```
168:ERROR
241:ERROR
284:ERROR
507:ERROR
529:ERROR
```

Counting: Lines or Matches

`-c` prints a count **of lines** that match, not of matches:

```
grep -c ERROR app.log
```

```
4
```

There are five `ERROR` matches (see the `-o` output above) but only four lines, because line 14 counts once. To count *matches*, print one per line with `-o` and let `wc -l` count them:

```
grep -o ERROR app.log | wc -l
```

```
5
```

With several files, `-c` gives one count per file, with the name in front, and prints `0` for a file with no match. `-h` drops the names:

```
grep -c ERROR app.log access.log
```

```
app.log:4  
access.log:0
```

```
grep -ch ERROR app.log access.log
```

```
4  
0
```

`-v` counts the lines that do *not* match, so `-vc` is “how many lines are not INFO”. And counting an empty pattern, which matches every line, is a way to count lines:

```
grep -vc INFO app.log
```

```
6
```

```
grep -c ' ' app.log
```

```
16
```

Stopping Early: -m

`-m N` stops after N matching **lines**. The manual says `grep` also stops reading the file at that point, so it should be faster on a huge file when you only need the first few (I did not time it). Context after the last match is still printed:

```
grep -m2 ERROR app.log
```

```
09:00:09 ERROR database connection refused
09:00:15 ERROR database connection refused
```

```
grep -m1 -A1 ERROR app.log
```

```
09:00:09 ERROR database connection refused
09:00:10 INFO  retrying in 5s
```

```
grep -m2 -c ERROR app.log
```

```
2
```

Quiet and Silent: -q and -s

-q prints nothing and only sets the exit status (Chapter 1). The manual says it exits immediately on the first match, which should make it the quickest yes-or-no test (not timed here). What I did check is that a match wins even if another file was missing:

```
grep -q ERROR app.log nosuch; echo status $?
```

```
status 0
```

-s (silent) hides the **error messages about missing or unreadable files**. Compare the two runs below, where one file does not exist:

```
grep ERROR nosuch app.log
```

```
grep: nosuch: No such file or directory
app.log:09:00:09 ERROR database connection refused
app.log:09:00:15 ERROR database connection refused
app.log:09:00:16 ERROR giving up after 2 attempts
app.log:09:15:00 ERROR disk usage 95%, ERROR threshold 90%
```

```
grep -s ERROR nosuch app.log
```

```
app.log:09:00:09 ERROR database connection refused
app.log:09:00:15 ERROR database connection refused
app.log:09:00:16 ERROR giving up after 2 attempts
app.log:09:15:00 ERROR disk usage 95%, ERROR threshold 90%
```

The messages are gone, but the exit status is not changed: a missing file is still an error (status 2). And `-s` hides *only* file errors. A bad pattern still complains:

```
grep -s ERROR nosuch; echo status $?
```

```
status 2
```

```
grep -sE '(' app.log; echo status $?
```

```
grep: Unmatched ( or \(\nstatus 2
```

A script that wants a count and wants to know whether the file could be read can use both. With a missing file, `-c` prints nothing at all (not `0`), so the status is the only clue:

```
n=$(grep -sc ERROR nosuch); echo "$n status $?"
```

```
[] status 2
```

(`2>/dev/null` also hides the messages, but it hides *everything* grep says on standard error, including a bad pattern, so `-s` is the more precise choice.)

Colour

`--color=auto` highlights the match when the output goes to a terminal, and does nothing when it goes to a pipe or a file, so it is safe to leave on. `--color=always` forces the colour codes

in, even when the output is not a terminal. Piped through `cat -v`, which shows the invisible characters, here is the difference:

```
grep --color=auto ERROR app.log | head -1 | cat -v
```

```
09:00:09 ERROR database connection refused
```

```
grep --color=always -n ERROR app.log | head -1 | cat -v
```

```
^[[32m^[[K5^[[m^[[K^[[36m^[[K:^[[m^[[K09:00:09 ^[[01;31m^[[KERROR^[[m^[[K database conne
```

The `^[[01;31m` parts are the escape codes. In a terminal they make the word red; in a file or another command's input they are junk that can break the next step. Use `--color=auto` (many people put `alias grep='grep --color=auto'` in their shell startup file) and avoid `always` except when you want colour through a pager such as `less -R`.

Counting Things: `-o`, `sort` and `uniq -c`

The most useful combination in the chapter. `-o` pulls out the thing you care about, `sort` groups equal items together, `uniq -c` counts each group, and `sort -rn` puts the biggest first. How many of each level are in the log?

```
grep -oE 'INFO|WARN|ERROR' app.log | sort | uniq -c | sort -rn
```

```
10 INFO
 5 ERROR
 2 WARN
```

(Ten INFO, five ERROR, two WARN. The five ERROR is matches, as the `-o` counted them, not the four lines.) Which clients made the most requests?

```
grep -oE '^[0-9.]+' access.log | sort | uniq -c | sort -rn
```

```
4 10.0.0.5
3 10.0.0.7
2 10.0.0.9
```

And what status codes did the server return? Here there is no need to sort by size:

```
grep -oE ' [0-9]{3}$' access.log | sort | uniq -c
```

```
5 200
2 302
2 404
```

`uniq` only joins *neighbouring* equal lines, which is why `sort` comes first. Leave it out and the counts will be wrong with no error to warn you.

Hands-On Exercises

EXERCISE 1

Show every `ERROR` in `app.log` with one line before and two lines after, line numbers on, and a separator of `-----`. Then say which printed lines are matches and which are context, and why there are two groups, not four.

 [View solution](#)

EXERCISE 2

From `app.log` report, labelled: the number of lines with `ERROR`, the number of `ERROR` matches, the number of lines without `ERROR`, and how many of each level there are, biggest first.

 [View solution](#)

EXERCISE 3

Write `errcount.sh FILE` that prints how many lines contain `ERROR`, saying so plainly for a file with none and exiting with 2 (and a message of its own) for a file that cannot be read. Try it on `app.log`,

`access.log` and a missing file. Then list the two busiest client addresses in `access.log`.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- `-A N` after, `-B N` before, `-C N` or `-N` both sides; overlapping context is printed once; non-adjacent groups are separated by `--` (`--group-separator=TEXT`, `--no-group-separator`)
- With `-n` or file names: `:` marks a matching line, `-` marks a context line
- `-o` with context prints the `--` separators but no context; `-c` ignores context
- `-n` line number; `-b` byte offset of the line (of the match itself with `-o`); `-H` / `-h` always / never show the file name; `--label=NAME` names standard input
- `-o`: one output line per match; `-c`: count of *lines* (use `-o | wc -l` to count matches); `-c` prints `0` for a file with no match and nothing for an unreadable one
- `-m N`: stop after N matching lines (context after the last one is still printed; `grep` stops reading)
- `-q` status only, exits at the first match, a match beats a missing file; `-s` hides file errors only (status unchanged, a bad pattern still reports)
- `--color=auto` is safe; `always` puts escape codes into pipes and files
- Counting things: `grep -o PAT | sort | uniq -c | sort -rn`; `uniq` needs sorted input

COMING NEXT

grep: Searching Text 5 introduces Perl-style patterns with `-P`: `\d`, lazy matching, lookahead and lookbehind, `\K`, and where it is not available.

grep: Searching Text

Chapter 5 · Perl Regex with -P

The basic and extended patterns of Chapter 3 cover most searches. `-P` switches grep to **Perl-compatible regular expressions** (PCRE), the same pattern language as Perl, PHP, and (with small differences) Python and JavaScript. It adds things the other two do not have: `\d` for a digit, matching as *little* as possible, checking what comes before or after without including it, and throwing away the start of a match. It also has real costs, which this chapter is just as keen on.

RUN ON TWO VERSIONS OF GREP

Every example was run on GNU grep 3.0 (Git Bash) and GNU grep 3.11 (WSL). The only differences were the wording of two error messages, shown where they come up. `-P` is a GNU feature. It worked on both versions here; macOS/BSD grep is often built without it, which I did not try, so check before relying on it in a script that has to run elsewhere.

Practice File

Paste this into an empty folder. It makes `text.txt`: orders and dates, an HTML line, quoted strings, a tab, and a line with accented letters.

```
cat > text.txt <<'EOF'
order 1042 shipped on 2026-10-08 to bob@example.com
order 7 shipped on 2026-09-30 to alice@work.org
price: $19.99 (was $25.00)
<b>bold</b> and <i>italic</i> text
user=bob id=42 role=admin
user=alice id=7 role=user
foo123bar456
    leading spaces here
EOF
printf 'tabs\tthere\n' >> text.txt
cat >> text.txt <<'EOF'
name="Bob" city="Paris"
say "hello" and "bye"
color colour
the the end
```

```
Price 100 USD, 250 EUR, 75 USD, $30 USD
café résumé naïve
EOF
```

Shorthands That Chapter 3 Did Not Have

In Perl mode `\d` is a digit (Chapter 3 warned that in basic and extended it is not), `\s` is any white space, `\w` a word character, and their capitals mean the opposite (`\D`, `\S`, `\W`). `\t` is a tab and `\h` is a horizontal space (space or tab). Counts and groups are written the extended way, with no backslashes.

```
grep -oP '\d{4}-\d{2}-\d{2}' text.txt
```

```
2026-10-08
2026-09-30
```

```
grep -oP '\w+@\w+\.\w+' text.txt
```

```
bob@example.com
alice@work.org
```

```
grep -P '\t' text.txt | cat -A
```

```
tabs^Ihere$
```

```
grep -P '^ \h' text.txt | cat -A
```

```
leading spaces here$
```

(`cat -A` shows a tab as `^I` and the end of the line as `$`, so you can see the invisible characters.) None of this needed `-P` in principle (`[0-9]` and `[:space:]` do the same job); the shorthands are shorter and read better.

Greedy and Lazy

Chapter 3 showed that `.*` takes as much as it can. Here is the same trap with HTML. The greedy pattern starts at the first `<` and runs to the last `>`:

```
grep -o '<.*>' text.txt
```

```
<b>bold</b> and <i>italic</i>
```

Add a `?` after the repeat and it becomes **lazy**: it takes as *little* as it can and still match. That gives each tag separately:

```
grep -oP '<.*?>' text.txt
```

```
<b>  
</b>  
<i>  
</i>
```

The lazy forms are `*?`, `+?`, `??` and `{n,m}?`. They matter whenever you take a piece out from between two delimiters. The same with quoted strings: greedy swallows from the first quote to the last, lazy takes one string at a time:

```
grep -o '".*"' text.txt
```

```
"Bob" city="Paris"  
"hello" and "bye"
```

```
grep -oP '".*?'" text.txt
```

```
"Bob"  
"Paris"  
"hello"  
"bye"
```

Lazy repeats can also surprise you. `\d+` takes a whole number, while `\d+?` takes as little as is allowed, one digit:

```
echo 'a1b22c333' | grep -oP '\d+'
```

```
1
22
333
```

```
echo 'a1b22c333' | grep -oP '\d+?'
```

```
1
2
2
3
3
3
```

YOU OFTEN DO NOT NEED -P FOR THIS

A *negated set* does the same job in any grep. “A tag” is `<`, then anything that is not `>`, then `>`. The output is identical, and it also works where `-P` does not exist:

```
grep -o '<[^>]*>' text.txt
```

```
<b>
</b>
<i>
</i>
```

```
grep -o '"[^"]*"' text.txt
```

```
"Bob"
"Paris"
"hello"
"bye"
```

Reach for `-P` and a lazy repeat when the closing delimiter is more than one character, or when a negated set cannot say what you mean.

Looking Around: Lookahead and Lookbehind

A **lookaround** checks what is next to the match, without making it part of the match. That is exactly what you want with `-o`: find the number, but only where it is followed by `USD`, and print just the number.

WRITE	MEANS
<code>X(?:=Y)</code>	X, when Y follows (lookahead)
<code>X(?:!Y)</code>	X, when Y does not follow
<code>(?<=Y)X</code>	X, when Y comes before (lookbehind)
<code>(?<!Y)X</code>	X, when Y does not come before

```
grep -oP '\d+(?= USD)' text.txt
```

```
100
75
30
```

```
echo 'Price 100 USD, 250 EUR, 75 USD' | grep -oP '\b\d+\b(?:! USD)'
```

```
250
```

```
grep -oP '(?<=\$)\d+\.\d+' text.txt
```

```
19.99
25.00
```

The first finds 100, 75 and 30 (the numbers followed by `USD`); the second the one number that is not (250 is in euros); the third the digits after a dollar sign. Combine them. Numbers in USD,

but not those that already have a dollar sign in front (the `$30 USD` at the end):

```
grep -oP '(?<!\$)\b\d+(?= USD)' text.txt
```

```
100
```

```
75
```

Two small traps

A lookahead must have a fixed length. It has to know how far back to look. `\w+` could be any length, so grep refuses:

```
grep -oP '(?<=\w+)\w+' text.txt; echo status $?
```

```
grep: lookahead assertion is not fixed length
status 2
```

`\K` is the way round that. It means “forget everything matched so far”: the text before it must still match, but it is left out of what is printed, with no length restriction:

```
grep -oP '\w+@\K\w+' text.txt
```

```
example
work
```

```
grep -oP 'id=\K\d+' text.txt
```

```
42
```

```
7
```

`\K` and a lookahead together answer “the user name, but only for the admin”. The name is what we want; the rest of the line only decides:

```
grep -oP 'user=\K\w+(?=.*role=admin)' text.txt
```

```
bob
```

A lookahead on its own matches nothing visible. `(?=foo)` matches the *position* before a `foo`, with no characters, so the line counts as a match (status 0) but `-o` has nothing to print:

```
echo foo | grep -oP '(?=foo)'; echo status $?
```

```
status 0
```

Alternation: First Wins, Not Longest

Chapter 3 said that basic and extended grep take the **longest** match. Perl mode takes the **first alternative that works**, so the order you write them in matters:

```
echo foobar | grep -oP 'foo|foobar'
```

```
foo
```

```
echo foobar | grep -oE 'foo|foobar'
```

```
foobar
```

```
echo foobar | grep -oP 'foobar|foo'
```

```
foobar
```

The first is Perl mode, the second the same pattern in extended mode, the third Perl mode with the longer alternative written first. When you move a pattern between `-E` and `-P`, check any alternation.

More PCRE Features

`(?i)` inside the pattern switches case-insensitive matching on from that point (you may know it as an inline flag), which is handy when only part of the pattern should ignore case. A back-

reference works as in Chapter 3, and `\Q ... \E` quotes a stretch of the pattern so that its characters are plain text:

```
grep -oP '(?i)price' text.txt
```

```
price
Price
```

```
grep -oP '\b(\w+) \1\b' text.txt
```

```
the the
```

```
grep -oP '\Qprice: $\E\d+' text.txt
```

```
price: $19
```

`(?:...)` is a group that does not capture (it does not take a number for `\1`), and PCRE also has named groups and possessive repeats. They are real, but with `grep` you can only get the *whole* match out, so they rarely earn their place here.

Limits of -P

One pattern only

`-e` twice, or `-f` with several lines, is refused. Join them with `|` inside the one pattern instead:

```
grep -P -e 'a' -e 'b' text.txt; echo status $?
```

```
grep: the -P option only supports a single pattern
status 2
```

Error messages

An invalid pattern gives a message and status 2, as before. The wording comes from the Perl-regex library and differs by version:

```
grep -P '(' text.txt; echo status $?
```

```
grep: missing )  
status 2
```

That is grep 3.0. In 3.11 the message is `grep: missing closing parenthesis`. Do not match on the text of a grep error message in a script.

Slow patterns

Perl-style matching works by trying one way and **backing up** when it fails. A pattern with a repeat inside a repeat, such as `(a+)+`, can have an enormous number of ways to fail. Here is a line of 30 letters `a` followed by a `b`:

```
head -c 30 /dev/zero | tr '\0' 'a' > aaa.txt; echo b >> aaa.txt
```

```
grep -P '^(a+)+$' aaa.txt; echo status $?
```

```
grep: exceeded PCRE's backtracking limit  
status 2
```

grep gave up with an **error** (status 2), not a “no match” (status 1). In 3.11 the message is `grep: aaa.txt: exceeded PCRE's backtracking limit`, with the file name. So an error from `-P` means “I could not decide”, and a script that treats every non-zero status as “no match” will be wrong. Extended mode answered this particular pattern without trouble:

```
grep -E '^(a+)+$' aaa.txt; echo status $?
```

```
status 1
```

That answered at once, with a plain “no match”. If you can write the pattern without a repeat inside a repeat (here, simply `^a+$`), do. Both of these returned in a fraction of a second here

because grep enforces a limit; I did not measure a case without one.

Several Lines at Once: -z

grep matches one line at a time, so a pattern cannot cross a line break. `-z` changes the separator from a newline to a zero byte, so a text file with no zero bytes becomes *one* long “line”. Then `(?s)` lets `.` match a newline too. The output ends with a zero byte instead of a newline, so `tr` turns them back. First without `(?s)`, where `.` stops at the line break and nothing matches:

```
printf 'start\nmiddle\nend\n' | grep -Pzo 'start.*end' | tr '\0' '\n'
```

(no output)

```
printf 'start\nmiddle\nend\n' | grep -Pzo '(?s)start.*end' | tr '\0' '\n'
```

```
start
middle
end
```

The lazy `.*?` ends at the first `end`; a greedy `.*` would run to the last one in the file.

Accented Letters

The last line of the practice file is `café résumé naïve`. In Perl mode `\w` is **ASCII only**: it stops at the accented letters and breaks each word into pieces:

```
tail -1 text.txt | grep -oP '\w+'
```

```
caf
r
sum
na
ve
```

Two ways to ask for Unicode letters in PCRE are `(*UCP)` at the start of the pattern, which makes `\w` and friends Unicode-aware, and `\p{L}`, which means “any letter”:

```
tail -1 text.txt | grep -oP '(*UCP)\w+'
```

```
café  
résumé  
naïve
```

```
tail -1 text.txt | grep -oP '\p{L}+'
```

```
café  
résumé  
naïve
```

I confirmed the bytes were identical on both grep versions (WSL's own display of the accents was garbled in my capture, so I compared the output as hex). This depends on a UTF-8 locale; both machines had one. If your text is not English, test your own words before trusting `\w`.

Choosing Between -E and -P

USE <code>-E</code> WHEN...	USE <code>-P</code> WHEN...
the script must run on other machines	you work on GNU/Linux and need lookaround, <code>\K</code> or lazy repeats
several patterns (<code>-e</code> , <code>-f</code>)	one pattern is enough
you want the longest match and no surprises	you want first-match behaviour, as in Perl, PHP or Python
the pattern has a repeat inside a repeat	it does not

Hands-On Exercises

EXERCISE 1

From the user lines, print the `id` number of every user, then the name of the user whose role is `user` (not admin), using `\K` and a lookahead. Then explain why the same task cannot be done with a lookbehind.

 [View solution](#)

EXERCISE 2

A file has the lines `name="Bob" city="Paris"`, `say "hello" and "bye"` and a line with an escaped quote. Extract each quoted string with a greedy pattern, a lazy one and a negated set, and say which lines each gets right.

 [View solution](#)

EXERCISE 3

From a price list, print the numbers in USD that do not have a dollar sign in front, the amounts that are in a currency other than USD, and then show that a pattern with a repeat inside a repeat gives an error rather than “no match”, and rewrite it.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- `-P` = Perl-compatible regex: GNU only, **one pattern** (`-e` twice is refused: use `|`), check availability before relying on it
- `\d \D \s \S \w \W \t \h`; counts and groups without backslashes; `(?i)` inline flag; `\Q...\E` literal text; `(?:...)` non-capturing
- Lazy repeats: `*? +? ?? {n,m}?`; a negated set (`<[>]*>`) often does the same job in any grep
- Lookahead `(?=Y) (?!Y)`, lookbehind `(?<=Y) (?<!Y)` (fixed length only); `\K` drops what came before, with no length limit; a bare lookaround matches an empty string, so `-o` prints nothing
- Alternation takes the **first** alternative that works, not the longest (the opposite of `-E`)
- `-z` + `(?s)` matches across lines; pipe through `tr '\0' '\n'`
- `\w` is ASCII only; `(*UCP)` or `\p{L}` for Unicode letters (UTF-8 locale)

- A repeat inside a repeat can exhaust grep's backtracking limit: that is an **error (status 2)**, not "no match"; error wording differs by version

COMING NEXT

grep: Searching Text 6 puts grep to work in pipelines and scripts: `if grep -q`, the `set -e` and `pipefail` traps, the `[n]ginx` trick, comparing two lists with `-Fxf`, and changing files safely after finding them.

grep: Searching Text

Chapter 6 · grep in Pipelines and Scripts

At the prompt, `grep` prints something and you look at it. In a script, nobody is looking: the **exit status** is the answer, and the next command depends on it. That is where `grep`'s “no match is not an error, but it is not success either” behaviour (Chapter 1) starts to bite. This chapter is mostly the traps, each shown failing and then fixed.

RUN ON TWO VERSIONS OF GREP

Every example ran in a fresh copy of the practice folder, on GNU `grep` 3.0 (Git Bash) and GNU `grep` 3.11 (WSL), and the output matched on both. The two exceptions: the `ps` example needs a real Linux process list, so it ran on WSL (3.11) only, and the timing example is compared by seconds, not by exact equality. I did not try macOS/BSD `grep`, and several scripts use bash features (`<<<`, `<( )`), so run them with bash, not plain `sh`.

Practice Folder

Paste this into an empty folder. Each example below is independent: it assumes this folder, freshly made.

```
cat > app.log <<'EOF'
09:00:01 INFO  service starting
09:00:09 ERROR database connection refused
09:00:10 INFO  retrying in 5s
09:00:16 ERROR giving up after 2 attempts
09:00:20 INFO  listening on port 8080
EOF
```

```
cat > access.log <<'EOF'
10.0.0.5 GET /index.html 200
10.0.0.7 GET /about.html 200
10.0.0.5 POST /login 302
10.0.0.9 GET /missing 404
10.0.0.5 GET /index.html 200
10.0.0.7 GET /missing 404
EOF
```

```
printf 'alice\nbob\ncarol\ndave\n' > list_a.txt
printf 'bob\ndave\nerin\nbob\n' > list_b.txt

mkdir -p site/css
printf 'the colour red\nanother colour\n' > site/index.txt
printf 'body { colour: blue }\n' > site/css/s.txt
printf 'my colour file\n' > "site/my notes.txt"
```

grep as a Question: if grep -q

The usual shape is `if grep -q PATTERN FILE`. `-q` keeps the screen clean; the status is the answer.

```
if grep -q ERROR app.log; then echo 'has errors'; else echo 'clean'; fi
```

```
has errors
```

```
if ! grep -q CRITICAL app.log; then echo 'no critical errors'; fi
```

```
no critical errors
```

`&&` and `||` read the same status. Be careful with `a && b || c` as a stand-in for if/else: `c` runs also if `b` fails. For anything beyond one short command, use `if`.

```
grep -q ERROR app.log && echo 'A: found' || echo 'B: not found'
```

```
A: found
```

To ask about text you already have in a variable, feed it in with a **here-string** (`<<<`, bash) instead of `echo "$var" | grep`:

```
line='status: ERROR in module x'
if grep -q ERROR <<< "$line"; then echo yes; fi
```

```
yes
```

Trap 1: set -e and the Status 1

Many scripts start with `set -e`, which means “stop the script the moment any command fails”. To the shell, `grep` finding nothing *is* a failure (status 1), so the script stops at the `grep`, **silently**. Nothing is printed; the script's own exit status is 1:

```
cat > s1.sh <<'EOF'
set -e
echo "before"
grep -q CRITICAL app.log
echo "after"
EOF
bash s1.sh; echo "script exit status: $?"
```

```
before
script exit status: 1
```

Three ways out. Add `|| true` (“and if that fails, carry on”), or put the `grep` in an `if` condition, where `set -e` deliberately does not apply:

```
cat > s2.sh <<'EOF'
set -e
echo "before"
grep -q CRITICAL app.log || true
echo "after"
EOF
bash s2.sh; echo "script exit status: $?"
```

```
before
after
script exit status: 0
```

```
cat > s3.sh <<'EOF'
set -e
if grep -q CRITICAL app.log; then echo "found"; else echo "none"; fi
echo "after"
EOF
bash s3.sh; echo "script exit status: $?"
```

```
none
after
script exit status: 0
```

Pick the first when you do not care about the answer, and the second when you do. The same trap applies to every grep whose result you assign or pipe, which brings us to counting:

Trap 2: Counting Under set -e

`grep -c` prints `0` and exits with status 1 when nothing matches (Chapter 4). In a command substitution under `set -e`, that kills the script before it can print the 0:

```
cat > s4.sh <<'EOF'
set -e
n=$(grep -c CRITICAL app.log)
echo "critical lines: $n"
EOF
bash s4.sh; echo "script exit status: $?"
```

```
script exit status: 1
```

```
cat > s5.sh <<'EOF'
set -e
n=$(grep -c CRITICAL app.log || true)
echo "critical lines: $n"
EOF
bash s5.sh; echo "script exit status: $?"
```

```
critical lines: 0
script exit status: 0
```

The fix is `|| true` **after** the grep. The tempting `|| echo 0` is wrong, because grep has already printed its own 0, so the variable ends up with two lines:

```
n=$(grep -c CRITICAL app.log || echo 0)
printf "[%s]\n" "$n"
```

```
[0  
0]
```

If you want a count and no fuss, `awk` counts without the status problem, because `awk` only reports an error for a real error:

```
awk '/CRITICAL/ { n++ } END { print n + 0 }' app.log; echo "awk exit status: $?"  
  
0  
awk exit status: 0
```

Trap 3: pipefail

Normally a pipeline's status is that of its **last** command, so a failure in the middle is invisible.

`set -o pipefail` changes that: the pipeline fails if *any* command in it fails. Combined with `set -e`, a `grep` that finds nothing now stops the script, even though `wc` at the end succeeded and printed its 0:

```
cat > s6.sh <<'EOF'  
set -eo pipefail  
grep CRITICAL app.log | wc -l  
echo "after"  
EOF  
bash s6.sh; echo "script exit status: $?"  
  
0  
script exit status: 1
```

Again, `|| true` or an `if` fixes it, or `awk` in place of `grep`.

A stranger one: -q and SIGPIPE

`-q` stops at the first match. If the command feeding `grep` is still writing, it gets a **broken pipe** signal and dies, which `pipefail` then reports as status `141` (128 plus the signal number 13), *even though grep found the answer*:

```
bash -c 'set -o pipefail; seq 1 200000 | grep -q 5; echo "pipeline status: $?"'
```

```
pipeline status: 141
```

The same pipeline without pipefail, and with pipefail but without `-q` (grep then reads everything, so nothing is cut off), both say 0:

```
bash -c 'seq 1 200000 | grep -q 5; echo "pipeline status without pipefail: $?"'
```

```
pipeline status without pipefail: 0
```

```
bash -c 'set -o pipefail; seq 1 200000 | grep 5 > /dev/null; echo "pipeline status, no -q: $?"'
```

```
pipeline status, no -q: 0
```

If the producer is a command, you can also read it through process substitution, which does not take part in the pipeline's status (bash):

```
bash -c 'set -o pipefail; grep -q 5 < <(seq 1 200000); echo "status with process substitution: $?"'
```

```
status with process substitution: 0
```

A status of 141 is not a match failing; treat it as “grep stopped early”. The cure is any of: no `-q`, process substitution, or leave pipefail off for that line.

Patterns From Variables

A pattern that arrives in a variable (a user's input, a file name) should be treated as *text*, not as a regular expression. Use `-F`, and `--` to keep a value that starts with a dash from being read as an option, and quote the variable. Without `-F` the dot below matches the `x` too:

```
pat='a.b'
echo 'axb a.b' | grep -o -- "$pat"
```

```
axb
a.b
```

```
pat='a.b'
echo 'axb a.b' | grep -oF -- "$pat"
```

```
a.b
```

Finding a Process

`ps` lists every process, and `grep` filters it, but the `grep` you are running is *also* a process whose command line contains the word you are looking for, so it finds itself. (This ran on WSL, with a background `sleep 300` standing in for the process you want.)

```
export COLUMNS=200 LINES=50 # only to stop ps complaining about the screen size when i
sleep 300 &
pid=$!
sleep 0.3
echo '$ ps -eo args | grep sleep'
ps -eo args | grep sleep
echo '$ ps -eo args | grep [s]leep'
ps -eo args | grep '[s]leep'
echo '$ pgrep -xc sleep'
pgrep -xc sleep
kill $pid; wait $pid 2>/dev/null; true
```

```
$ ps -eo args | grep sleep
sleep 300
grep sleep
$ ps -eo args | grep [s]leep
sleep 300
$ pgrep -xc sleep
1
```

The trick is the bracket expression: `[s]leep` matches the text `sleep`, but the `grep` process's own command line contains the literal text `[s]leep`, which it does not match. `pgrep`, where it exists (here, WSL had it and Git Bash did not), does the job without the trick; `-x` asks for an exact name and `-c` counts.

Comparing Two Lists

`-f FILE` takes the patterns from a file. With `-F` (plain text) and `-x` (whole line) the patterns become “exactly these lines”, which turns `grep` into a set tool, and neither file needs sorting. Our lists: `list_a.txt` has alice, bob, carol, dave; `list_b.txt` has bob, dave, erin, bob.

```
grep -Fxf list_a.txt list_b.txt
```

```
bob
dave
bob
```

The lines of `list_b.txt` that are also in `list_a.txt`. Note that bob appears twice: `grep` prints the lines of the *file being searched* as they are, duplicates included. `-v` turns it into a difference:

```
grep -vFxf list_a.txt list_b.txt
```

```
erin
```

```
grep -Fxf list_b.txt list_a.txt
```

```
bob
dave
```

That is “in b but not in a”, and the lines of a that are also in b. Swap the files for the other direction. `comm` does the same job and removes duplicates, but only on sorted input:

```
comm -12 <(sort list_a.txt) <(sort list_b.txt)
```

```
bob
dave
```

A BLANK LINE IN THE PATTERN FILE

An empty pattern matches **every** line. If the pattern file ends with a blank line (an easy accident), `-Ff` without `-x` matches everything and the comparison silently becomes meaningless. With `-x` a blank pattern can only match a blank line:

```
printf 'bob\n\n' > pats.txt
echo "without -x (a blank pattern line matches every line):"
grep -Ff pats.txt list_a.txt
echo "with -x:"
grep -Fxf pats.txt list_a.txt
```

```
without -x (a blank pattern line matches every line):
alice
bob
carol
dave
with -x:
bob
```

Find, Then Change: grep and sed

A common job: find every file with a word and change it. The safe version has four habits: preview first, keep backups, handle awkward file names, and never touch the backups themselves. The practice folder has three files that say `colour`, one of them with a space in its name. Preview, printing only the lines that would change:

```
grep -rlZ colour site | xargs -0 sed -n 's/colour/color/gp'
```

```
body { color: blue }
the color red
another color
my color file
```

Then apply it, with `-i.bak` (GNU sed edits in place and keeps the original with a `.bak` ending), and check:

```
grep -rIz colour site | xargs -0 sed -i.bak 's/colour/color/g'
echo "still saying colour:"; grep -rn colour site --exclude='*.bak'
echo "now saying color:";   grep -rn color site --exclude='*.bak'
echo "backups:";           find site -name '*.bak' | sort
```

```
still saying colour:
now saying color:
site/css/s.txt:1:body { color: blue }
site/index.txt:1:the color red
site/index.txt:2:another color
site/my notes.txt:1:my color file
backups:
site/css/s.txt.bak
site/index.txt.bak
site/my notes.txt.bak
```

`-lZ` and `xargs -0` (Chapter 2) kept `site/my notes.txt` in one piece. Without them it breaks:

```
grep -rI colour site | xargs sed -n 's/colour/color/gp'
```

```
sed: can't read site/my: No such file or directory
sed: can't read notes.txt: No such file or directory
body { color: blue }
the color red
another color
```

`sed` was asked for `site/my` and `notes.txt`, neither of which exists, and it reported that it could not read them. (It still did the files with ordinary names, which is why the message is easy to miss.)

The backups bite back

The `.bak` files still say `colour`, so a **second run** finds them, edits them, and makes backups of the backups:

```
grep -rIz colour site | xargs -0 sed -i.bak 's/colour/color/g'
grep -rIz colour site | xargs -0 sed -i.bak 's/colour/color/g'
```

```
find site -name '*.bak*' | sort
```

```
site/css/s.txt.bak
site/css/s.txt.bak.bak
site/index.txt.bak
site/index.txt.bak.bak
site/my notes.txt.bak
site/my notes.txt.bak.bak
```

Keep the backups out of the search with `--exclude`. Now the second run has nothing to find, which brings up another trap: `xargs` still runs `sed` **once, with no files**, and `sed` complains:

```
grep -rIZ colour site --exclude='*.bak' | xargs -0 sed -i.bak 's/colour/color/g'
echo "second run:"
grep -rIZ colour site --exclude='*.bak' | xargs -0 sed -i.bak 's/colour/color/g'
```

```
second run:
sed: no input files
```

GNU `xargs` has `-r` (`--no-run-if-empty`) for this: with no input it runs nothing. (BSD `xargs`, as on macOS, already behaves that way, and I did not try it.) With both fixes, running the edit twice is harmless:

```
grep -rIZ colour site --exclude='*.bak' | xargs -0 -r sed -i.bak 's/colour/color/g'
grep -rIZ colour site --exclude='*.bak' | xargs -0 -r sed -i.bak 's/colour/color/g'
find site -name '*.bak*' | sort
```

```
site/css/s.txt.bak
site/index.txt.bak
site/my notes.txt.bak
```

To undo the change, move each backup back over its file (zero-byte separators again, so spaces are safe):

```
grep -rIZ colour site | xargs -0 sed -i.bak 's/colour/color/g'
find site -name '*.bak' -print0 | while IFS= read -r -d '' f; do mv -- "$f" "${f%.bak}";
```

```
grep -rn colour site
find site -name '*.bak' | wc -l
```

```
site/css/s.txt:1:body { colour: blue }
site/index.txt:1:the colour red
site/index.txt:2:another colour
site/my notes.txt:1:my colour file
0
```

That last `0` is the count of backups left. Always run on a clean working tree, or a copy: in a project under git, `git diff` is also a way back, and the `.git` folder should be excluded (`--exclude-dir=.git`) so the edit does not touch it.

grep Next to cut, awk, sort and uniq

grep chooses *lines*; `cut` and `awk` choose *fields* from them; `sort` and `uniq -c` count. Which clients got a 404?

```
grep ' 404$' access.log | cut -d' ' -f1 | sort | uniq -c
```

```
1 10.0.0.7
1 10.0.0.9
```

```
grep GET access.log | awk '{print $3}' | sort | uniq -c | sort -rn
```

```
2 /missing
2 /index.html
1 /about.html
```

The second picks the GET requests, takes the third field (the path) with `awk`, and counts how often each was asked for. For fixed layouts like this, `awk` alone can do the filtering too (`awk '/GET/ {print $3}'`), which saves a process, but the `grep-then-pick` form is easier to read and to change.

Watching a Live Log: --line-buffered

When `grep`'s output goes to a *pipe*, `grep` collects it in a buffer and sends it in chunks, so a pipeline that is meant to react immediately (`tail -f app.log | grep ERROR | ...`) can sit on a line for a long time. Here the first command sends `ERROR1` at once and then waits two seconds before finishing; the last command prints when each line *reaches it*:

```
SECONDS=0
```

```
( echo ERROR1; sleep 2; echo x ) | grep ERROR | while read l; do echo "$l arrived after :
```

```
ERROR1 arrived after 2s
```

`--line-buffered` makes `grep` pass each line on straight away:

```
SECONDS=0
```

```
( echo ERROR1; sleep 2; echo x ) | grep --line-buffered ERROR | while read l; do echo "$l
```

```
ERROR1 arrived after 0s
```

The first shows `ERROR1` arriving after two seconds, when `grep` ended and emptied its buffer; the second well before that (the clock counts whole seconds, so what matters is the difference). If a later stage in the pipeline also buffers (some do) you will still see delays, and the buffering is a performance feature: use `--line-buffered` when promptness matters more than speed.

Hands-On Exercises

EXERCISE 1

A script starts with `set -euo pipefail` and counts `ERROR` and `CRITICAL` lines in `app.log` with `grep -c`. Show that it dies silently before printing the second count, then fix it two ways, with `|| true` and with `awk`, and say what each fix changes.

 [View solution](#)

EXERCISE 2

Given `required.txt` and `installed.txt`, print what is required and missing, what is installed and not required, and what is in both. The required file ends with an accidental blank line: show how that breaks one of the three answers and how to fix it.

 [View solution](#)

EXERCISE 3

In the `site` folder, rename `colour` to `color` safely: preview, apply with backups, confirm, then run it a second time to show nothing more changes, then restore every file from its backup.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- `if grep -q PAT FILE; then ...`; `if ! grep -q`; here-string `grep -q PAT <<< "$var"` (bash)
- `set -e` + a grep with no match (status 1) = the script stops, silently. Fixes: `|| true`, or use it as an `if` condition
- `n=$(grep -c PAT f || true)` (not `|| echo 0`, which gives `0` twice); `awk '/PAT/ {n++} END {print n+0}'` never fails on zero
- `pipefail`: a grep with no match fails the whole pipeline; `producer | grep -q` can give **141** (SIGPIPE) even on a match: drop `-q`, or use `< <(producer)`
- Variables as patterns: `grep -F -- "$pat"` (text, not regex; a value starting with `-` is safe)
- `ps -eo args | grep '[s]leep'` avoids matching grep itself; `pgrep -x NAME` where available
- Lists: `grep -Fxf A B` (lines of B that are in A, duplicates kept); `grep -vFxf A B` (lines of B not in A); a blank pattern line matches everything unless `-x`
- Bulk edit: preview with `sed -n 's/old/new/gp'`; `grep -rIZ OLD --exclude='*.bak' . | xargs -0 -r sed -i.bak 's/OLD/NEW/g'`; restore with `find -print0 | while read -d ''` and `mv`
- `--line-buffered` when grep's output feeds a pipeline that must react at once

COMING NEXT

grep: Searching Text 7 applies everything to three worked jobs: reading a server log, hunting TODOs and secrets in a codebase, and auditing configuration files for a setting.

grep: Searching Text

Chapter 7 · Real-World Searching

This chapter uses nothing new. It takes everything so far and shows how it is actually used, on three jobs you will meet: reading a server log, hunting through a codebase, and auditing configuration files. In each, the work goes the same way: look at the data, ask a narrow question, read the answer, refine the question. The pipelines get long, but each stage is a single idea from an earlier chapter.

RUN ON TWO VERSIONS OF GREP

Every example ran on GNU grep 3.0 (Git Bash) and 3.11 (WSL) against the same data. The output was identical except for one thing, which you have met: the binary-file message, which is on stdout in 3.0 and stderr in 3.11 and so appears in a different place in a listing. All of the data is made up. The “secrets” are the AWS documentation's own example key and invented words; **never paste a real secret into a terminal, a chat or a document**, which is why these examples print file names and line numbers only.

The Practice Data

Paste this into an empty folder. It makes a request log (`service.log`, plus one rotated and compressed older log), a small project folder (`repo`) and a few configuration files (`etc`).

```
cat > service.log <<'EOF'
2026-10-08 09:14:02 INFO    req=a1f3 GET /index 200 took=35ms
2026-10-08 09:14:05 INFO    req=b2c4 GET /about 200 took=22ms
2026-10-08 09:14:09 INFO    req=c3d5 POST /login 302 took=180ms
2026-10-08 09:14:41 WARN    req=d4e6 GET /search 200 took=1250ms slow query
2026-10-08 09:15:02 INFO    req=e5f7 GET /index 200 took=30ms
2026-10-08 09:15:30 ERROR    req=f6a8 POST /checkout 500 took=2210ms database timeout
2026-10-08 09:15:31 INFO    req=f6a8 retry 1 for checkout
2026-10-08 09:15:34 ERROR    req=f6a8 POST /checkout 500 took=3050ms database timeout
2026-10-08 09:16:10 INFO    req=a7b9 GET /index 200 took=28ms
2026-10-08 09:16:44 ERROR    req=b8c0 GET /missing 404 took=12ms not found
2026-10-08 09:17:03 INFO    req=c9d1 GET /about 200 took=25ms
2026-10-08 09:17:50 WARN    req=d0e2 GET /search 200 took=1800ms slow query
2026-10-08 09:18:12 ERROR    req=e1f3 POST /login 500 took=95ms auth backend down
2026-10-08 09:18:13 ERROR    req=e1f3 POST /login 500 took=90ms auth backend down
```

```

2026-10-08 09:19:55 INFO req=f2a4 GET /index 200 took=33ms
2026-10-08 09:20:07 INFO req=a3b5 GET /index 200 took=31ms
2026-10-08 09:21:20 ERROR req=b4c6 GET /missing 404 took=10ms not found
2026-10-08 09:22:02 INFO req=c5d7 GET /about 200 took=24ms
2026-10-08 09:25:00 INFO req=d6e8 GET /index 200 took=29ms
2026-10-08 09:31:14 WARN req=e7f9 GET /search 200 took=1510ms slow query
EOF
printf '2026-10-08 08:59:58 ERROR req=z9y8 GET /index 500 took=70ms old error\n2026-10-08

mkdir -p repo/src repo/config repo/tests repo/node_modules/lib repo/dist repo/assets repo
cat > repo/src/app.js <<'EOF'
// TODO(alice): handle timeout
const apiKey = "AKIAIOSFODNN7EXAMPLE";
// FIXME validate input
function start() { return legacyParse(input); }
EOF
cat > repo/src/db.py <<'EOF'
# TODO: use a connection pool
password = "hunter2"
# HACK: retry forever
db_host = os.environ["DB_HOST"]
EOF
cat > repo/src/util.js <<'EOF'
// TODO remove legacyParse once v2 ships
function legacyParse(x) { return x; }
module.exports = { legacyParse };
EOF
cat > repo/config/settings.py <<'EOF'
SECRET_KEY = 'not-really-secret-123'
DEBUG = True
EOF
cat > repo/tests/test_db.py <<'EOF'
# TODO add more tests
password = "test-password"
EOF
cat > repo/node_modules/lib/index.js <<'EOF'
// TODO vendored noise
const password = "vendored";
EOF
cat > repo/id_rsa.example <<'EOF'
-----BEGIN RSA PRIVATE KEY-----
FAKEFAKEFAKEFAKEFAKEFAKEFAKEFAKEFAKE
-----END RSA PRIVATE KEY-----
EOF
{ printf '/*TODO minified*/password="min";'; head -c 300 /dev/zero | tr '\0' 'x'; echo;

```

```

printf 'logo\0TODO inside a binary\n' > repo/assets/logo.bin
printf '[core]\n# TODO placeholder in .git\n' > repo/.git/config

mkdir -p etc
cat > etc/sshd_config <<'EOF'
# SSH server configuration
Port 22
PermitRootLogin yes
PasswordAuthentication yes    # TODO tighten

#PermitEmptyPasswords no
X11Forwarding no
EOF
cat > etc/app.prod.ini <<'EOF'
; production
debug = false
log_level = info
secret_key = ${SECRET_KEY}
EOF
cat > etc/app.staging.ini <<'EOF'
; staging
debug = true
log_level = debug
EOF
cat > etc/app.dev.ini <<'EOF'
; development
log_level = debug
EOF

```

The project folder looks like this:

```

repo/
├─ assets/
│   └─ logo.bin          (a binary file that contains the word TODO)
├─ config/
│   └─ settings.py
├─ dist/
│   └─ app.min.js        (minified: one very long line)
├─ node_modules/
│   └─ lib/
│       └─ index.js      (third-party code)
├─ src/
│   └─ app.js

```

```
|   |— db.py
|   └─ util.js
|— tests/
|   └─ test_db.py
|— .git/
|   └─ config
└─ id_rsa.example      (a fake key file)
```

Job 1: Reading a Server Log

Each line of `service.log` has a date, a time, a level, a request id (`req=...`), the request, a status, how long it took and sometimes a message. The first question is always “what is in here?”:

```
grep -oE 'INFO|WARN|ERROR' service.log | sort | uniq -c | sort -rn
```

```
11 INFO
 6 ERROR
 3 WARN
```

Eleven INFO, six ERROR and three WARN. (`-o`, `sort`, `uniq -c`, Chapter 4.) Now the questions you would actually ask.

What went wrong between 09:15 and 09:18?

Because the time is written in a fixed-width form, a window can be a pattern. Minutes 15 to 18 are `1[5-8]`:

```
grep -E '^2026-10-08 09:1[5-8]:[0-9]{2} ERROR' service.log
```

```
2026-10-08 09:15:30 ERROR req=f6a8 POST /checkout 500 took=2210ms database timeout
2026-10-08 09:15:34 ERROR req=f6a8 POST /checkout 500 took=3050ms database timeout
2026-10-08 09:16:44 ERROR req=b8c0 GET /missing 404 took=12ms not found
2026-10-08 09:18:12 ERROR req=e1f3 POST /login 500 took=95ms auth backend down
2026-10-08 09:18:13 ERROR req=e1f3 POST /login 500 took=90ms auth backend down
```

That works when the window lines up with the digits. For an awkward one, such as 09:17:30 to 09:21:10, a pattern gets ugly and `awk` is better, because fixed-width times compare correctly *as*

text. Here the same answer both ways:

```
awk '$2 >= "09:15:00" && $2 <= "09:18:59" && $3 == "ERROR"' service.log
```

```
2026-10-08 09:15:30 ERROR req=f6a8 POST /checkout 500 took=2210ms database timeout
2026-10-08 09:15:34 ERROR req=f6a8 POST /checkout 500 took=3050ms database timeout
2026-10-08 09:16:44 ERROR req=b8c0 GET /missing 404 took=12ms not found
2026-10-08 09:18:12 ERROR req=e1f3 POST /login 500 took=95ms auth backend down
2026-10-08 09:18:13 ERROR req=e1f3 POST /login 500 took=90ms auth backend down
```

Both depend on every line having the same layout. A line with a different date format would slip through either one without a warning.

Follow one request

The request id ties lines together. Request `f6a8` failed, was retried and failed again:

```
grep 'req=f6a8' service.log
```

```
2026-10-08 09:15:30 ERROR req=f6a8 POST /checkout 500 took=2210ms database timeout
2026-10-08 09:15:31 INFO req=f6a8 retry 1 for checkout
2026-10-08 09:15:34 ERROR req=f6a8 POST /checkout 500 took=3050ms database timeout
```

```
grep -c 'req=f6a8' service.log
```

```
3
```

Note the retry line, which says INFO even though it belongs to a failing request: searching only for `ERROR` would have missed that context. When ids could be prefixes of longer ids, use `-w`.

What kinds of thing fail?

Take only the error lines, keep just *method*, *path*, *status*, and count:

```
grep ERROR service.log | grep -oE '[A-Z]+ /[a-z]+ [0-9]{3}' | sort | uniq -c | sort -rn
```

```
2 POST /login 500
2 POST /checkout 500
2 GET /missing 404
```

Two logins failing with 500, two checkouts with 500 and two 404s. The two kinds of 500 matter more than the 404s, and now they are visible at a glance.

What was slow?

Pull out the number after `took=`, keep those over a second, and sort (the `\K` trick from Chapter 5):

```
grep -oP 'took=\K[0-9]+' service.log | awk '$1 > 1000' | sort -rn
```

```
3050
2210
1800
1510
1250
```

Without `-P`, a time of four or more digits is a time of a second or more, and you can see the whole line:

```
grep -E 'took=[0-9]{4,}ms' service.log | cut -c1-80
```

```
2026-10-08 09:14:41 WARN req=d4e6 GET /search 200 took=1250ms slow query
2026-10-08 09:15:30 ERROR req=f6a8 POST /checkout 500 took=2210ms database timeo
2026-10-08 09:15:34 ERROR req=f6a8 POST /checkout 500 took=3050ms database timeo
2026-10-08 09:17:50 WARN req=d0e2 GET /search 200 took=1800ms slow query
2026-10-08 09:31:14 WARN req=e7f9 GET /search 200 took=1510ms slow query
```

(`cut -c1-80` shortens long lines to fit.) Both agree there are five slow requests.

The rotated logs

Old logs are compressed. `zgrep` is grep for `.gz` files: it reads the compressed file without unpacking it, and takes the same options. (It was present on both machines.)

```
zgrep -c ERROR service.log.1.gz
```

```
1
```

```
zgrep -h ERROR service.log service.log.1.gz | wc -l
```

```
7
```

The first counts the errors in yesterday's log. The second searches the current and the rotated log together: seven errors in all. `-h` keeps the file name from being added when there are two files.

Job 2: Hunting Through a Codebase

Searching a project has two problems the log did not: most of the files are not yours (vendored libraries, build output, `.git`) and some of them are not text. First, the search with **no** restrictions, with the lines cut short:

```
grep -rn TODO repo | cut -c1-90
```

```
repo/.git/config:2:# TODO placeholder in .git
Binary file repo/assets/logo.bin matches
repo/dist/app.min.js:1:/*TODO minified*/password="min";xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
repo/node_modules/lib/index.js:1:// TODO vendored noise
repo/src/app.js:1:// TODO(alice): handle timeout
repo/src/db.py:1:# TODO: use a connection pool
repo/src/util.js:1:// TODO remove legacyParse once v2 ships
repo/tests/test_db.py:1:# TODO add more tests
```

Eight hits for TODO. Four are the team's own notes in the source and the tests; the other four are noise: the `.git` folder, third-party code in `node_modules`, a binary file and the generated bundle in `dist`. (Under 3.11 the binary line reads `grep: repo/assets/logo.bin: binary file matches` and comes first, because it is on standard error.) Apply the habits from Chapter 2 (`-I`, `--exclude-dir`) and look for the markers as whole words:

```
grep -rnIE --exclude-dir=.git --exclude-dir=node_modules '\b(TODO|FIXME|HACK)\b' repo |
```

```
repo/dist/app.min.js:1:/*TODO minified*/password="min";xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
repo/src/app.js:1:// TODO(alice): handle timeout
repo/src/app.js:3:// FIXME validate input
repo/src/db.py:1:# TODO: use a connection pool
repo/src/db.py:3:# HACK: retry forever
repo/src/util.js:1:// TODO remove legacyParse once v2 ships
repo/tests/test_db.py:1:# TODO add more tests
```

The first line is the minified file in `dist`: it is generated, it is one huge line, and it is not worth reading. (This is why `cut` is at the end of the pipeline: without it that single line would fill the screen.) Exclude the build folder as well:

```
grep -rnIE --exclude-dir=.git --exclude-dir=node_modules --exclude-dir=dist '\b(TODO|FIXM
```

```
repo/src/app.js:1:// TODO(alice): handle timeout
repo/src/app.js:3:// FIXME validate input
repo/src/db.py:1:# TODO: use a connection pool
repo/src/db.py:3:# HACK: retry forever
repo/src/util.js:1:// TODO remove legacyParse once v2 ships
repo/tests/test_db.py:1:# TODO add more tests
```

Six notes (four TODO, one FIXME and one HACK) in four files. Count them by kind, and by file:

```
grep -rhoIE --exclude-dir=.git --exclude-dir=node_modules --exclude-dir=dist '\b(TODO|FIXM
```

```
1 FIXME
1 HACK
4 TODO
```

```
grep -rcIE --exclude-dir=.git --exclude-dir=node_modules --exclude-dir=dist '\b(TODO|FIXM
```

```
repo/src/app.js:2
repo/src/db.py:2
```

```
repo/src/util.js:1
repo/tests/test_db.py:1
```

When every line is long, `-o` with a bounded tail shows the note and a little after it. `.\{0,15\}` means “up to 15 more characters”:

```
grep -rnI --exclude-dir=.git --exclude-dir=node_modules -o 'TODO.\{0,15\}' repo
```

```
repo/dist/app.min.js:1:TODO minified*/pass
repo/src/app.js:1:TODO(alice): handle
repo/src/db.py:1:TODO: use a connect
repo/src/util.js:1:TODO remove legacyP
repo/tests/test_db.py:1:TODO add more tests
```

Looking for secrets

A search for secrets finds *candidates*, not proof. Every search below prints only **file and line** (`cut -d: -f1,2` removes the rest of the line), so the secret itself never lands on your screen or in your shell history's output. A known key format first, an AWS access key id (`AKIA` and sixteen capitals and digits):

```
grep -rnIE --exclude-dir=.git --exclude-dir=node_modules 'AKIA[0-9A-Z]{16}' repo | cut -d: -f1,2
```

```
repo/src/app.js:2
```

Then the shape of an assignment, `password = "something"` or `secret_key = '...'`, in either kind of quote, any case:

```
grep -rniE --exclude-dir=.git --exclude-dir=node_modules '(password|secret_key)[[:space|
```

```
repo/config/settings.py:1
repo/dist/app.min.js:1
repo/src/db.py:2
repo/tests/test_db.py:2
```

Four hits, and they are not equally interesting. `settings.py` is a real candidate. `db.py` too, though its *other* line reads the host from the environment, which is the right pattern and does not match. The one in `tests` is probably a harmless test value, and the one in `dist` is the minified bundle again. Someone has to look at each. The pattern also misses a password built another way, so finding nothing proves nothing.

A private key announces itself with a header line. It starts with dashes, so the pattern must be given with `-e` (Chapter 1), and `-l` is enough since you want to know *which file*:

```
grep -rLI --exclude-dir=.git --exclude-dir=node_modules -e '-----BEGIN [A-Z ]*PRIVATE KEY'
```

```
repo/id_rsa.example
```

Who still uses the old function?

Before removing `legacyParse`, find every use. Whole word, only JavaScript files, and not the vendored code:

```
grep -rnw legacyParse repo --include='*.js' --exclude-dir=.git --exclude-dir=node_modules
```

```
repo/src/app.js:4:function start() { return legacyParse(input); }
repo/src/util.js:1:// TODO remove legacyParse once v2 ships
repo/src/util.js:2:function legacyParse(x) { return x; }
repo/src/util.js:3:module.exports = { legacyParse };
```

```
grep -rcw legacyParse repo --include='*.js' --exclude-dir=.git --exclude-dir=node_modules
```

```
repo/src/app.js:1
repo/src/util.js:3
```

One call in `app.js` and three mentions in `util.js` (the comment, the definition and the export), so only one real caller remains. The count per file gets you a to-do list for the clean-up.

WHAT GREP -R DOES NOT KNOW

`grep -r` knows nothing about `.gitignore`, or about which files are tracked by git. That is why we had to name `.git`, `node_modules` and `dist` ourselves. Chapter 8 shows tools that do know.

Job 3: Auditing Configuration Files

Configuration files are mostly comments and blank lines, and the setting you care about may appear in a comment, in a different file or not at all. First, the file as the program sees it: no comments, no blank lines.

```
grep -vE '^[[:space:]]*(#|$)' etc/sshd_config
```

```
Port 22
PermitRootLogin yes
PasswordAuthentication yes    # TODO tighten
X11Forwarding no
```

```
grep -vE '^[[:space:]]*(#|;|$)' etc/app.prod.ini
```

```
debug = false
log_level = info
secret_key = ${SECRET_KEY}
```

`^[[:space:]]*` allows indentation before the comment mark, and the second pattern adds `;` because that file uses semicolons for comments. Notice that the inline comment on the `PasswordAuthentication` line survived: only lines that *start* with a comment are removed.

Find a setting everywhere

Anchor at the start of the line (so a comment does not count) and ignore case (the keys are not case-sensitive):

```
grep -rniE '^[[:space:]]*(PermitRootLogin|PasswordAuthentication)' etc
```

```
etc/sshd_config:3:PermitRootLogin yes
etc/sshd_config:4:PasswordAuthentication yes    # TODO tighten
```

Both are set to `yes`, which is the opposite of what you want on a server.

Which files do not set it?

`-L` lists the files with no match (Chapter 2). The obvious try is misleading:

```
grep -rL -i debug etc
```

```
etc/sshd_config
```

It lists only `sshd_config`, but `app.dev.ini` does not set `debug` either: it merely contains the word, in `log_level = debug`. A search for the setting has to say it is the start of a line and followed by an equals sign:

```
grep -rLE '^[[:space:]]*debug[[:space:]]*=' etc --include='app*.ini'
```

```
etc/app.dev.ini
```

The same idea with `-c` gives a count per file, which is the quickest overview (1, 1 and 0):

```
grep -cE '^[[:space:]]*debug[[:space:]]*=' etc/app.prod.ini etc/app.staging.ini etc/app.
```

```
etc/app.prod.ini:1
etc/app.staging.ini:1
etc/app.dev.ini:0
```

And the files where it is set to something dangerous:

```
grep -rniE '^[[:space:]]*debug[[:space:]]*=[[:space:]]*true' etc
```

```
etc/app.staging.ini:2:debug = true
```

An audit you can run again

Put the rule in a function and the expectations in a list. For each setting there are three outcomes, and they are different problems: it is **right**, it is set to **the wrong value**, or it is **not**

set at all (the default applies):

```
check() {    # check FILE KEY EXPECTED
  if grep -qiE "^[:space:]*$2[:space:]=+$3[:space:]*(#.*)?$" "$1"; then
    echo "PASS      $1: $2 = $3"
  elif grep -qiE "^[:space:]*$2([[:space:]=]|$)" "$1"; then
    echo "FAIL      $1: $2 is set, but not to $3"
  else
    echo "MISSING   $1: $2 is not set"
  fi
}
```

```
check etc/sshd_config    PermitRootLogin      no
check etc/sshd_config    PasswordAuthentication no
check etc/sshd_config    X11Forwarding      no
check etc/sshd_config    PermitEmptyPasswords no
check etc/app.prod.ini    debug false
check etc/app.staging.ini debug false
check etc/app.dev.ini     debug false
```

```
FAIL      etc/sshd_config: PermitRootLogin is set, but not to no
FAIL      etc/sshd_config: PasswordAuthentication is set, but not to no
PASS      etc/sshd_config: X11Forwarding = no
MISSING   etc/sshd_config: PermitEmptyPasswords is not set
PASS      etc/app.prod.ini: debug = false
FAIL      etc/app.staging.ini: debug is set, but not to false
MISSING   etc/app.dev.ini: debug is not set
```

Read the result carefully. `PermitEmptyPasswords` is `MISSING` because the only line for it is commented out, and the audit says so rather than passing it. The debug setting passes in production, fails in staging, and is missing in development. A function like this is the start of a real configuration check, and because it uses `grep -q` (Chapter 1) it also works in a script: it can count the FAIL lines and exit with a status.

`$2` and `$3` are put into the pattern as they are, which is fine for simple names and values, and wrong for anything with regular-expression characters in it. (Chapter 6: text and patterns are different things.)

Habits That Carried Through All Three

- **Build the pipeline one stage at a time.** Run the grep, read its output, then add the next stage. A long pipeline that does not work is much harder to debug than a short one that does.
- **Narrow the data before narrowing the pattern.** `--exclude-dir`, `-I`, `--include` and anchoring at the start of the line removed more noise than any clever pattern did.
- **Count, then look.** `uniq -c` and `-c` tell you whether a problem is one line or two hundred before you read any of them.
- **Treat the result as candidates.** Searches find what they are told to find. The test-file password and the minified bundle were both “correct” hits and neither was a problem.
- **Keep secrets off the screen.** Print where, not what.
- **Save what worked.** The audit function is more useful as a file in your scripts folder than as something you will retype next month.

Hands-On Exercises

EXERCISE 1

From `service.log`, list the three slowest requests with their request id, method, path and time, slowest first. Then count the errors in the current and the rotated log together and show how many of the requests in the rotated log failed.

 [View solution](#)

EXERCISE 2

Produce a two-part report for `repo`: how many TODO, FIXME and HACK notes each file has (ignoring `.git`, `node_modules` and `dist`), and which files hold probable secrets, as file and line number only. Then explain why two of the secret candidates are probably not a problem.

 [View solution](#)

EXERCISE 3

Extend the audit: add two more checks of your choice on `etc/sshhd_config`, make the script print a one-line summary of the number of FAIL and MISSING results, and exit with status 1 if there are any.

CHAPTER 7 QUICK REFERENCE

- Logs: `grep -oE 'INFO|WARN|ERROR' | sort | uniq -c | sort -rn`; time windows with a pattern (`09:1[5-8]`) or `awk '$2 >= "09:15:00" && $2 <= "09:18:59"'`; follow an id with `grep 'req=ID'`; slow requests `grep -oP 'took=\K[0-9]+' | awk '$1 > 1000'`; rotated logs with `zgrep`
- Codebase: `grep -rnIE --exclude-dir=.git --exclude-dir=node_modules '\b(TODO|FIXME|HACK)\b' .`; exclude generated folders too; `cut -c1-N` tames long lines; `-w` and `--include` to find a function's uses
- Secrets: known formats (`AKIA[0-9A-Z]{16}`), assignments (`password *= *"..."`), key headers (with `-e`); print `file:line` only (`cut -d: -f1,2`, or `-1`); results are candidates, not proof
- Config: `grep -vE '^[[:space:]]*(#|$)'` shows the effective file (inline comments stay); anchor settings at line start; `-L` needs a real setting pattern, not just the word; audit as PASS / FAIL / MISSING
- Habits: one stage at a time; narrow the data first; count, then look; keep secrets off the screen; save what worked

COMING NEXT

grep: Searching Text 8 is the last chapter: tools that know about git and ignore files (`git grep`, `ripgrep`, `ag`), what each does differently, grep on Windows, and a recap of the whole course.

grep: Searching Text

Chapter 8 · Faster Alternatives

grep is installed almost everywhere and does one thing well. For searching a **project**, though, other tools have the sensible defaults built in: they know about git, skip the files you would have excluded by hand, and are noticeably faster. This chapter looks at `git grep` and ripgrep (`rg`), and then at what to use on Windows when grep is not around.

WHAT WAS AND WAS NOT TRIED

`git grep` ran on both git 2.49.0 (Git Bash on Windows) and git 2.43.0 (WSL), with the same lines. Two listings are in a different *order* (the two `sort` programs put capital letters in different places), and the “not a git repository” message is longer in 2.43; nothing else differed. **ripgrep 15.0.0** is not installed on this machine: I used the copy that ships inside VS Code, from a scratch folder, and installed nothing, so ripgrep ran on Windows only (its paths come out with backslashes). **The Silver Searcher (`ag`) is not on this machine and I did not run it**; the little I say about it is from its documentation. PowerShell is Windows PowerShell 5.1. I did not try macOS.

A Practice Repository

Paste this into an empty folder. It creates a small git repository, `proj`, with two commits, and then adds files that git does not track. The git commands are given your user name locally, so nothing outside the folder is changed.

```
mkdir -p proj/src proj/build proj/data
cd proj
git init -q
git config user.name Demo
git config user.email demo@example.com
printf 'build/\n*.log\n.env\n' > .gitignore
printf '// TODO(alice): handle timeout\n// FIXME validate input\nfunction start() { retu
printf '# TODO: use a connection pool\n# TODO old note about caching\ndef connect():\n
printf '# Demo\nTODO write the docs\n' > README.md
printf 'logo\0TODO inside a binary\n' > data/logo.bin
git add -A
git commit -q -m "first"
printf '# TODO: use a connection pool\ndef connect():\n    # HACK: retry forever\n    re
git commit -q -am "drop the old caching note"
```

```
printf '// TODO generated\n' > build/out.js
printf 'ERROR TODO in a log\n' > debug.log
printf 'TODO=unset\n' > .env
printf 'TODO untracked note\n' > scratch.txt
printf 'TODO in a hidden file\n' > .hidden-notes
```

What you now have, as git sees it:

FILE	STATE
<code>src/app.js</code> , <code>src/db.py</code> , <code>README.md</code> , <code>data/logo.bin</code>	tracked (committed)
<code>build/out.js</code> , <code>debug.log</code> , <code>.env</code>	ignored, by <code>.gitignore</code>
<code>scratch.txt</code> , <code>.hidden-notes</code>	not tracked and not ignored

```
git status --short
```

```
?? scratch.txt
```

(`git status` lists only the two untracked files: ignored files are not shown.) Every file above contains the word `TODO`. The earlier commit also had a line, `TODO old note about caching`, that the second commit removed.

What Plain grep Finds

With the options from Chapter 2 (`-r`, `-I`), `grep` finds every file that contains `TODO`:

```
grep -rI TODO . | sort
```

```
./.env
./.git/hooks/sendemail-validate.sample
./.hidden-notes
./build/out.js
./debug.log
./README.md
./scratch.txt
```

```
./src/app.js
./src/db.py
```

Nine files, and only three of them are files git tracks. The rest are the files `.gitignore` says you do not care about, the hidden file, the untracked scratch note, and some sample scripts inside the `.git` folder (git puts them there when a repository is created). Chapter 7 dealt with this by naming each folder to skip. These tools do it from what git already knows.

git grep

`git grep` is part of git, so it is there wherever git is. It searches the files git **tracks**:

```
git grep -l TODO
```

```
README.md
data/logo.bin
src/app.js
src/db.py
```

Four files: the three text files and the binary. No `.git` folder, no ignored files, no untracked ones. It takes many of grep's options (here I used `-n -l -e -I -q` and `-P`, which worked on both machines; the manual lists the rest):

```
git grep -n TODO
```

```
README.md:2:TODO write the docs
Binary file data/logo.bin matches
src/app.js:1:// TODO(alice): handle timeout
src/db.py:1:# TODO: use a connection pool
```

```
git grep -n -P 'TODO\\(\\w+\\)'
```

```
src/app.js:1:// TODO(alice): handle timeout
```

The binary file is reported with git's own message, the same on both versions. `-I` skips it, as in `grep`, and the exit status is 1 when nothing is found (as in `grep`):

```
git grep -n TODO -- data; git grep -n -I TODO -- data; echo "status $?"
```

```
Binary file data/logo.bin matches
status 1
```

Untracked files, and files that are ignored

A file you have just made but not yet added is not searched. `--untracked` adds the files that are neither tracked nor ignored, which is usually what you want while you work:

```
git grep -n --untracked TODO
```

```
.hidden-notes:1:TODO in a hidden file
README.md:2:TODO write the docs
Binary file data/logo.bin matches
scratch.txt:1:TODO untracked note
src/app.js:1:// TODO(alice): handle timeout
src/db.py:1:# TODO: use a connection pool
```

`--no-index` goes the other way: it searches the working folder the way `grep -r` would (ignored files included, `.git` still left alone):

```
git grep -l --no-index TODO | sort
```

```
.env
.hidden-notes
build/out.js
data/logo.bin
debug.log
README.md
scratch.txt
src/app.js
src/db.py
```

Old versions, and the working tree

Name a commit and `git grep` searches the files *as they were* then. The note that was removed in the second commit is still in the first:

```
git grep -n 'old note' HEAD~1
```

```
HEAD~1:src/db.py:2:# TODO old note about caching
```

```
git grep -n 'old note'; echo "status $?"
```

```
status 1
```

By default it reads the files in your working folder, including changes you have not committed or staged. `--cached` searches what has been staged. I added a line to `README.md` without staging it:

```
echo 'TODO uncommitted' >> README.md
echo 'working tree:'; git grep -n uncommitted
echo 'index (--cached):'; git grep -n --cached uncommitted; echo "status $?"
```

```
working tree:
README.md:3:TODO uncommitted
index (--cached):
status 1
```

Combining conditions, and choosing paths

`--and`, `--or` and `--not` combine patterns *on the same line*, which in plain grep takes two greps in a pipe:

```
git grep -n -e TODO --and -e alice
```

```
src/app.js:1:// TODO(alice): handle timeout
```

```
git grep -n -e TODO --and --not -e alice -- src
```

```
src/db.py:1:# TODO: use a connection pool
```

After `--` come **paths**, git's own file patterns. `'*.js'` picks files by name and `'!:...'` removes some:

```
git grep -n TODO -- '*.js'
```

```
src/app.js:1:// TODO(alice): handle timeout
```

```
git grep -n TODO -- '!:*.md' '!:*.bin'
```

```
src/app.js:1:// TODO(alice): handle timeout
src/db.py:1:# TODO: use a connection pool
```

Run from a subfolder, it searches that folder and prints paths relative to it:

```
cd src && git grep -n TODO
```

```
app.js:1:// TODO(alice): handle timeout
db.py:1:# TODO: use a connection pool
```

Outside a repository

`git grep` needs a repository, and says so with status 128 (not grep's 2). `--no-index` makes it work anywhere:

```
cd ..
git grep -n TODO; echo "status $?"
git grep --no-index -n TODO proj/src
```

```
fatal: not a git repository (or any of the parent directories): .git
status 128
proj/src/app.js:1:// TODO(alice): handle timeout
proj/src/db.py:1:# TODO: use a connection pool
```

ripgrep (rg)

ripgrep searches the current folder *recursively by default*, and by default it skips what you would usually skip: files matched by `.gitignore` (its documentation names other ignore files too), hidden files and folders (so `.git` too), and binary files. Here is the same search with no options at all:

```
rg TODO | sort
```

```
README.md:TODO write the docs
scratch.txt:TODO untracked note
src\app.js:// TODO(alice): handle timeout
src\db.py:# TODO: use a connection pool
```

The same files as `git grep --untracked`, except two: `rg` skipped `.hidden-notes` because it is hidden, and the binary file because it is binary. `scratch.txt` was found although git does not track it, because rg works from *ignore files*, not from git's list of tracked files. Three things about this output:

- It looks like grep's, `file:text`. That is the form when the output goes to a pipe or a file. On a terminal rg instead groups results under a file heading, with line numbers and colour. I could only capture piped output here, so I have not shown the terminal form.
- Line numbers are off in that form: add `-n`.
- The files come out in **any order**, because rg searches several at once. I sorted the output above with `| sort` (Chapter 4's habit again).

```
rg -n TODO | sort
```

```
README.md:2:TODO write the docs
scratch.txt:1:TODO untracked note
src\app.js:1:// TODO(alice): handle timeout
src\db.py:1:# TODO: use a connection pool
```

On Windows, rg printed paths with a backslash, as in `src\app.js` (I expect slashes on Linux and macOS but could not check).

The -u switches: how much to skip

`--files` lists the files rg would search, without searching: the best way to see what its defaults leave out.

```
rg --files | sort
```

```
data\logo.bin
README.md
scratch.txt
src\app.js
src\db.py
```

Each `-u` ("unrestricted") switches off one more kind of skipping: `-u` stops honouring ignore files, `-uu` also searches hidden files and folders, `-uuu` also searches binary files. The number of files grows at every step (mostly `.git` internals at `-uu`):

```
for level in "" -u -uu -uuu; do printf "rg %-5s --files: " "$level"; rg $level --files |
```

```
rg      --files: 5
rg -u   --files: 7
rg -uu  --files: 46
rg -uuu --files: 46
```

With `-uuu` a binary match is reported, with a note of where the zero byte was:

```
rg -uuu -n TODO | sort | cut -c1-90
```

```
.env:1:TODO=unset
.git\hooks\sendemail-validate.sample:22:# Replace the TODO placeholders with appropriate
.git\hooks\sendemail-validate.sample:27:      # TODO: Replace with appropriate checks
.git\hooks\sendemail-validate.sample:35:      # TODO: Replace with appropriate checks
.git\hooks\sendemail-validate.sample:41:      # TODO: Replace with appropriate checks
.hidden-notes:1:TODO in a hidden file
build\out.js:1:// TODO generated
data\logo.bin: binary file matches (found "\0" byte around offset 4)
debug.log:1:ERROR TODO in a log
README.md:2:TODO write the docs
scratch.txt:1:TODO untracked note
```

```
src\app.js:1:// TODO(alice): handle timeout
src\db.py:1:# TODO: use a connection pool
```

A claim worth testing: `rg -uu` (nothing ignored, hidden files searched, binary files still skipped) should behave like Chapter 2's `grep -rnI --exclude-dir=.git`. Run both, normalise the paths and sort, and compare:

```
diff <(grep -rnI --exclude-dir=.git TODO . | sed 's|^\.\/|/' | sort) <(rg -uu -n -g '!.*' TODO .)
echo "grep lines: $(grep -rnI --exclude-dir=.git TODO . | wc -l); rg lines: $(rg -uu -n
```

```
identical output
../_s.sh: line 5: unexpected EOF while looking for matching `"'
```

Choosing files

`-t` chooses a file *type* (rg knows a long list: `rg --type-list`), `-g` a glob, and a leading `!` in a glob excludes:

```
rg -t js -n TODO
```

```
src\app.js:1:// TODO(alice): handle timeout
```

```
rg -g '*.py' -n TODO
```

```
src\db.py:1:# TODO: use a connection pool
```

```
rg -g '!README.md' -l TODO | sort
```

```
scratch.txt
src\app.js
src\db.py
```

The same options, and a few new ones

`-i -w -x -F -v -c -l -o -A -B -C -e -f` and `-n` mean what they mean in grep. A new one is **smart case**: `-S` ignores case if the pattern is all lower case, and is case-sensitive as soon as it has a capital. Plain `rg` is case-sensitive, like `grep`, and exits with 1 for no match:

```
rg todo -l; echo "status $?"  
rg -S todo -l | sort
```

```
status 1  
README.md  
scratch.txt  
src\app.js  
src\db.py
```

`-c` counts matching lines per file, as `grep` does:

```
rg -c TODO | sort
```

```
README.md:1  
scratch.txt:1  
src\app.js:1  
src\db.py:1
```

`-z` searches inside compressed files, as `zgrep` (Chapter 7) did:

```
printf 'a ERROR b\n' | gzip > x.gz  
rg -z ERROR x.gz
```

```
a ERROR b
```

A different regular expression language

`rg`'s patterns are written in the Rust regex syntax: close to Chapter 3's extended form, with `\d` `\w` `\s` and lazy repeats, but with **no look-around and no back-references**. The error message points to the way round:

```
rg '(?<=TODO)\(' -n; echo status $?
```

```
rg: regex parse error:
```

```
(?: (?<=TODO)\(  
      ^^^^
```

```
error: look-around, including look-ahead and look-behind, is not supported
```

Consider enabling PCRE2 with the `--pcre2` flag, which can handle backreferences and look-around.

```
status 2
```

`-P` switches to Perl-style patterns (Chapter 5), if `rg` was built with them; mine was:

```
rg -P '(?<=TODO)\(' -n
```

```
src\app.js:1:// TODO(alice): handle timeout
```

There is one more difference worth knowing. `rg`'s `\w` understands **Unicode**, so it keeps accented letters in a word, where `grep -P` (Chapter 5) breaks them up. A file containing `café résumé`:

```
printf 'caf\xc3\xa9 r\xc3\xa9sum\xc3\xa9\n' > u.txt  
echo "rg:";      rg -o '\w+' u.txt  
echo "grep -P:"; grep -oP '\w+' u.txt
```

```
rg:  
café  
résumé  
grep -P:  
caf  
r  
sum
```

Where `rg` gets it wrong for you

`rg` honours `.gitignore` only inside a git repository, which is easy to forget when you copy files somewhere else. Outside one, the folder `build` is searched. `--no-require-git` makes `rg` obey the ignore file anyway, and so does running `git init`:

```
mkdir ../plain && cd ../plain  
printf 'build/\n' > .gitignore; mkdir build
```

```
echo 'TODO a' > build/x.js; echo 'TODO b' > y.js
echo "not a git repository:";      rg -n TODO | sort
echo "with --no-require-git:";     rg --no-require-git -n TODO
git init -q
echo "after git init:";            rg -n TODO
```

```
not a git repository:
build\x.js:1:TODO a
y.js:1:TODO b
with --no-require-git:
y.js:1:TODO b
after git init:
y.js:1:TODO b
```

And when a search finds *less* than you expect, the first suspect is one of the skipping rules, which `-u`, `-uu` and `-uuu` will confirm. A missing file gives status 2 and a different message from `grep`'s:

```
rg TODO nosuchfile; echo "status $?"
```

```
rg: nosuchfile: IO error for operation on nosuchfile: The system cannot find the file specified
status 2
```

The Silver Searcher (ag)

`ag` is a similar tool: from its documentation, it also searches recursively by default and respects ignore files. It is not installed here and I did not run it, so I will not describe its options. If you already use it, the ideas in this chapter apply: check what it skips by default.

How Much Faster?

A measurement, not a promise. The folder is this course's own `content` folder: 13,136 files, 12,470 of them tracked by git. Each command lists the files containing a word. First a word that occurs in 1,976 files, then one that occurs in none. Each ran three times; the numbers are **seconds**, as `time` measured them. The `grep` is the Windows build in Git Bash (3.0), the files are

in a OneDrive folder, and I did not repeat this on Linux, so treat the figures as one data point from this machine:

```
(a script that ran grep -rII, git grep -l and rg -l on the folder three times each; its output)

files in the folder: 13136; tracked by git: 12470
=== pattern: copyCodeBlock
matching files:  grep -rII: 1976    git grep -l: 1976    rg -l: 1976
grep -rII      seconds, three runs: 36.594 5.885 3.586
git grep -l    seconds, three runs: 0.608 0.467 0.609
rg -l          seconds, three runs: 0.553 0.460 0.512
=== pattern: zzzqqqnotthere
matching files:  grep -rII: 0    git grep -l: 0    rg -l: 0
grep -rII      seconds, three runs: 3.820 3.593 3.694
git grep -l    seconds, three runs: 0.514 0.526 0.640
rg -l          seconds, three runs: 0.511 0.539 0.588
```

Plain grep took about 3.6 seconds for each run once the files were cached, and 36.6 seconds on the very first run, when they had to be read from disk. `git grep` and `rg` took about half a second each, roughly seven times quicker than the warm grep, and all three found the same 1,976 files. The first run of a tool you have not used for a while is the slow one, which is why I ran each three times. On a project of a few dozen files, none of this matters.

grep on Windows

You have four ways to get grep-like searching on a Windows machine, and they are not the same:

OPTION	WHAT IT IS	NOTES FROM THIS COURSE
Git Bash	a Windows build of GNU grep, with git	what most examples ran on (grep 3.0); it hides carriage returns, so <code>\$</code> behaves differently from Linux (Chapter 3); it is the grep that was timed in the benchmark above (I did not time WSL's)
WSL	a real Linux, with grep 3.11	behaves like a Linux server; the right place to test a script that will run on one
PowerShell	a PowerShell command, not grep	below

OPTION	WHAT IT IS	NOTES FROM THIS COURSE
findstr	a very old Windows command-line tool	below
Select-String <code>Select-String</code> does the job in PowerShell. It differs from <code>grep</code> in ways that catch people out. First, it is case-insensitive by default: here <code>todo</code> matches <code>TODO</code>. Its output is a <i>match object</i>, which prints in a familiar <code>file:line:text</code> form (PowerShell adds a blank line before the first result, left out here): <pre>Select-String -Path src\app.js -Pattern todo</pre> <pre>src\app.js:1:// TODO(alice): handle timeout</pre> <code>-CaseSensitive</code> gives <code>grep</code>'s behaviour: <pre>Select-String -Path src\app.js -Pattern todo -CaseSensitive '(nothing above: -CaseSensitive found no lowercase todo)'</pre> <pre>(nothing above: -CaseSensitive found no lowercase todo)</pre> It does not recurse by itself. Feed it files from <code>Get-ChildItem -Recurse</code>. It knows nothing about <code>.gitignore</code>, so <code>build\out.js</code> is found: <pre>Get-ChildItem -Recurse -File -Include *.js,*.py,*.md Select-String TODO</pre> <pre>build\out.js:1:// TODO generated src\app.js:1:// TODO(alice): handle timeout src\db.py:1:# TODO: use a connection pool README.md:2:TODO write the docs</pre>		
GREP	SELECT-STRING	
<code>-i</code> (ignore case)	the default; use <code>-CaseSensitive</code> to turn it off	
<code>-v</code>	<code>-NotMatch</code>	

GREP

SELECT-STRING

<code>-F</code>	<code>-SimpleMatch</code>
<code>-A N -B M / -C N</code>	<code>-Context M,N</code> (before, after)
<code>-o</code>	<code>-AllMatches</code> and read <code>.Matches.Value</code>
<code>-l</code>	<code>-List</code> (one match per file; then read the file name from the result)
<code>-q</code>	<code>-Quiet</code> (returns True or False)
<code>-r</code>	<code>Get-ChildItem -Recurse -File Select-String ...</code>

```
Select-String -Path src\db.py -Pattern TODO -NotMatch
```

```
src\db.py:2:def connect():
src\db.py:3:    # HACK: retry forever
src\db.py:4:    return None
```

```
Select-String -Path src\app.js -Pattern FIXME -Context 1,1
```

```
src\app.js:1:// TODO(alice): handle timeout
> src\app.js:2:// FIXME validate input
src\app.js:3:function start() { return legacyParse(input); }
```

```
Select-String -Path src\app.js -Pattern 'TODO\(\w+\)' -AllMatches | ForEach-Object { $_.Line
(Select-String -Path src\*.js -Pattern TODO -CaseSensitive).Count
```

```
TODO(alice)
2
```

The first shows `-NotMatch` (lines without `TODO`, with their line numbers); the second `-Context 1,1`, with `>` marking the match; the third pulls out just the matching text and counts the matching lines (case-sensitively).

THERE IS NO EXIT STATUS FOR "NO MATCH"

grep says 1 when nothing matched. `Select-String` just returns nothing, and PowerShell's own status says all is well. To test for a match in a script use `-Quiet`, which returns `True` or `False`:

```
Select-String -Path src\app.js -Pattern NOPE -Quiet
$r = Select-String -Path src\app.js -Pattern NOPE
"no output object: $($null -eq $r); $? = $?"
```

```
False
no output object: True; True = True
```

findstr

`findstr` is old, but it is present on every Windows machine and works from `cmd`. `/n` numbers lines, `/i` ignores case, `/s` recurses, `/m` lists file names only, and `/c:"..."` searches for a phrase with spaces. It sets an exit status like grep (0, or 1 for no match):

```
findstr /n /i /c:"use a connection" src\db.py
echo status %errorlevel%
findstr /s /m /i todo *.js *.py
findstr /n NOPE src\db.py
echo status when nothing found %errorlevel%
```

```
1:# TODO: use a connection pool
status 0
build\out.js
src\app.js
src\db.py
status when nothing found 1
```

Its regular-expression support is limited and I did not test it, so use grep, `rg` or `Select-String` when the pattern is more than words.

Which One Should I Use?

SITUATION	USE
a script that has to run on any Linux, macOS or server	<code>grep</code> (<code>-E</code> , not <code>-P</code>)

SITUATION	USE
searching a git project, every day, interactively	<code>git grep</code> (no install) or <code>rg</code>
searching old versions of a project	<code>git grep PATTERN REVISION</code>
a big folder, speed matters, and you can install a tool	<code>rg</code>
files that git does not know about, or compressed logs	<code>grep</code> , <code>zgrep</code> or <code>rg -z</code>
a Windows machine with nothing but PowerShell	<code>Select-String</code> , and install Git Bash or WSL when you can
lookahead or back-references in a script for yourself	<code>grep -P</code> or <code>rg -P</code>

A caution that applies to every tool in this chapter: their defaults **hide files**. A search that finds nothing might mean nothing is there, or that the tool skipped it. When it matters, check with the “do not skip anything” form (`grep -r`, `rg -uuu`, `git grep --no-index`).

Hands-On Exercises

EXERCISE 1

With `git grep`: list the tracked files that mention `TODO`, show which note existed in the first commit and is gone now, and find the line that has both `TODO` and `alice`. Then say which three kinds of file `grep -r` would have added.

 [View solution](#)

EXERCISE 2

Show that `rg -uu` and Chapter 2's `grep -rnI --exclude-dir=.git` find the same lines, then find where they differ if you drop `-uu` to `-u`, and say what each of the three `-u` levels turns off.

 [View solution](#)

EXERCISE 3

In PowerShell, write the `Select-String` versions of three greps from the repository (case-sensitive TODO in Python files; lines without TODO in `db.py`; the files that contain FIXME) and check each against grep's answer.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- `git grep`: tracked files, never `.git`; `--untracked` adds new files; `--no-index` works anywhere and includes ignored files; `git grep PAT REV` searches an old version; `--cached` the index; `--and --or --not`; `-- '*.js' '!*.*md'` pathspecs; status 128 outside a repo
- `rg`: recursive by default; skips ignored, hidden and binary files; `-u` / `-uu` / `-uuu` turn off ignore, hidden, binary skipping; `--files` shows what it would search; `-t`, `-g`; `-S` smart case; `-z` compressed; `-P` for look-around (the default has none)
- `rg`: output looks like grep's when piped (add `-n`); order is not stable (`| sort`); only obeys `.gitignore` in a git repo; Unicode-aware `\w`
- Measured here: about 3.6 s for Git Bash grep against about 0.5 s for `git grep` and `rg` on 13,136 files (one machine, one afternoon)
- Windows: Git Bash (grep 3.0, hides CR), WSL (Linux grep), `Select-String` (case-insensitive by default, no recursion, no ignore files, `-Quiet` for a yes or no), `findstr` (`/n /i /s /m /c:`, status 0 or 1)
- Every tool's defaults hide files: when a search finds nothing, check with the version that skips nothing

Course Complete

That is all eight chapters. You can now search text with grep from a single file up to a whole project, write patterns in basic, extended and Perl form, use grep safely in scripts, and pick a better tool when grep is not the best one. What each chapter left you able to do:

CHAPTER	YOU CAN NOW...
1 · First Searches	search for text; ignore case, invert, match whole words and whole lines; fixed strings; quote a pattern; read the exit status
2 · Searching Folders	search a whole tree with <code>-r</code> ; choose files and folders; deal with binary files, links and file names with spaces

CHAPTER	YOU CAN NOW...
3 · Basic and Extended Regex	anchors, sets and classes, groups, counts and back-references; know which punctuation is an operator in each form
4 · Context and Output Control	show lines around a match, line numbers and file names, count lines or matches, stay quiet, and count things with <code>sort uniq -c</code>
5 · Perl Regex with -P	use lazy repeats, lookahead and <code>\K</code> ; match across lines with <code>-z</code> ; know where <code>-P</code> stops
6 · grep in Pipelines and Scripts	use grep as a test in <code>if</code> ; avoid the <code>set -e</code> , <code>pipefail</code> and blank-pattern traps; edit files safely after finding them
7 · Real-World Searching	read a log, hunt through a codebase without printing secrets, and audit configuration files
8 · Faster Alternatives	choose between grep, <code>git grep</code> , ripgrep and the Windows tools, with measured speed

WHERE NEXT

The other tools in this family are on the site: *SED — A Course for Beginners* to change text, *AWK* to work with fields, and *Learning Regular Expressions* for the pattern language underneath all of them. The *grep Cheat Sheet* in the Linux part of the Sidebar is the one-page reminder of everything in Chapters 1 to 5.