

COMPANION VOLUME • STAGE 00 OF THE ENGINEER'S SYLLABUS

The Engineer's Whetstone

A whetstone cuts nothing. It keeps every other blade sharp. This volume is about the tools you'll use every working day: the shell and the small programs it runs, the filesystem underneath them, Git, one editor, and the skill of learning itself. Each turns out to rest on a handful of ideas, and once you see them, commands you used to copy and paste become things you can reason about. Every chapter runs the real tools and shows exactly what they did.

CHAPTERS

16

CODE EXAMPLES

16

EXERCISES

48

STAGE BUDGET

60 h

Contents

How to use this volume	3
Part I · The shell	5
Chapter 1 · The shell as a language	6
Chapter 2 · Streams, pipes and exit codes	13
Chapter 3 · Searching and transforming text	20
Chapter 4 · Scripts that don't break	28
Part II · The machine	36
Chapter 5 · Files, paths and links	37
Chapter 6 · Permissions	44
Chapter 7 · Processes, the environment and PATH	51
Chapter 8 · Dotfiles	58
Part III · Git	65
Chapter 9 · Git's objects	66
Chapter 10 · Refs, branches and HEAD	73
Chapter 11 · Merging, rebasing and conflicts	80
Chapter 12 · History as a debugging tool	88
Chapter 13 · Remotes, forks and pull requests	95
Part IV · Learning the craft	102
Chapter 14 · One editor, to fluency	103
Chapter 15 · Error messages, docs and good questions	111
Chapter 16 · How to learn	118
The deliverable	125
The sixteen rules	127
Reading list	129

Almost nobody teaches this stage, and skipping it costs beginners hundreds of hours. The cost is spread thin: twenty minutes lost to a quoting mistake, an afternoon to a Git state nobody understands, a week of half-remembered tutorials. None of it feels like the thing holding you back, which is why it goes on for years. Sixty hours here pays for itself many times over, in every stage after it. The table shows where each of the stage's topics is taught, and what to read or practise alongside.

STAGE 00 TOPIC	CHAPTERS	READ OR PRACTISE ALONGSIDE
The shell as a language: pipes, redirection, globbing, exit codes, <code>grep</code>, <code>find</code>, <code>sed</code>, <code>awk</code>	Ch. 1–4	The Missing Semester, lectures 1, 2 and 4
The filesystem model: paths, permissions, symlinks, environment variables, <code>PATH</code>	Ch. 5–8	The Linux Command Line, parts 1 and 2
Git as a data structure: commits, trees, refs, branching, rebase, merge conflicts, <code>bisect</code>	Ch. 9–12	Pro Git, chapters 1–3, 7 and 10; Learn Git Branching
Collaborating on GitHub: forks, pull requests, review comments, issues	Ch. 13	GitHub Docs on pull requests; Pro Git chapter 6
One editor to fluency, with its keybindings, search and multi-file navigation	Ch. 14	The Missing Semester, lecture 3
Reading documentation and error messages instead of pasting them somewhere	Ch. 15	Your system's man pages; the Python tutorial on errors
Asking a question well: minimal reproduction, what you tried, what you expected	Ch. 15	Stack Overflow's guide to a minimal, reproducible example
Study technique: spaced repetition, deliberate practice, retrieval over rereading	Ch. 16	Make It Stick

Four parts, one a week. At fifteen hours a week the stage takes a month. Spend the first week on the shell, the second on the machine underneath it, finishing with your dotfiles repository. Give the third, the longest, to Git, and the fourth to your editor, error messages and study habits, while you finish the deliverable. Chapter 16 is short and can be read at any point. If you're starting the whole syllabus here, read it first: it changes how you'll study everything else.

Predict, run, explain. Each chapter's example is a Python program that runs the real tools, `bash`, `grep`, `sed`, `awk`, `find` and `Git`, in a temporary folder, and shows exactly what they printed. Before you read an output, cover it and predict what it will be. Then run the example yourself and, when your prediction was wrong, find out why before going on. The examples give the same output on any Mac or Linux machine. They give every commit a fixed author and time, ignore your own `Git` settings, and sort anything whose order would otherwise depend on the filesystem. On Windows, use the Windows Subsystem for Linux: the stage is about the Unix tools that every server runs.

Type the commands as well. The examples show what happens; your fingers have to learn it. Keep a terminal open while you read, and try each command in a scratch folder, made with `mkdir -p ~/scratch && cd ~/scratch`, where nothing can be lost. MIT's *The Missing Semester of Your CS Education* covers the same ground as lectures with exercises, and is the best two weeks of free teaching in the syllabus. Do its exercises too.

The deliverable is two things you'll keep: a public repository of your dotfiles, the settings that make your tools yours, with a script that installs them on a new machine, and one shell script that automates a chore you genuinely repeat. Chapters 8 and 4 build a small working version of each, and the appendix lists what the full versions need. Write each README as if a stranger will use it, because one will: you, on a new laptop, in a year.

THE ONE HABIT

The tools in this volume hide almost nothing; they just don't volunteer it. The shell will tell you exactly what a line expands to, `Git` will draw its whole graph, and every error message names what went wrong.

So look before you change anything. Run `type` before blaming `PATH`, `git status` and `git log --graph --all` before any `Git` command you're unsure of, and read the whole error message before searching for it. Most confusion ends the moment you look at the actual state instead of the one you imagined.

PART I

The shell

Four chapters on the program you'll type into for the rest of your career. The shell is a small programming language whose real job is starting other programs and connecting them together. First, how it turns a line you type into a list of arguments, which explains almost every quoting bug. Then the streams and exit codes that let programs cooperate, the four tools that answer most questions about text, and the habits that make a shell script safe to run while you sleep. By the end of this part you can automate a chore and trust the result.

The shell as a language

3 exercises

The shell is a programming language whose main job is starting other programs. Before it runs a line, it rewrites it: it replaces variables, expands patterns into file names, splits the result into words, and only then hands the program a list of arguments. The program never sees your quotes, your `$` signs or your `*`. Almost every surprise in the shell is a surprise about that rewriting, so this chapter shows exactly which list a program receives for each kind of line.

A command is a list of words

Three things are easy to confuse. The **terminal** is the window: it draws characters and sends your key presses on. The **shell** is the program running inside it, usually `bash` or `zsh`, which reads each line you type and decides what to do with it. And the **commands** are ordinary programs, `ls`, `grep`, `git`, which the shell starts. When you type `ls -l Documents`, the shell finds a program called `ls`, which chapter 7 explains, and starts it with a list of three strings: `ls`, `-l` and `Documents`. That list, called `argv` in C and `sys.argv` in Python, is everything the program gets from the line. What `-l` means is entirely up to `ls`.

Programs mostly follow the same conventions for their arguments. Single letters after one dash are options, and can often be combined: `ls -l -a` is `ls -la`. Long names take two dashes, as in `ls --all`. Some options take a value, like `head -n 5`, and a lone `--` means "no more options", so `rm -- -notes.txt` deletes a file whose name begins with a dash. These are conventions, not rules the shell enforces. The shell's own job ends at producing the list.

Quoting

The shell gives special meaning to a few characters: spaces separate words, `$` starts a variable or substitution, `*`, `?` and `[` make patterns, and `|`, `>`, `<`, `&` and `;` connect or separate commands. Quoting turns those meanings off. Inside **single quotes**, everything is literal: `'$HOME *'` is exactly those seven characters. Inside **double quotes**, spaces and pattern characters are literal, but `$`, backquotes and backslash still work, so `"$HOME"` becomes your home folder as one word, however many spaces it contains. A **backslash** outside quotes makes the next character literal: `it\'s`. Quotes can be mixed within one word: `'single'"double"` is one argument, `singledouble`. The quotes themselves are removed before the program starts, which is why it never sees them.

The expansions, in order

Before running a line, bash performs a fixed sequence of **expansions**, and the order explains the surprises. **Brace expansion** comes first, turning `file{1..3}.log` into three words. **Tilde expansion** turns `~` at the start of a word into your home folder. Then, from left to right, **parameter expansion** replaces `$name` and `${name}` with variables' values, **arithmetic expansion** computes `$(2 + 3 * 4)`, and **command substitution** runs `$(command)` and puts its output in its place. Only then does **word splitting** break the results of those three, wherever they weren't in double quotes, at spaces, tabs and newlines. **Pathname expansion**, usually called globbing, then replaces each word containing `*`, `?` or `[` with the matching file names. Finally, **quote removal** deletes the quotes.



One line, stage by stage. The unquoted `$f` is split after it's expanded, but file names found by a glob never are: the space in `b c.txt` survives.

The best way to learn this is to watch it. The example creates a tiny program, `showargs`, which prints its arguments one per bracket, so you can see exactly where each word begins and ends. It then runs it through bash in a folder holding `a.txt`, `b c.txt` and `notes.md`, with a variable `f` holding `my notes.txt` and another, `empty`, holding nothing.

EXAMPLE

`shell.py` · a program that prints its arguments, run through bash with quotes, variables, globs, braces, arithmetic and command substitution, then run with no shell at all

```
import os, pathlib, re, shutil, subprocess, tempfile

# A tiny program that prints the arguments it receives, each in brackets so spaces are visible.
SHOWARGS = '#!/bin/sh\nprintf "%d argument(s):" "$#\nfor a in "$@"; do printf " [%s]" "$a";\ndone\nnecho\n'

def bash(line, cwd, env):
    """Run one line with bash, after a fixed prelude, and return what it printed."""
    prelude = 'f="my notes.txt"; empty=""\n'
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", prelude + line],
                       cwd=cwd, env=env, capture_output=True, text=True)
    if r.returncode:
        why = re.search(r"([^\n]+: command not found)", r.stderr)
```

```

        return f"exit status {r.returncode}" + (f': "{why.group(1).strip()}"' if why else "")
    return r.stdout.rstrip("\n")

```

```

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp)
    (root / "bin").mkdir(); (root / "demo").mkdir()
    tool = root / "bin" / "showargs"; tool.write_text(SHOWARGS); tool.chmod(0o755)
    for name in ("a.txt", "b c.txt", "notes.md"): (root / "demo" / name).write_text("")
    env = {"PATH": f"{root / 'bin'}:{os.environ['PATH']}", "HOME": "/home/learner", "LC_ALL":
        "C"}

    groups = {
        "1. The shell splits a line into words at unquoted spaces; quotes keep spaces in": [
            "showargs hello    world", 'showargs "hello    world"',
            "showargs it\\'s \"say \\\"hi\\\"\" 'a\\nb'"],
        "2. $name is replaced, then split again unless it's in double quotes": [
            "showargs $f", 'showargs "$f"', "showargs $empty", 'showargs "$empty"',
            "showargs 'single $HOME' \"double $HOME\""],
        "3. Globs are matched against file names; a glob that matches nothing is left alone": [
            "showargs *.txt", "showargs *.csv", 'showargs "*.txt"'],
        "4. Other expansions: tilde, braces, arithmetic, command substitution": [
            'showargs ~/notes "~/notes"', "showargs file{1..3}.log {red,green}-pen",
            "showargs $((2 + 3 * 4))",
            "showargs $(printf 'one    two')", 'showargs "$(printf \'one    two\')"'],
        "5. The order matters: a substitution's result is split and then globbed": [
            "showargs $(printf '*.txt')"],
        "6. Spaces around = make it a command, not an assignment": [
            "x = 5", 'x=5 y="a b"; showargs $x "$y"'],
    }
    for title, lines in groups.items():
        print(title)
        for line in lines:
            print(f"    ${line}\n    {bash(line, root / 'demo', env)}")

    print("7. Without a shell there is no expansion: the program gets the list exactly as
        given")
    r = subprocess.run([str(tool), "*.txt", "my notes.txt", "$HOME"], cwd=root / "demo",
        env=env, capture_output=True, text=True)
    print(f"    subprocess.run(['showargs', '*.txt', 'my notes.txt', '$HOME'])\n
        {r.stdout.strip()}")

```

OUTPUT

```

1. The shell splits a line into words at unquoted spaces; quotes keep spaces in
$ showargs hello    world
  2 argument(s): [hello] [world]
$ showargs "hello    world"
  1 argument(s): [hello    world]
$ showargs it\\'s \"say \\\"hi\\\"\" 'a\\nb'
  3 argument(s): [it's] [say "hi"] [a\\nb]
2. $name is replaced, then split again unless it's in double quotes
$ showargs $f
  2 argument(s): [my] [notes.txt]
$ showargs "$f"
  1 argument(s): [my notes.txt]
$ showargs $empty
  0 argument(s):
$ showargs "$empty"
  1 argument(s): []
$ showargs 'single $HOME' \"double $HOME\"
  2 argument(s): [single $HOME] [double /home/learner]
3. Globs are matched against file names; a glob that matches nothing is left alone
$ showargs *.txt
  2 argument(s): [a.txt] [b c.txt]
$ showargs *.csv
  1 argument(s): [*.csv]

```

```

$ showargs "*.txt"
1 argument(s): [*.txt]
4. Other expansions: tilde, braces, arithmetic, command substitution
$ showargs ~/notes "~/notes"
2 argument(s): [/home/learner/notes] [~/notes]
$ showargs file{1..3}.log {red,green}-pen
5 argument(s): [file1.log] [file2.log] [file3.log] [red-pen] [green-pen]
$ showargs $((2 + 3 * 4))
1 argument(s): [14]
$ showargs $(printf 'one two')
2 argument(s): [one] [two]
$ showargs "$(printf 'one two')"
1 argument(s): [one two]
5. The order matters: a substitution's result is split and then globbed
$ showargs $(printf '*.txt')
2 argument(s): [a.txt] [b c.txt]
6. Spaces around = make it a command, not an assignment
$ x = 5
exit status 127: "x: command not found"
$ x=5 y="a b"; showargs $x "$y"
2 argument(s): [5] [a b]
7. Without a shell there is no expansion: the program gets the list exactly as given
subprocess.run(['showargs', '*.txt', 'my notes.txt', '$HOME'])
3 argument(s): [*.txt] [my notes.txt] [$HOME]

```

What the example shows

Unquoted expansions are split, and quoted ones aren't. `showargs $f` receives two arguments, `my` and `notes.txt`, because the value was split at its space. A program given those would look for two files that don't exist. `"$f"` keeps it whole. An empty variable unquoted disappears altogether, leaving zero arguments, while `"$empty"` is one empty argument. That difference is why `[ $answer = yes ]` fails with a strange error when `answer` is empty: the test receives `= yes`. The rule that prevents most shell bugs is simple: **put double quotes around every expansion**, `"$var"`, `"$(command)"`, `"$@"`, unless you can say why you want it split.

Globs match file names, and are expanded after splitting. `*.txt` became `a.txt` and `b c.txt`, sorted, with the space intact: names produced by globbing are never split again. A glob that matches nothing is passed on unchanged, so `showargs *.csv` received the literal `*.csv`; that's bash's default, and zsh's default is different: it stops with "no matches found". Quoted, `"*.txt"` is just text. Section 5 shows the order's sharpest edge: the output of `$(printf '*.txt')` is the text `*.txt`, and because substitution happens before globbing, it is then expanded into file names. Command output containing a `*`, a space or a newline is rewritten unless it's quoted.

Assignments have no spaces. `x=5` sets a variable. `x = 5` runs a program called `x` with arguments `=` and `5`, and fails with status 127, "command not found". Putting an assignment before a command, as in `x=5 y="a b" showargs`, sets them for that one command only, which chapter 7 uses. And the last line shows the other side: when Python's `subprocess.run` is given a list, there's no shell, so nothing is expanded. `*.txt` and `$HOME` arrive exactly as written. When you start programs from your own code, pass a list rather than a string to a shell, and the quoting problems disappear.

Globs are not regular expressions

Glob patterns look like the regular expressions of chapter 3, and mean different things. In a glob, `*` matches any run of characters, `?` exactly one, and `[abc]` one of a set; the pattern must match the whole name. A `*` doesn't match a leading dot, so `*` skips hidden files like `.bashrc`, and it doesn't cross `/`, so `src/*.py` looks in one folder only. To match in every subfolder, `zsh` understands `**/*.py` by default, and `bash` does after `shopt -s globstar`, which needs `bash 4`; for anything more, use `find`, in chapter 3. In a regular expression, by contrast, `*` means "the previous thing, any number of times", and `.` means any character.

Seeing what the shell will run

You never have to guess what a line expands to. `set -x` makes `bash` print each command, after expansion, before running it, with a `+` in front; `set +x` turns it off. It's the most useful debugging tool the shell has, and it works in scripts too. For a single line, put `echo` or `printf "[%s]\n"` in front of a command to see its arguments without running it. Two keyboard habits save hours as well: **Tab** completes commands and file names, and quotes awkward names for you, and **Ctrl-R** searches back through the commands you've run before. History is kept in a file, usually `~/.bash_history` or `~/.zsh_history`.

COMMON TRAPS

Unquoted variables, which break on the first file name with a space and vanish when empty.

.....
`for f in $(ls)`, which splits names at spaces; loop over a glob, `for f in *`, instead.
.....

Spaces around `=` in an assignment, which turn it into a command.

.....
Expecting `*` to match hidden files, or an unmatched glob to vanish: `bash` passes it on unchanged.
.....

Curly quotes pasted from a web page or word processor: `"` and `"` are ordinary characters to the shell, not quotes.

EXERCISES

EXERCISE 1.1 Predict the list

After `a="1 2"; b=""`, in a folder with no files whose names start with `1`, predict what `showargs` receives from each line: `showargs $a$b`, `showargs "$a"$b"`, `showargs $b $a`, `showargs '$a' "$a"`, `showargs {x,y}{1,2}` and `showargs "$a"*`.

SOLUTION

```

showargs $a$b          2 argument(s): [1] [2]
showargs "$a"$b"       1 argument(s): [1 2]
showargs $b $a         2 argument(s): [1] [2]
showargs '$a' "\$a"    2 argument(s): [$a] [$a]
showargs {x,y}{1,2}    4 argument(s): [x1] [x2] [y1] [y2]
showargs "$a"*         1 argument(s): [1 2*]

```

`$a$b` is `1 2` followed by nothing, split into two words. Quoted, `"$a"$b"` is one word, and the empty `"$b"` adds nothing to it because it's part of the same word. In `$b $a` the unquoted empty `$b` disappears. Single quotes and a backslash both stop expansion. Two brace expressions in one word multiply, giving every combination. In the last line the quoted part isn't split, so the glob is the single pattern `1 2*`, which matches nothing and is passed on unchanged. Each answer was produced by running the line in bash.

EXERCISE 1.2 The loop that breaks on spaces

In a folder holding `my notes.txt` and `todo.txt`, this loop prints three lines instead of two: `for f in $(ls *.txt); do echo "loop got [$f]"; done`. Explain why, and write a loop that's correct for any file name, including when there are no matching files.

SOLUTION

```

for f in *.txt; do
  [ -e "$f" ] || continue      # no match: bash leaves the literal "*.txt"
  echo "loop got [$f]"
done

```

The glob `*.txt` is expanded into two names for `ls`, which prints them, one per line. But the output of `$(ls *.txt)` is then split at every space and newline, so `my notes.txt` becomes two words, and the loop runs three times, with `my`, `notes.txt` and `todo.txt`. Looping over the glob itself gives each name as one word, spaces and all, because names produced by globbing are never split. The `[ -e "$f" ]` test handles the empty case: with no `.txt` files, the loop would otherwise run once with the literal `*.txt`. In bash, `shopt -s nullglob` makes an unmatched glob expand to nothing instead. And inside the loop, `"$f"` stays quoted wherever it's used.

EXERCISE 1.3 The empty variable

In 2015, Steam's installer for Linux ran a line like `rm -rf "$STEAMROOT/*"`. On some machines `STEAMROOT` turned out to be empty. What did the line expand to, and how should it have been written?

SOLUTION

With `STEAMROOT` empty, `"$STEAMROOT/*"` is the glob `/*`, every entry at the top of the filesystem. With `showargs` in place of `rm`, on the Mac this volume was written on, the line expands to seventeen arguments: `-rf`, then `/Applications`, `/Library`, `/System`, `/Users` and every other top-level folder. For an ordinary user, that deleted every file they owned, on every mounted drive, which is what the bug report described. The defences, in order of importance: `${STEAMROOT:?}` makes the shell stop with an error if the variable is unset or empty, instead of expanding to nothing; `set -u` at the top of a script makes any unset variable an error, which chapter 4 recommends; and checking a value before using it to delete anything, `[ -d "$STEAMROOT" ] || exit 1`, catches values that are set but wrong. Running a destructive line with `echo` in front first would have shown the problem on the developer's machine.

CHECKLIST

- ☐ Can you say exactly which list of arguments a program receives from a line?
- ☐ Can you explain the difference between single quotes, double quotes and no quotes?
- ☐ Can you name the shell's expansions in order, and say why command output gets globbed?
- ☐ Can you use `set -x` to see what a line really runs?
- ☐ Can you write a loop over files that works when names contain spaces?

GO DEEPER

COURSE [The Missing Semester: The Shell MIT](#)

DOCS [Bash Reference Manual: Shell Expansions GNU](#)

WIKI [Bash Pitfalls Greg's Wiki](#)

Streams, pipes and exit codes

3 exercises

Every program starts with three open streams, for its input, its output and its error messages, and finishes by returning a small number that says whether it worked. Redirection points those streams at files. A pipe connects one program's output to the next program's input, and the exit status lets the shell decide what to run next. Small programs that each do one thing become powerful through these three mechanisms. Two details cause most of the bugs: redirections are applied from left to right, and a pipeline reports only its last command's status.

Three streams and a number

When a program starts, it inherits three open **file descriptors**, small numbers that stand for open files: **0** is standard input, **1** standard output and **2** standard error. In a terminal all three are the terminal itself, which is why both output and error messages appear on screen. They're separate so that a program's results and its complaints can go to different places: results into a file or the next program, errors to a person. A well-behaved program writes what it computed to stdout and anything meant for a human, warnings, progress, errors, to stderr.

When it finishes, every program returns an **exit status** between 0 and 255. Zero means success, and every other value is some kind of failure; the shell keeps the last one in `$?`. A few values have conventional meanings: 1 for a general failure, 2 for misuse such as a bad option, 126 for a file that was found but isn't executable, 127 for a command that wasn't found, and 128 plus a signal's number for a program that was killed. That makes 130 a program stopped by Ctrl-C, signal 2, and 141 one killed by `SIGPIPE`, signal 13, which the example shows. The exit status is the only result every caller checks without reading anything.

Redirection

`> file` sends stdout to a file, creating it or emptying it first; `>> file` appends instead; `< file` reads stdin from a file; and putting a number in front picks the descriptor, so `2> errors.txt` redirects stderr. `2>&1` means "make descriptor 2 point wherever descriptor 1 points *now*". That last word is the whole difficulty. The shell processes redirections from left to right, each one changing a table of where the descriptors point. So `> out.txt 2>&1` first points 1 at the file, then points 2 at the same place: both go to the file. But `2>&1 > out.txt` first points 2 at the terminal, where 1 still points, and only then moves 1 to the file: errors stay on the screen. Bash's `&> file` is shorthand for the first form. `/dev/null` is a file that discards whatever is written to it.

```
talk > out.txt 2>&1
```

START



AFTER > out.txt



AFTER 2>&1



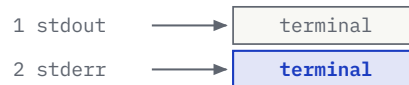
both lines land in out.txt

```
talk 2>&1 > out.txt
```

START



AFTER 2>&1



AFTER > out.txt



stderr still reaches the terminal

Redirections edit a table, left to right. `2>&1` copies where 1 points at that moment, so its position decides where errors go.

The example uses a tiny program, `talk`, that writes one line to stdout, one to stderr, and exits with whatever status it's given. For each line, it reports which text reached the terminal's stdout, which reached its stderr, and what ended up in any file.

EXAMPLE

streams.py · one program writing to both streams, run with redirections, pipes, tee, exit statuses, pipefail and PIPESTATUS, set -e and here-documents

```
import os, pathlib, shutil, subprocess, tempfile

# A tiny program that writes one line to each output stream and exits with the status it's
# given.
TALK = '#!/bin/sh\necho "to stdout"\necho "to stderr" >&2\nexit "${1:-0}"\n'

def show(script, cwd, env):
    """Run a script with bash and report where each piece of text ended up."""
    for f in cwd.glob("*.txt"): f.unlink()
    r = subprocess.run([shutil.which("bash"), "--nrc", "--noprofile", "-c", script],
                       cwd=cwd, env=env, capture_output=True, text=True)
    first, *rest = script.splitlines()
    print(f"  $ {first}" + "".join(f"\n  > {line}" for line in rest))
    places = [("the terminal's stdout", r.stdout), ("the terminal's stderr", r.stderr)]
    places += [(f"file {f.name}", f.read_text()) for f in sorted(cwd.glob("*.txt"))]
    for place, text in places:
        if text: print(f"      {place + ':':24}" + " | ".join(text.splitlines()))
    if r.returncode: print(f"      {'the script exited with:':24}status {r.returncode}")

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp); (root / "bin").mkdir(); (root / "work").mkdir()
    tool = root / "bin" / "talk"; tool.write_text(TALK); tool.chmod(0o755)
    env = {"PATH": f"{root / 'bin'}:{os.environ['PATH']}", "HOME": "/home/learner", "LC_ALL":
           "C"}
    work = root / "work"

    print("1. Every program starts with three streams: 0 stdin, 1 stdout, 2 stderr")
    for s in ["talk", "talk > out.txt", "talk 2> /dev/null"]: show(s, work, env)

    print("2. Redirections are applied left to right, and 2>&1 means 'make 2 point where 1
    points now'")
```

```

for s in ["talk > out.txt 2>&1", "talk 2>&1 > out.txt"]: show(s, work, env)

print("3. A pipe connects stdout only; stderr goes around it unless you redirect it")
for s in ["talk | tr a-z A-Z", "talk 2>&1 | tr a-z A-Z", "talk 2>&1 | tee all.txt | grep -c
std"]:
    show(s, work, env)

print("4. Every command exits with a status: 0 means success, anything else is a failure")
for s in ['talk 3 2> /dev/null; echo "status $?"',
          "talk 3 > /dev/null 2>&1 && echo worked || echo failed",
          'if grep -q needle /dev/null; then echo found; else echo "not found, status $?";
fi']:
    show(s, work, env)

print("5. A pipeline's status is its last command's, unless pipefail is set")
for s in ['false | true; echo "status $?"', 'set -o pipefail; false | true; echo "status
$?"',
          'false | true | talk 2 > /dev/null 2>&1; echo "statuses ${PIPESTATUS[*]}"',
          'yes | head -n 2; echo "statuses ${PIPESTATUS[*]}"']:
    show(s, work, env)

print("6. set -e stops at the first failure, except inside a condition, however deep")
for s in ['set -e; false; echo "never printed"',
          'set -e\ncheck() { false; echo "check carried on after false"; }\n'
          'check || echo "check failed"']:
    show(s, work, env)

print("7. A here-document feeds lines to stdin; quoting the marker turns expansion off")
for s in ["name=Ada\ncat <<EOF\nHello, $name\nEOF", "name=Ada\ncat <<'EOF'\nHello,
$name\nEOF"]:
    show(s, work, env)

```

OUTPUT

- Every program starts with three streams: 0 stdin, 1 stdout, 2 stderr


```

$ talk
the terminal's stdout: to stdout
the terminal's stderr: to stderr
$ talk > out.txt
the terminal's stderr: to stderr
file out.txt:         to stdout
$ talk 2> /dev/null
the terminal's stdout: to stdout

```
- Redirections are applied left to right, and 2>&1 means 'make 2 point where 1 points now'


```

$ talk > out.txt 2>&1
file out.txt:         to stdout | to stderr
$ talk 2>&1 > out.txt
the terminal's stdout: to stderr
file out.txt:         to stdout

```
- A pipe connects stdout only; stderr goes around it unless you redirect it


```

$ talk | tr a-z A-Z
the terminal's stdout: TO STDOUT
the terminal's stderr: to stderr
$ talk 2>&1 | tr a-z A-Z
the terminal's stdout: TO STDOUT | TO STDERR
$ talk 2>&1 | tee all.txt | grep -c std
the terminal's stdout: 2
file all.txt:         to stdout | to stderr

```
- Every command exits with a status: 0 means success, anything else is a failure


```

$ talk 3 2> /dev/null; echo "status $"
the terminal's stdout: to stdout | status 3
$ talk 3 > /dev/null 2>&1 && echo worked || echo failed
the terminal's stdout: failed
$ if grep -q needle /dev/null; then echo found; else echo "not found, status $?"; fi
the terminal's stdout: not found, status 1

```
- A pipeline's status is its last command's, unless pipefail is set

```

$ false | true; echo "status $?"
the terminal's stdout: status 0
$ set -o pipefail; false | true; echo "status $?"
the terminal's stdout: status 1
$ false | true | talk 2 > /dev/null 2>&1; echo "statuses ${PIPESTATUS[*]}"
the terminal's stdout: statuses 1 0 2
$ yes | head -n 2; echo "statuses ${PIPESTATUS[*]}"
the terminal's stdout: y | y | statuses 141 0
6. set -e stops at the first failure, except inside a condition, however deep
$ set -e; false; echo "never printed"
the script exited with: status 1
$ set -e
> check() { false; echo "check carried on after false"; }
> check || echo "check failed"
the terminal's stdout: check carried on after false
7. A here-document feeds lines to stdin; quoting the marker turns expansion off
$ name=Ada
> cat <<EOF
> Hello, $name
> EOF
the terminal's stdout: Hello, Ada
$ name=Ada
> cat <<'EOF'
> Hello, $name
> EOF
the terminal's stdout: Hello, $name

```

Pipes

A pipe, `a | b`, connects `a`'s stdout to `b`'s stdin. Only stdout: in section 3, `talk | tr a-z A-Z` upcased the stdout line, while the stderr line went around the pipe to the terminal, unchanged. To send both through, redirect first: `talk 2>&1 | tr a-z A-Z`. `tee file` copies its input to a file and passes it on, so you can save a stream and keep processing it. Both programs in a pipeline run at the same time, the second reading as the first writes, which is why a pipeline can process a file far larger than memory. It's also why `yes | head -n 2` finishes: `yes` prints `y` for ever, `head` exits after two lines, and the next write by `yes` finds no reader and kills it with `SIGPIPE`. Its status, 141, is in section 5.

Exit statuses, conditions and pipefail

The shell's conditions are built on exit statuses, not on true and false values. `a && b` runs `b` only if `a` succeeded, `a || b` only if it failed, and `if command; then` runs the branch when the command's status is 0. The `[ in if [ -f notes.txt ]` is itself a command, also called `test`, which exits 0 or 1. Programs like `grep -q` exist to be used as conditions: they print nothing and exit 0 if a line matched. One caution: `a && b || c` isn't if-then-else, because `c` also runs when `b` fails.

A pipeline's status is the status of its last command. `false | true` succeeds, which means a failure early in a pipeline, a download that broke halfway, a program that crashed, is silently ignored. `set -o pipefail` changes the rule: the pipeline fails if any command in it failed. And bash keeps every status from the last pipeline in the array `PIPESTATUS : 1 0 2` in section 5 are the statuses of `false`, `true` and `talk 2`. Like `$?`, it's overwritten by the next command, so read it immediately.

`set -e` makes a script exit when a command fails, and it has surprising exceptions. It doesn't apply to commands whose status is being tested, in an `if`, a `while`, or on the left of `&&` or `||`. And the exemption reaches into functions: in section 6, `check` was called on the left of `||`, so inside it `false` failed and the function carried on as if nothing had happened. `set -e` is still worth using, as chapter 4 does, but as a safety net. When a failure matters, check it explicitly: `cd "$dir" || exit 1`.

Here-documents

A **here-document** feeds the lines between `<<EOF` and a line containing just `EOF` to a command's stdin, which is the cleanest way to write a file or a block of text from a script. Variables and substitutions are expanded inside it, unless the marker is quoted, `<<'EOF'`, which section 7 shows. Bash also has the here-string, `grep x <<< "$text"`, which feeds one string as input. Any word can be the marker, and `EOF` is only a habit.

COMMON TRAPS

`cmd 2>&1 > file` when you meant both streams in the file: put `2>&1` last.

`sort data.txt > data.txt`, which empties the file: the shell truncates it before `sort` starts.

Reading `$_` too late, after another command, even an `echo`, has replaced it.

Pipelines without `pipefail`, where an early failure disappears behind a successful last command.

Trusting `set -e` inside functions that are called from a condition.

EXERCISES

EXERCISE 2.1 Where does each line go?

For the example's `talk`, say what reaches the terminal and what ends up in which file for: `talk > log 2>&1`, `talk 2>&1 > log`, `talk &> log` and `talk 2> err > out`.

SOLUTION

<code>talk > log 2>&1</code>	terminal: nothing	log: to stdout, to stderr
<code>talk 2>&1 > log</code>	terminal: to stderr	log: to stdout
<code>talk &> log</code>	terminal: nothing	log: to stdout, to stderr
<code>talk 2> err > out</code>	terminal: nothing	out: to stdout err: to stderr

Work through the table from left to right. In the second line, `2>&1` runs while 1 still points at the terminal, so 2 is pointed at the terminal, and the later `> log` moves only 1. `&>` is bash's shorthand for `> log 2>&1`; it doesn't exist in plain `sh`, so scripts meant for any shell spell it out. The last line sends each stream to its own file, and the order of the two redirections doesn't matter there because neither copies the other. Each answer was checked by running the line.

EXERCISE 2.2 The file that emptied itself

Someone sorts a list in place with `sort names.txt > names.txt`, and finds `names.txt` empty afterwards. Explain what happened, and give two correct ways to do it.

SOLUTION

```
sort -o names.txt names.txt           # sort can write over its own
input
sort names.txt > names.txt.new && mv names.txt.new names.txt # works for any command
```

Redirections are set up by the shell before the command starts, and `>` empties its file. So by the time `sort` opened `names.txt` to read it, it was already empty, and sorting nothing produced nothing: running the line leaves the file at 0 bytes. `sort -o` is safe because `sort` reads all its input before it opens the output. The general pattern, writing to a new file and renaming it over the old one only if the command succeeded, works for any program, and the `&&` means a failed command leaves the original untouched. The rename is atomic, as chapter 5 explains, so nothing ever sees a half-written file.

EXERCISE 2.3 The failure that disappeared

A build script runs `make 2>&1 | tee build.log`, and the job is marked successful even when `make` fails. Explain why, and fix it two ways: for the whole script, and for this one line.

SOLUTION

```
set -o pipefail                       # for the whole script: any failure fails the
pipeline
make 2>&1 | tee build.log

make 2>&1 | tee build.log              # for one line: check make's own status
status=${PIPESTATUS[0]}
[ "$status" -eq 0 ] || { echo "build failed with status $status" >&2; exit "$status";
}
```

The pipeline's status is `tee`'s, and `tee` succeeds whenever it can write the log, so `make`'s failure is lost. With `pipefail`, the pipeline fails if any part fails: running `false | wc -l` gives status 0 normally and 1 with `pipefail`. `PIPESTATUS` gives each command's status separately, so the script can report `make`'s own code. It has to be copied straight away, as here, because the next command replaces it. GitHub Actions adds `-o pipefail` for exactly this reason when a step names `bash` as its shell, but not when the shell is left to its default; a script shouldn't depend on where it's run.

CHECKLIST

- ☐ Can you name the three standard streams, and say what belongs on each?
- ☐ Can you predict where output goes for any order of redirections?
- ☐ Can you read an exit status, including 126, 127 and 128 plus a signal?
- ☐ Can you make a pipeline fail when any part of it fails?
- ☐ Can you say when `set -e` won't stop a script?

GO DEEPER

DOCS [Bash Reference Manual: Redirections GNU](#)

WIKI [Why doesn't set -e do what I expected? Greg's Wiki](#)

COURSE [The Missing Semester: Shell Tools and Scripting MIT](#)

Searching and transforming text

3 exercises

Most of what you'll want to ask a computer is already written down somewhere as text: in logs, configuration files, code and exported data. Four tools answer most of those questions. `find` chooses files, `grep` chooses lines, `sed` rewrites lines, and `awk` computes over the fields of each line. With `sort`, `uniq`, `cut` and `head` between them, a pipeline of five short commands answers questions that would take a page of code. A little knowledge of regular expressions makes all of them more precise. This chapter questions a web server's log and a project folder.

Regular expressions in ten minutes

A **regular expression** is a pattern that matches text. Most characters match themselves. A `.` matches any one character; `[abc]` matches one of a set, `[^abc]` anything outside it, and `[0-9]` a range. `^` and `$` anchor the pattern to the start and end of the line. Repetition applies to the thing just before it: `*` means zero or more, `+` one or more, `?` zero or one, and `{2,4}` between two and four. Parentheses group, and `|` means "or", so `^(GET|POST)/api/` matches lines that start with either word followed by `/api/`. A backslash makes a special character ordinary: `\.` is a real dot. Matching is **greedy**: `".*"` on `"a"` and `"b"` matches the whole thing, not just `"a"`, because `.*` takes as much as it can.

There are two dialects, and that causes confusion. **Basic** regular expressions, the default for `grep` and `sed`, treat `+`, `?`, `|`, `()` and `{}` as ordinary characters unless they have a backslash. **Extended** regular expressions, which you get with `grep -E` and `sed -E` and which `awk` always uses, give them their special meaning directly. Use `-E` everywhere and you only have to remember one. Named classes like `[[[:digit:]]` and `[[[:space:]]` work in both, and in every locale. Chapter 1's globs are a different language: there `*` means "anything" by itself.

grep chooses lines, find chooses files

`grep pattern file` prints the lines that match. The options you'll use constantly are `-i` to ignore case, `-v` for lines that don't match, `-c` to count them, `-n` to show line numbers, `-o` to print only the matching part, `-w` to match whole words, `-F` to treat the pattern as plain text, and `-r` to search a whole folder, with `-l` to list only the files. Like every good tool it reports through its exit status too: 0 if something matched, 1 if nothing did, 2 for an error, which is what makes `grep -q` useful in conditions.

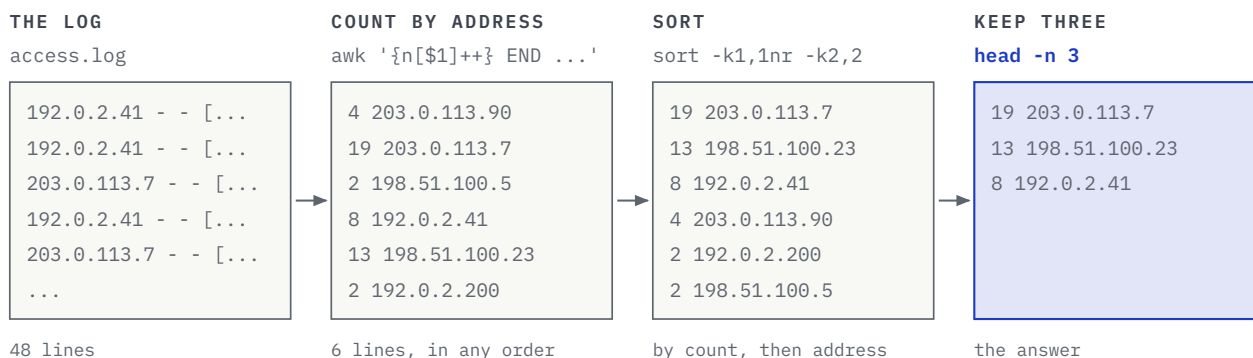
`find` walks a folder tree and tests each file against an expression. `-name '*.py'` matches names by glob, and must be quoted, or the shell expands it first. `-type f` chooses files and `-type d` folders. `-mtime +30` means last modified more than 30 days ago, and `-size +10M`

bigger than ten megabytes. Tests are combined with an implied "and", or with `-o` for "or". `-prune` stops `find` from going into a folder, which is how you skip `node_modules` or `.git`. The default action is to print the path. `-exec command {} +` runs a command on the files it found, with as many names per run as fit, and `-delete` deletes them.

sed rewrites lines, awk computes over fields

`sed`, the stream editor, applies editing commands to each line as it passes through. Almost all of its use is substitution: `s/old/new/` replaces the first match on each line, `s/old/new/g` every match, and parentheses in the pattern capture text that `\1`, `\2` put back in the replacement. A command can be limited to some lines by number, `3,5s/a/b/`, or by pattern, `/ERROR/s/a/b/`. With `-n`, `sed` prints nothing unless told to with `p`, so `sed -n '/404/p'` behaves like `grep`.

`awk` is a small programming language for tables of text. It splits each line into fields, `$1`, `$2` and so on to `$NF`, the last one, at runs of spaces or at the separator given with `-F`. A program is a list of `pattern {action}` rules: the action runs on lines that match the pattern, and a missing pattern matches every line. `BEGIN` and `END` rules run before the first line and after the last. Variables need no declaring, start at zero or empty, and can be associative arrays, so `{n[$1]++}` counts lines by their first field. `awk` was written in 1977 by Aho, Weinberger and Kernighan, whose initials give it its name, and their book about it was updated in 2023.



Each stage reads the previous one's output as it's produced; the four run at the same time.

A pipeline is a query: each stage narrows or reshapes the data and hands it on. The middle stages see six lines, not 48, and none of them needs to know where the data came from.

The example builds a web server's access log of 48 requests, each line holding the client's address, the time, the request, the status, the bytes sent and the milliseconds taken, plus a small project folder. Then it answers questions with one pipeline each.

EXAMPLE

text.py · a web server log and a project folder questioned with grep, cut, sort, uniq, awk, sed, find and xargs

```
import os, pathlib, random, shutil, subprocess, tempfile, time

def make_log(path):
    """A web server's access log: client, time, request, status, bytes sent, milliseconds
    taken."""
    rng = random.Random(3)
    clients = ["203.0.113.7"] * 9 + ["198.51.100.23"] * 6 + ["192.0.2.41"] * 4 + \
        ["203.0.113.90"] * 3 + \
        ["198.51.100.5", "192.0.2.200"]
    pages = [("GET", "/", 200), ("GET", "/api/orders", 200), ("POST", "/api/orders", 201),
        ("GET", "/api/users", 200), ("GET", "/static/app.js", 200), ("POST", "/login",
        302),
        ("GET", "/old-page", 404), ("GET", "/api/orders", 500), ("GET", "/api/report",
        503)]
    lines = []
    for i in range(48):
        method, page, status = rng.choice(pages)
        ms = rng.randint(900, 2400) if status >= 500 else rng.randint(8, 640)
        stamp = time.strftime("%d/%b/%Y:%H:%M:%S", time.gmtime(1773306000 + 37 * i))
        lines.append(f'{rng.choice(clients)} - - [{stamp} +0000] "{method} {page} HTTP/1.1" '
            f'{status} {rng.randint(120, 9000)} {ms}ms')
    path.write_text("\n".join(lines) + "\n")

def make_tree(root):
    """A small project folder, with a few files last changed long ago."""
    old = time.time() - 40 * 86400
    for name, text, aged in [("src/app.py", "# TODO: handle empty carts\n", False),
        ("src/util.py", "x = 1\n", True), ("docs/guide.md", "", True),
        ("tests/test_app.py", "# TODO: test refunds\n", False),
        ("build/out.tmp", "", False), ("old notes.tmp", "", True),
        ("node_modules/pkg/index.js", "", False), ("src/menu.js", "",
        False)]:
        p = root / name; p.parent.mkdir(parents=True, exist_ok=True); p.write_text(text)
        if aged: os.utime(p, (old, old))

def q(cmd, cwd, env):
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", cmd], cwd=cwd,
        env=env,
        capture_output=True, text=True)
    first, *rest = cmd.splitlines()
    print(f"  ${first}" + "".join(f"\n  > {line.strip()}" for line in rest))
    for line in r.stdout.splitlines(): print(f"    {line}")

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp); make_log(root / "access.log"); make_tree(root / "project")
    env = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C"}
    print("The log has lines like this one:")
    q("head -n 1 access.log", root, env)

    print("grep picks lines that match a regular expression")
    q("""grep -cE ' 5[0-9]{2} ' access.log""", root, env)
    q("""grep -E ' 5[0-9]{2} ' access.log | cut -d ' ' -f 1,7,9,11 | head -n 3""", root, env)
    q("""grep -oE '/api/[a-z]+' access.log | sort | uniq -c | awk '{print $1, $2}'""", root,
        env)

    print("awk splits each line into fields, $1 to $NF, and runs a little program on them")
    q("""awk '{n[$1]++} END {for (ip in n) print n[ip], ip}' access.log | sort -k1,1nr -k2,2 |
        head -n 3""",
        root, env)
    q("""awk '$NF + 0 > 1400 {print $7, $9, $NF}' access.log""", root, env)
```

```

q("""awk '{c = substr($9, 1, 1) "xx"; t[c] += $NF; n[c]++}
END {for (c in n) printf "%s %2d requests, average %4.0f ms\n", c, n[c], t[c] / n[c]}'
access.log |
sort""",
root, env)

print("sed edits a stream: here it hides the last two parts of each address, and finds
missing pages")
q("""sed -E 's/^([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/\1.x.x/' access.log |
cut -d ' ' -f 1,7 | head -n 3""",
root, env)
q("""sed -nE 's/.*"GET ([^ ]+) [^"]*" 404 .*/\1/p' access.log | sort -u""", root, env)

print("find walks a folder tree and selects files by name, type, age and more")
q("find . -name '*.py' | sort", root / "project", env)
q("find . -name node_modules -prune -o -name '*.js' -print", root / "project", env)
q("find . -type f -mtime +30 | sort", root / "project", env)
q("grep -rn --include='*.py' TODO . | sort", root / "project", env)

print("Names with spaces break naive pipelines; -print0 and xargs -0 separate names with a
zero byte")
q("find . -name '*.tmp' | xargs rm 2> /dev/null; find . -name '*.tmp' | sort", root /
"project", env)
q("find . -name '*.tmp' -print0 | xargs -0 rm; find . -name '*.tmp' | wc -l | tr -d ' '",
root / "project", env)

```

OUTPUT

The log has lines like this one:

```
$ head -n 1 access.log
192.0.2.41 - - [12/Mar/2026:09:00:00 +0000] "GET /api/users HTTP/1.1" 200 2256 614ms
```

grep picks lines that match a regular expression

```
$ grep -cE '" 5[0-9]{2}' access.log
5
$ grep -E '" 5[0-9]{2}' access.log | cut -d ' ' -f 1,7,9,11 | head -n 3
192.0.2.200 /api/report 503 1698ms
203.0.113.7 /api/report 503 1383ms
198.51.100.23 /api/report 503 1455ms
$ grep -oE '/api/[a-z]+' access.log | sort | uniq -c | awk '{print $1, $2}'
11 /api/orders
3 /api/report
4 /api/users
```

awk splits each line into fields, \$1 to \$NF, and runs a little program on them

```
$ awk '{n[$1]++} END {for (ip in n) print n[ip], ip}' access.log | sort -k1,1nr -k2,2 | head
-n 3
19 203.0.113.7
13 198.51.100.23
8 192.0.2.41
$ awk '$NF + 0 > 1400 {print $7, $9, $NF}' access.log
/api/report 503 1698ms
/api/report 503 1455ms
/api/orders 500 1454ms
$ awk '{c = substr($9, 1, 1) "xx"; t[c] += $NF; n[c]++}
> END {for (c in n) printf "%s %2d requests, average %4.0f ms\n", c, n[c], t[c] / n[c]}'
access.log |
> sort
2xx 25 requests, average 344 ms
3xx 6 requests, average 500 ms
4xx 12 requests, average 435 ms
5xx 5 requests, average 1387 ms
```

sed edits a stream: here it hides the last two parts of each address, and finds missing pages

```
$ sed -E 's/^([0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/\1.x.x/' access.log |
> cut -d ' ' -f 1,7 | head -n 3
192.0.x.x /api/users
192.0.x.x /login
203.0.x.x /
```

```

$ sed -nE 's/.*"GET ([^ ]+) [^"]*" 404 .*/\1/p' access.log | sort -u
/old-page
find walks a folder tree and selects files by name, type, age and more
$ find . -name '*.py' | sort
./src/app.py
./src/util.py
./tests/test_app.py
$ find . -name node_modules -prune -o -name '*.js' -print
./src/menu.js
$ find . -type f -mtime +30 | sort
./docs/guide.md
./old notes.tmp
./src/util.py
$ grep -rn --include='*.py' TODO . | sort
./src/app.py:1:# TODO: handle empty carts
./tests/test_app.py:1:# TODO: test refunds
Names with spaces break naive pipelines; -print0 and xargs -0 separate names with a zero byte
$ find . -name '*.tmp' | xargs rm 2> /dev/null; find . -name '*.tmp' | sort
./old notes.tmp
$ find . -name '*.tmp' -print0 | xargs -0 rm; find . -name '*.tmp' | wc -l | tr -d ' '
0

```

What the example shows

Each tool does one job, and the pipeline does the rest. `cut -d ' ' -f 1,7,9,11` picks fields by number from lines separated by single spaces, which is how it pulled the address, path, status and time from each server error. `grep -o` printed just the `/api/...` part of each line, ready for counting. `uniq -c` counts runs of identical *adjacent* lines, which is why it always follows a `sort`. It also pads its counts differently on macOS and Linux, so the example passes them through `awk '{print $1, $2}'` to print the same everywhere. That's a habit worth having for any output another program will read.

awk does the arithmetic. `n[$1]++` counted requests per address, and the `END` rule printed the totals, in an order `awk` doesn't promise, so `sort -k1,1nr -k2,2` put them in order: numerically by count, highest first, and by address when counts tie, so the result never depends on chance. In `$NF + 0 > 1400`, adding zero turned `1698ms` into the number 1698, because `awk` reads the leading digits of a string as its numeric value. The averages per status class came from two arrays and a `printf`, which formats numbers exactly as C's `printf` does.

Names with spaces need care. `xargs` reads names from its input and runs a command on them, but by default it splits its input at spaces, so `./old notes.tmp` became two names, `./old` and `notes.tmp`, neither of which exists. The file survived. `find -print0` ends each name with a zero byte, the one character a file name can't contain, and `xargs -0` splits only there. `find . -name '*.tmp' -exec rm {} +` avoids the problem entirely, by never turning names into text.

GNU, BSD and portability

Linux comes with the GNU versions of these tools, and macOS with the BSD ones. They agree on everything in this chapter's example, and differ in the corners. `sed -i` edits a file in place on Linux, while macOS needs `sed -i ''`. `grep -P`, for Perl-style patterns, is

GNU-only, as are many long options. `stat` and `date` take completely different flags. Scripts that must run on both should stick to the options POSIX defines, the standard both follow, or check which system they're on. On a Mac, Homebrew's `coreutils`, `gnu-sed` and `grep` packages install the GNU versions with a `g` in front, `gsed` and `ggrep`, if you want Linux behaviour. Two modern tools are worth installing anywhere: `rg` (ripgrep), a much faster `grep -r` that skips files ignored by Git, and `jq`, which does for JSON what `awk` does for columns.

COMMON TRAPS

`find . -name *.py` **unquoted**, which the shell expands before `find` sees it.

`uniq` **without** `sort`, which only merges duplicates that happen to be adjacent.

An **unescaped** `.` in a pattern, matching any character: `grep 2026.03` also matches `2026503`.

`sed -i` **in a script** that must run on both macOS and Linux.

Feeding names to plain `xargs` or to `for f in $(find ...)`, which split them at spaces.

EXERCISES

EXERCISE 3.1 Six commands

In 1986, Jon Bentley asked Donald Knuth to write a program that prints the most common words in a text, and Doug McIlroy reviewed the result. Knuth's answer was ten pages of carefully documented Pascal; McIlroy's review included a six-command pipeline that did the same job. Write that pipeline: print the ten most frequent words in `story.txt`, ignoring case and punctuation, and explain each stage.

SOLUTION

```
tr -cs 'A-Za-z' '\n' < story.txt |    # every run of non-letters becomes one newline: a
    word per line
tr 'A-Z' 'a-z' |                    # lower case, so "The" and "the" are the same
    word
sort |                              # identical words become adjacent
uniq -c |                           # count each run of identical lines
sort -rn |                          # highest count first
head -n 10                          # keep the top ten
```

`-c` makes `tr` act on characters *not* in the set, and `-s` squeezes each run of them into a single newline. On the two sentences "The cat sat on the mat. The dog sat on the log. A cat and a dog: the best of friends, or

the worst? The end." it puts the first with 7, then a five-way tie at 2. Ties come out in reverse alphabetical order, sat , on , dog , cat , a , because -r also reverses sort 's final whole-line comparison. McIlroy's point wasn't that Knuth's program was bad. It was that reusable tools, composed, solve most such problems in minutes, and that the pipeline is easy to change: sort -rn to sort -n gives the rarest words instead.

EXERCISE 3.2 A pattern that means what it says

A file contains the lines 2026-03-14 , 2026503914 , 2026.03.14 and meeting 2026-03-14 at 9 . How many lines does grep -c '2026.03' count? Write patterns that match lines consisting of exactly one date in the form 2026-03-14 , and lines that contain one anywhere.

SOLUTION

```
grep -c '2026.03' dates.txt           # 4: the dot matches any
      character
grep -cE '^[[0-9]]{4}-[[0-9]]{2}-[[0-9]]{2}$' dates.txt # 1: a whole line that is just a
      date
grep -cE '[[0-9]]{4}-[[0-9]]{2}-[[0-9]]{2}' dates.txt  # 2: a date anywhere in the line
```

All four lines match the first pattern: the dot matched a - , a 5 and a . , and the fourth line contains 2026-03 . To match a literal dot, write \. , or use grep -F for plain text. The anchors make the second pattern describe the whole line. Without them, as in the third, the pattern can match anywhere, so it also finds the date inside the meeting line. Neither pattern checks that the date is real: 2026-13-99 matches too. Regular expressions are good at shape and bad at meaning, so once you've found candidates, check them with a real date parser.

EXERCISE 3.3 Rename a whole tree

Rename every .jpeg file under pics , in any subfolder, to .jpg , without breaking on names with spaces and without overwriting a file that already exists.

SOLUTION

```
find pics -type f -name '*.jpeg' -exec sh -c '
  for f do
    mv -n -- "$f" "${f%.jpeg}.jpg"
  done' sh {} +
```

`find` hands the names straight to a small shell script, never as text to be split, so spaces are safe; `{}` + passes as many as fit at once, and the `sh` after the script fills `$0`, so the names arrive as `"$@"`, which `for f` `do` loops over. `${f%.jpeg}` removes the suffix, `mv -n` refuses to overwrite, and `--` protects names that start with a dash. Run on a tree with `a.jpeg`, `2025 trip/beach day.jpeg`, `2025 trip/b.JPEG` and `c.jpg`, it renamed the first two and left `b.JPEG` alone: `-name` is case-sensitive. `-iname` would find it, but then the suffix to strip could be any mix of cases. Try such a command with `echo` in front of `mv` first, and look at the list before letting it run.

CHECKLIST

- ☐ Can you write an extended regular expression with classes, anchors, groups and repetition?
- ☐ Can you find files by name, type, size and age, skipping folders you don't want?
- ☐ Can you count, rank and summarise a log with `sort`, `uniq` and `awk`?
- ☐ Can you rewrite text with `sed` substitutions and capture groups?
- ☐ Can you process file names safely with `-print0`, `xargs -0` or `-exec ... {} +`?

GO DEEPER

COURSE [The Missing Semester: Data Wrangling MIT](#)

DOCS [The GNU Awk User's Guide: Getting Started GNU](#)

GUIDE [The Art of Command Line Joshua Levy and contributors](#)

Scripts that don't break

3 exercises

A shell script is the cheapest automation there is: the commands you'd type, saved in a file. It's also the easiest to get dangerously wrong, because the shell carries on after errors, splits what it shouldn't and deletes what it's told. A handful of habits make the difference. Stop at the first failure, check the arguments, show what you'd do before doing it, never overwrite or delete without checking, clean up however the script ends, and make it safe to run twice. This chapter writes the kind of script the stage asks for, one that tidies a downloads folder, and puts it through its paces.

A script is a file of commands

A script starts with a **shebang** line, `#!/usr/bin/env bash`, which tells the system which program runs the file. `env` finds `bash` on your `PATH`, which matters on a Mac. Its `/bin/bash` is version 3.2, first released in 2006, kept because later versions use a licence Apple won't ship, so a newer `bash`, if you install one, lives elsewhere. Make the file executable with `chmod +x tidy`, and run it as `./tidy`, or put it in a folder on your `PATH`, such as `~/local/bin`, and run it by name. `bash tidy` works without the execute bit, because then `bash` only reads the file.

Inside, `$1`, `$2` and so on are the arguments, `$#` is how many there are, and `"$@"` is all of them, each kept as a separate word: it's the only correct way to pass a script's arguments on to another command. `shift` drops the first. For options, the built-in `getopts` handles `-n`, combined flags like `-nv` and options with values. A **function** is a named group of commands, `name() { ...; }`, called like any command, with its own `$1` and so on. A function's `return` sets its exit status, like a program's. To give back a value, it prints it, and the caller captures it with `$(name args)`. `local` keeps a variable inside the function instead of overwriting one outside.

Strict mode

Put `set -euo pipefail` near the top of every script. `-e` exits at the first command that fails, with the exceptions chapter 2 listed, instead of carrying on with the next line. `-u` makes using an unset variable an error, which turns a typo like `$flie`, or chapter 1's empty `$STEAMROOT`, into a stop instead of an empty string. `pipefail` makes a pipeline fail when any part of it does. Together they mean a script that hits something unexpected stops, and stopping is almost always the right thing for a script that runs without anyone watching. When you expect a command might fail, say so explicitly, with `|| true` or an `if`. And `${1:-}` reads a possibly missing argument without upsetting `-u`.

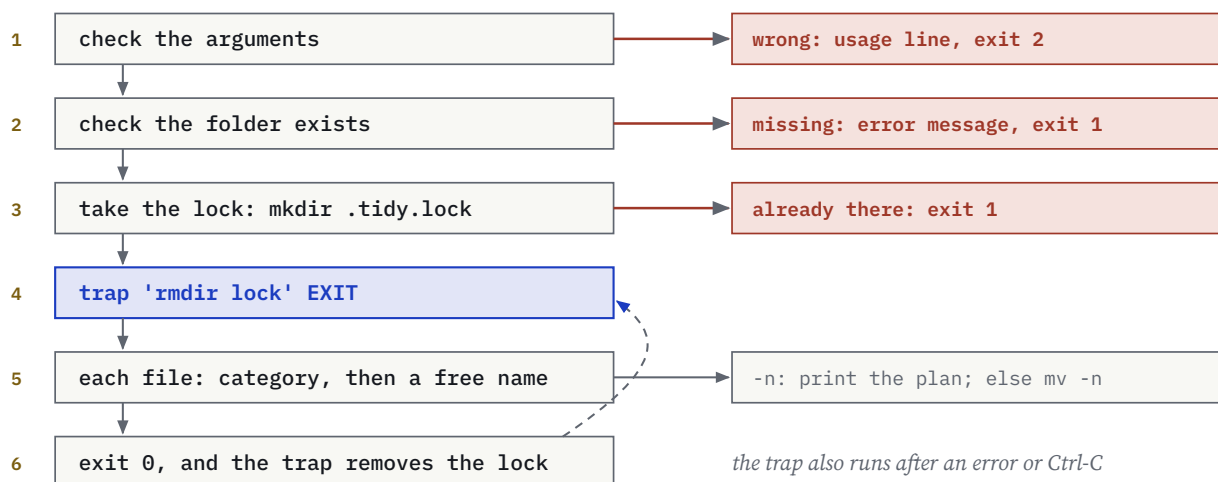
Failing well, and never destroying data

A script should fail the way a good program does. Messages go to stderr, with `>&2`, so they don't mix with output. Wrong usage prints a usage line and exits with status 2. Any other failure exits non-zero with a message that names the problem and the value that caused it: `tidy: no such folder: Nowhere`, not `error`. Check everything the script relies on before it starts changing things.

Then make destruction hard. Offer a **dry run** that prints what would happen, and use it first. Move rather than delete, and never overwrite: `mv -n` refuses to replace an existing file, and choosing a free name first, as the example's `free_name` does, means the refusal never comes up. Write new files under a temporary name and rename them into place only when complete, so nothing ever sees half a file. Before anything like `rm -rf "$dir"`, check that `$dir` is set, exists and is what you think it is. Finally, make the script **idempotent**, safe to run any number of times. Run twice, it should find nothing left to do rather than doing it all again or failing halfway.

Locks and cleanup

Two copies of a script working on the same files at once, perhaps from a timer and from you, will trip over each other. A lock prevents it, and `mkdir` makes a good one. Creating a folder either succeeds or fails, in one indivisible step, so exactly one copy can create `.tidy.lock` and the others see that it exists. The lock must be removed however the script ends, and that's what `trap` is for. `trap 'rmdir "$lock"' EXIT` runs its command when the script exits: normally, through `set -e` after an error, or after Ctrl-C. Only `kill -9`, which nothing can catch, skips it. The same pattern deletes temporary files made with `mktemp`.



Check first, then change. Every way out before the lock leaves nothing behind; every way out after it passes through the trap.

The example writes `tidy`, a script that sorts the files at the top of a folder into `images`, `documents`, `archives` and `other` by their extension. Then it runs it on a downloads folder in every way that matters: wrongly, as a dry run, for real, a second time, on a missing folder, and while another run holds the lock. The script is written for bash 3.2, so it runs unchanged on a Mac and on Linux.

EXAMPLE

script.py · a script that tidies a downloads folder, with strict mode, getopts, a dry run, no overwriting, a mkdir lock and a trap, run seven ways

```
import os, pathlib, shutil, subprocess, tempfile

TIDY = r'''#!/usr/bin/env bash
# tidy: sort the files at the top of a folder into subfolders by type. Safe to run again.
set -euo pipefail

usage() { echo "usage: tidy [-n] FOLDER    -n: only show what would move"; }

dry_run=false
while getopts "nh" opt; do
    case "$opt" in
        n) dry_run=true ;;
        h) usage; exit 0 ;;
        *) usage >&2; exit 2 ;;
    esac
done
shift $((OPTIND - 1))
[ "$#" -eq 1 ] || { usage >&2; exit 2; }
folder=${1%/}
[ -d "$folder" ] || { echo "tidy: no such folder: $folder" >&2; exit 1; }

# mkdir is atomic, so only one run can hold the lock; the trap removes it however the script
# ends
lock="$folder/.tidy.lock"
mkdir "$lock" 2> /dev/null || { echo "tidy: already running on $folder" >&2; exit 1; }
trap 'rmdir "$lock"' EXIT

category() {
    local ext
    ext=$(printf '%s' "${1##*.}" | tr '[:upper:]' '[:lower:]')
    case "$ext" in
        jpg|jpeg|png|gif|heic) echo images ;;
        pdf|doc|docx|txt|md|csv) echo documents ;;
        zip|tar|gz|dmg) echo archives ;;
        *) echo other ;;
    esac
}

free_name() {
    # $2, or "$2 (1)", "$2 (2)"... whichever doesn't exist yet in folder $1
    local dir=$1 name=$2 base ext n=1 candidate=$2
    if [[ $name == ?*.? ]]; then base=${name%.*} ext=${name##*.}; else base=$name ext=; fi
    while [ -e "$dir/$candidate" ]; do candidate="$base ($n)$ext"; n=$((n + 1)); done
    printf '%s\n' "$candidate"
}

count=0
for path in "$folder"/*; do
    [ -f "$path" ] || continue          # skip folders, and the unexpanded * of an empty folder
    name=${path##*/}
    dest=$(category "$name")
    target=$(free_name "$folder/$dest" "$name")
    if $dry_run; then
        echo "would move: $name -> $dest/$target"
    else
        mkdir -p "$folder/$dest"
        mv -n "$path" "$folder/$dest/$target"
        echo "moved: $name -> $dest/$target"
    fi
    count=$((count + 1))
done
```

```

[ "$count" -gt 0 ] || echo "nothing to tidy in $folder"
'''

def run(cmd, cwd, env):
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", cmd], cwd=cwd,
                        env=env,
                        capture_output=True, text=True)

    print(f"$ {cmd}")
    for line in r.stdout.splitlines(): print(f"    {line}")
    for line in r.stderr.splitlines(): print(f"    (stderr) {line}")
    if r.returncode: print(f"    (exit status {r.returncode})")

def tree(folder):
    for p in sorted(folder.rglob("*")):
        if p.is_file(): print(f"    {p.relative_to(folder.parent)}")

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp); (root / "bin").mkdir()
    script = root / "bin" / "tidy"; script.write_text(TIDY); script.chmod(0o755)
    downloads = root / "Downloads"
    for name in ["Holiday 2025.JPG", "invoice-0423.pdf", "report.pdf", "notes.txt",
                "photo.png",
                "backup.tar.gz", "Makefile", "documents/report.pdf", "projects/plan.md"]:
        p = downloads / name; p.parent.mkdir(parents=True, exist_ok=True); p.write_text(name)
    env = {"PATH": f"{root / 'bin'}:{os.environ['PATH']}", "HOME": "/home/learner", "LC_ALL":
          "C"}

    run("tidy", root, env)
    run("tidy -n Downloads", root, env)
    run("tidy Downloads/", root, env)
    print("After one run:"); tree(downloads)
    run("tidy Downloads", root, env)
    run("tidy Nowhere", root, env)
    run('mkdir Downloads/.tidy.lock; tidy Downloads || echo "refused, status $?"; '
        'rmdir Downloads/.tidy.lock', root, env)
    kept = (downloads / "documents/report.pdf").read_text() == "documents/report.pdf"
    print(f"no lock left behind: {not (downloads / '.tidy.lock').exists()}; "
          f"the report already in documents/ is untouched: {kept}")

```

OUTPUT

```

$ tidy
(stderr) usage: tidy [-n] FOLDER    -n: only show what would move
(exit status 2)
$ tidy -n Downloads
would move: Holiday 2025.JPG -> images/Holiday 2025.JPG
would move: Makefile -> other/Makefile
would move: backup.tar.gz -> archives/backup.tar.gz
would move: invoice-0423.pdf -> documents/invoice-0423.pdf
would move: notes.txt -> documents/notes.txt
would move: photo.png -> images/photo.png
would move: report.pdf -> documents/report (1).pdf
$ tidy Downloads/
moved: Holiday 2025.JPG -> images/Holiday 2025.JPG
moved: Makefile -> other/Makefile
moved: backup.tar.gz -> archives/backup.tar.gz
moved: invoice-0423.pdf -> documents/invoice-0423.pdf
moved: notes.txt -> documents/notes.txt
moved: photo.png -> images/photo.png
moved: report.pdf -> documents/report (1).pdf
After one run:
Downloads/archives/backup.tar.gz
Downloads/documents/invoice-0423.pdf
Downloads/documents/notes.txt
Downloads/documents/report (1).pdf
Downloads/documents/report.pdf

```

```

Downloads/images/Holiday 2025.JPG
Downloads/images/photo.png
Downloads/other/Makefile
Downloads/projects/plan.md
$ tidy Downloads
nothing to tidy in Downloads
$ tidy Nowhere
(stderr) tidy: no such folder: Nowhere
(exit status 1)
$ mkdir Downloads/.tidy.lock; tidy Downloads || echo "refused, status $?"; rmdir
Downloads/.tidy.lock
refused, status 1
(stderr) tidy: already running on Downloads
no lock left behind: True; the report already in documents/ is untouched: True

```

What the example shows

It stops before doing harm. With no folder, it printed its usage line to stderr and exited with status 2; with a missing folder, a precise message and status 1; with the lock already taken, it refused rather than racing the other run. None of those runs changed anything, because every check comes before the first move. `${1%/}` removes a trailing slash, so `tidy Downloads/` works too: the shell's Tab completion adds that slash, so users will type it.

The plan and the run agree, and nothing is lost. The dry run printed exactly the seven moves the real run then made. The report that was already in `documents` kept its name, and the new one became `report (1).pdf`. The subfolder `projects` was skipped because the loop only moves plain files. The order, capital letters first, is the C locale's byte order, the same everywhere. A second run found nothing to tidy, and no lock was left behind. Category names come from a `case` statement rather than an associative array, because those need bash 4.

Testing a script, and knowing when to stop

Test scripts like any program. `bash -n script` checks the syntax without running anything, and `bash -x script` traces every command, as chapter 1's `set -x` does. **ShellCheck** reads a script and points out unquoted variables, useless `cat s`, `cd s` without error handling and dozens of other known mistakes. Run it on everything you write. Tests can be shell scripts too, with a framework such as `bats`, or a Python program like the example, which builds a folder, runs the script and checks the result.

Shell is the right language for gluing programs together and the wrong one for nearly everything else. Google's style guide for shell puts the line at about a hundred lines, or any non-trivial control flow. Past that point, or once you need data structures, arithmetic on anything but whole numbers, JSON, or careful error handling, rewrite it in Python: its `subprocess`, `pathlib` and `shutil` modules do everything a script does, without the quoting.

No `set -u`, so a misspelt variable silently expands to nothing.

`cd somewhere` without `|| exit`, so the rest of the script runs in the wrong folder.

Parsing the output of `ls` instead of looping over a glob.

Results and messages on the same stream, so another program can't use the results.

Bash 4 features, like associative arrays and `${x,,}`, in a script that must run on a Mac's bash 3.2.

EXERCISES

EXERCISE 4.1 An undo

Make `tidy` keep a record of its moves, and write `untidy FOLDER`, which puts every file back where it came from, newest move first.

SOLUTION

```
# in tidy, after the mv:
printf '%s\t%s\n' "$dest/$target" "$name" >> "$folder/.tidy-log"

#!/usr/bin/env bash
# untidy: undo tidy's moves in FOLDER, newest first, using the log tidy kept.
set -euo pipefail
folder=${1:?usage: untidy FOLDER}
log="$folder/.tidy-log"
[ -f "$log" ] || { echo "untidy: nothing to undo in $folder" >&2; exit 1; }
awk '{ line[NR] = $0 } END { for (i = NR; i > 0; i--) print line[i] }' "$log" |
while IFS=$'\t' read -r moved original; do
    mv -n -- "$folder/$moved" "$folder/$original" && echo "restored: $original"
done
rm -- "$log"
```

Each line of the log holds the new and old names, separated by a tab, which file names almost never contain. The `awk` program reverses the lines, because GNU's `tac` and BSD's `tail -r` each exist on only one system. Undoing newest first matters when one name was used twice.

`IFS=$'\t' read -r` splits each line at the tab only, and `-r` keeps backslashes as they are. Run on a folder with three files, `tidy` moved them, `untidy` restored all three, and a second `untidy` said there was nothing to undo. The empty `images` and `documents` folders stay behind; removing them only if they're empty, with `rmdir`, would be a careful improvement. A name containing a newline would still break the log, which is the honest limit of line-based records.

EXERCISE 4.2 Find the bugs

List everything wrong with this script, which compresses old logs, and rewrite it.

```
#!/bin/bash
backup_dir=$1
cd $backup_dir
for f in $(ls *.log); do
    gzip $f
done
rm -rf $backup_dir/old/*
echo "done"
```

SOLUTION

```
#!/usr/bin/env bash
set -euo pipefail
[ "$#" -eq 1 ] || { echo "usage: compress-logs FOLDER" >&2; exit 2; }
backup_dir=$1
[ -d "$backup_dir/old" ] || { echo "compress-logs: no old/ in $backup_dir" >&2; exit
    1; }
cd -- "$backup_dir"
for f in *.log; do
    [ -e "$f" ] || continue
    gzip -- "$f"
done
find old -mindepth 1 -delete      # empty old/, here and only here
echo "done" >&2
```

The original has no strict mode, so it carries on after any failure. With no argument, `backup_dir` is empty and `cd $backup_dir` goes to your home folder; with a missing folder, `cd` fails and the script continues in whatever folder it was started from. `$(ls *.log)` splits names at spaces, and the unquoted `$f` splits them again. Worst, with an empty `backup_dir`, the `rm` line becomes `rm -rf /old/*`, and with a relative path like `logs` it points at `logs/old` *inside* the folder it has already moved into. The rewrite checks its argument and the folder before changing anything, uses paths relative to the folder it has entered, quotes everything, and deletes with `find old -mindepth 1 -delete`, which can only touch `old`'s contents. The message goes to stderr, and in a script run by a timer it could be dropped entirely.

EXERCISE 4.3 Shell or Python?

For each task, decide whether to write a shell script or a Python program, and say why: (a) run the tests, then the linter, and deploy only if both pass; (b) rename photos to the date they were taken, read from inside each file; (c) call a web API every hour, retry when it fails, and save one field of the JSON reply.

SOLUTION

(a) **Shell.** It's pure glue: three programs in order, each decision made by an exit status. `set -euo pipefail` followed by the three commands, each on its own line, is the whole script, and anything else would be longer. (b) **Python.** The date is inside the image's metadata, which needs a library to read, and the renaming needs care with collisions and time zones that shell makes awkward. Shell could call a tool like `exiftool`, but by the time it handles a missing date and two photos taken in the same second, it's past the point where shell is clear. (c) **Python,** using the standard library's `urllib` and `json`, or `requests`. Retrying with a delay that grows, telling a timeout from an error, and reading JSON are all things shell does badly. `curl` and `jq` can do a simple version, but error handling is where the shell version always grows a hundred lines. The rule: when you find yourself fighting the shell's quoting or reaching for its arrays, switch.

CHECKLIST

- ☐ Can you write a script with a shebang, strict mode, a usage message and `getopts`?
- ☐ Can you pass arguments on correctly with `"$@"`, and return values from functions?
- ☐ Can you add a dry run, avoid overwriting files, and make a script safe to run twice?
- ☐ Can you take a lock with `mkdir` and clean up with `trap ... EXIT`?
- ☐ Can you check a script with ShellCheck, `bash -n` and `bash -x`, and say when to switch to Python?

GO DEEPER

TOOL ShellCheck: finds bugs in your shell scripts Vidar Holen

GUIDE Shell Style Guide Google

WIKI The Bash Guide Greg's Wiki

PART II

The machine

Four chapters on what lies under the shell. Files and the names that point at them; who may read, change and run each one; the processes that run programs, and the environment each one inherits, including the `PATH` that decides which program a name means. The part ends with dotfiles, the small text files that configure your tools, kept in a repository with a script that installs them, which is the first half of the deliverable.

Files, paths and links

3 exercises

*A filesystem is a tree of names, and each name points to an **inode**, the record that holds a file's data, size, owner and permissions, but not its name. A path is directions through the tree. A **hard link** is a second name for the same inode, and a **symbolic link** is a small file that holds a path. Keeping those apart explains several everyday puzzles: why renaming a huge file is instant, why a symbolic link can break when nothing it points to was touched, and why deleting a log file doesn't always free the space.*

One tree, and directions through it

On Linux and macOS every file is somewhere in a single tree whose root is `/`. Other disks don't get letters as on Windows; they're **mounted** into the tree at a folder, such as `/Volumes/Backup` on a Mac or `/mnt/data` on Linux. A path that starts with `/` is **absolute**: directions from the root. Any other path is **relative**: directions from the current working directory, which `pwd` prints and `cd` changes. Every folder contains two special entries: `.`, the folder itself, and `..`, its parent. Section 1 of the example resolves a few paths the way the system does: `../data/./in.csv` from `/home/learner/project` is `/home/learner/data/in.csv`. `cd -` returns to the previous folder, and `~` is your home folder.

Names starting with a dot are hidden from `ls` and from `*`, which is all "hidden" means. Linux treats `README.md` and `readme.md` as different names. A Mac's disk, by default, does not: it remembers the case you typed, but finds the file whatever case you ask for. So a Git repository containing both names, which is fine on Linux, breaks on a Mac, where only one of them can exist.

Names and inodes

A folder is a table that maps names to inode numbers, and the inode holds everything else about the file. `ls -li` shows the numbers, and `stat` shows the rest. Because the name isn't part of the file, renaming a file within one filesystem only rewrites an entry in a folder's table, which is why `mv` renames a ten-gigabyte video instantly. Moving it to another disk copies all the data and then deletes the original, because inode numbers only mean something within one filesystem. The rename is also **atomic**: anyone looking sees either the old name or the new one, never something in between. That's why "write a new file, then rename it over the old one" is the safe way to replace a file, as chapter 2 used.

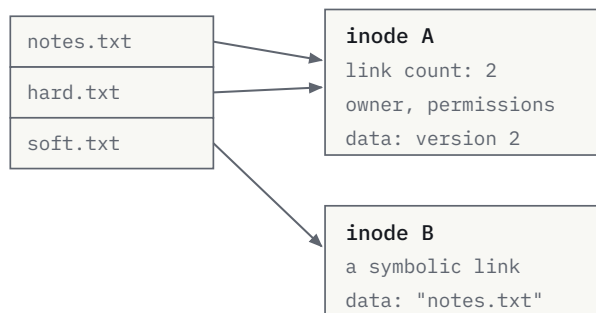
Hard links and symbolic links

`ln existing newname` makes a **hard link**: another entry, in the same or another folder, pointing at the same inode. Both names are equally real; neither is the original. The inode keeps a **link count**, the number of names pointing at it, and the file's data is freed only when that reaches zero. Hard links can't cross filesystems, and you can't make them to folders.

`ln -s target linkname` makes a **symbolic link**, or symlink: a tiny file whose contents are a path. Whenever something opens the link, the system reads that path and follows it, so a symlink can point at a folder, at another disk, or at nothing at all, in which case it's **broken**, or dangling. `ls -l` shows a symlink as `name -> target`, and `readlink` prints the stored path. A relative target is resolved from the folder the *link* is in, not from wherever you were when you made it, which is the most common way dotfile links go wrong.

BEFORE

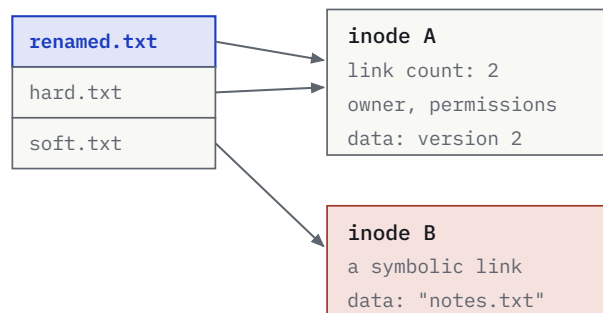
the folder's table



looked up by name on every use

AFTER `mv notes.txt renamed.txt`

the folder's table



that name is gone: broken

Names point to inodes; a symbolic link points to a name. Renaming touches only the table, so a hard link never notices, and a symbolic link that relied on the old name breaks.

The example makes the commands' effects visible. It runs each command in bash, then inspects the results with Python's `os` module, labelling inodes A, B and so on, because the real numbers differ on every machine.

EXAMPLE

files.py · paths resolved, hard and symbolic links made, written through, renamed and deleted, a relative link resolved, and a deleted file still open

```

import os, pathlib, posixpath, shutil, subprocess, tempfile

def sh(cmd, cwd):
    """Run a shell command; the rest of the example inspects what it did with Python's os
    module."""
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", cmd], cwd=cwd,
                       env={"PATH": os.environ["PATH"], "LC_ALL": "C"}, capture_output=True,
                           text=True)
    print(f"$ {cmd}" + (f" (exit status {r.returncode})" if r.returncode else ""))
    for line in r.stdout.splitlines(): print(f" {line}")

def describe(d, *names):
  
```

```

"""For each name: what kind of entry it is, which inode it names, and how many names that
inode has."""
inodes = {}
for name in names:
    p = d / name
    if p.is_symlink():
        target = os.readlink(p)
        state = "works" if p.exists() else "is broken: nothing at that path"
        print(f"    {name:12} symbolic link to {target!r}, which {state}")
    elif p.exists():
        st = p.stat(); label = inodes.setdefault(st.st_ino, chr(ord("A") + len(inodes)))
        print(f"    {name:12} file, inode {label}, {st.st_nlink} name(s), "
              f"contains {p.read_text().strip()!r}")
    else:
        print(f"    {name:12} does not exist")

print("1. A path is directions through the tree; . is here, .. is one level up")
for cwd, path in [("/home/learner/project", "notes.txt"), ("/home/learner/project",
    "../data/./in.csv"),
    ("/home/learner/project", "/etc/hosts"), ("/", "home/learner/../../tmp")]:
    print(f"    from {cwd:22} {path:24} is {posixpath.normpath(posixpath.join(cwd, path))}")

with tempfile.TemporaryDirectory() as tmp:
    d = pathlib.Path(tmp)
    print("2. A hard link is a second name for the same inode; a symbolic link is a file
        holding a path")
    sh("echo 'version 1' > notes.txt; ln notes.txt hard.txt; ln -s notes.txt soft.txt", d)
    describe(d, "notes.txt", "hard.txt", "soft.txt")
    sh("echo 'version 2' > hard.txt", d)
    describe(d, "notes.txt", "soft.txt")

    print("3. Renaming changes a name, not the file; symbolic links follow names, so they can
        break")
    sh("mv notes.txt renamed.txt", d)
    describe(d, "renamed.txt", "hard.txt", "soft.txt")
    sh("rm renamed.txt", d)
    describe(d, "hard.txt", "soft.txt")

    print("4. A relative link is resolved from the folder the link is in, not from where you
        made it")
    (d / "dotfiles").mkdir(); (d / "home").mkdir(); (d / "dotfiles" / "zshrc").write_text("# my
        settings\n")
    sh("ln -s dotfiles/zshrc home/.zshrc; ln -s ../dotfiles/zshrc home/.zshrc-fixed", d)
    describe(d / "home", ".zshrc", ".zshrc-fixed")

    print("5. Deleting removes a name; the data is freed only when no name and no open file
        refers to it")
    log = open(d / "big.log", "w"); log.write("x" * 1_000_000); log.flush()
    sh("rm big.log; ls big.log 2> /dev/null || echo 'no such name any more'", d)
    st = os.fstat(log.fileno())
    print(f"    but the program that opened it still has it: {st.st_size:,} bytes, {st.st_nlink}
        names")
    log.close()

```

OUTPUT

```

1. A path is directions through the tree; . is here, .. is one level up
    from /home/learner/project notes.txt          is /home/learner/project/notes.txt
    from /home/learner/project ../data/./in.csv    is /home/learner/data/in.csv
    from /home/learner/project /etc/hosts          is /etc/hosts
    from /home/learner/project home/learner/../../tmp is /tmp

2. A hard link is a second name for the same inode; a symbolic link is a file holding a path
$ echo 'version 1' > notes.txt; ln notes.txt hard.txt; ln -s notes.txt soft.txt
notes.txt    file, inode A, 2 name(s), contains 'version 1'
hard.txt     file, inode A, 2 name(s), contains 'version 1'
soft.txt     symbolic link to 'notes.txt', which works

```

```

$ echo 'version 2' > hard.txt
notes.txt    file, inode A, 2 name(s), contains 'version 2'
soft.txt     symbolic link to 'notes.txt', which works
3. Renaming changes a name, not the file; symbolic links follow names, so they can break
$ mv notes.txt renamed.txt
renamed.txt  file, inode A, 2 name(s), contains 'version 2'
hard.txt     file, inode A, 2 name(s), contains 'version 2'
soft.txt     symbolic link to 'notes.txt', which is broken: nothing at that path
$ rm renamed.txt
hard.txt     file, inode A, 1 name(s), contains 'version 2'
soft.txt     symbolic link to 'notes.txt', which is broken: nothing at that path
4. A relative link is resolved from the folder the link is in, not from where you made it
$ ln -s dotfiles/zshrc home/.zshrc; ln -s ../dotfiles/zshrc home/.zshrc-fixed
.zshrc       symbolic link to 'dotfiles/zshrc', which is broken: nothing at that path
.zshrc-fixed symbolic link to '../dotfiles/zshrc', which works
5. Deleting removes a name; the data is freed only when no name and no open file refers to it
$ rm big.log; ls big.log 2> /dev/null || echo 'no such name any more'
no such name any more
but the program that opened it still has it: 1,000,000 bytes, 0 names

```

What the example shows

A hard link is the same file. After `ln notes.txt hard.txt`, both names showed inode A and two names; writing `version 2` through `hard.txt` changed what `notes.txt` contains, because there's only one file. Renaming `notes.txt` left inode A and its contents exactly as they were. Deleting `renamed.txt` dropped the count to one name, and `hard.txt` still holds the data. The symlink went the other way. It held the path `notes.txt` all along, so the moment that name stopped existing, the link broke, though nothing it once reached had been deleted at the time.

A relative link is read from where it lives. In section 4, both links were made from the same folder, but stored different text. `home/.zshrc` holds `dotfiles/zshrc`, which, read from inside `home`, means `home/dotfiles/zshrc`, which doesn't exist. `../dotfiles/zshrc` climbs out of `home` first, and works. And section 5 showed the reason a full disk can survive a cleanup. After `rm big.log` the name was gone, but the program that had the file open could still use all million bytes, and the inode had zero names. The space returns only when that program closes the file or exits.

Where things live

The layout of the tree is conventional, and knowing it saves searching. Programs live in `/usr/bin` and `/bin`, and ones you install yourself in `/usr/local/bin`; on Apple silicon Macs, Homebrew uses `/opt/homebrew`. System configuration lives in `/etc`, logs in `/var/log`, temporary files in `/tmp`. Your own files and settings are under your home folder, `/Users/you` on a Mac and `/home/you` on Linux, and many programs keep their settings in `~/.config`, which chapter 8 uses. `df -h` shows how full each mounted filesystem is, and `du -sh folder` how much a folder uses; when they disagree, a deleted file that's still open is the usual reason.

A relative symlink whose target was written relative to the wrong folder.

Deleting a big log file that a running program still has open, and expecting the space back.

Two names that differ only in case, which are one file on a Mac.

Assuming `mv` is instant and safe everywhere: across disks it copies, and can be interrupted halfway.

Editing a file through a hard link and forgetting the other name changes too.

EXERCISES

EXERCISE 5.1 Which names share a file?

After `echo a > f; ln f g; ln -s f s; cp f c; mv f f2`, which names exist, which share an inode, what is each one's link count, and does `s` work?

SOLUTION

```
f      gone: renamed to f2
f2     inode A, 2 names
g      inode A, 2 names
s      symbolic link to "f", broken
c      inode B, 1 name: a copy is a new file
```

`ln f g` gave inode A a second name. `cp` made a new inode with the same contents, so `c` is independent from then on. `mv f f2` renamed one of A's two names, leaving the count at two, and since `s` holds the text `f`, it broke. Running the commands confirms all five lines. Note that changing `f2` or `g` changes both, while changing `c` changes only `c`. That difference matters for backups: a "backup" made by hard-linking your files isn't one, because editing the original edits the backup too.

EXERCISE 5.2 The disk that stayed full

A server's disk is at 100%. You delete a 40-gigabyte log file, and `df` still says 100%, while `du` finds only 2 gigabytes in use. What's going on, and how do you fix it?

SOLUTION

```

sudo lsof +L1          # open files with a link count below 1: deleted, but
                        still open
: > /proc/1234/fd/7     # Linux: empty the deleted file through the process
                        holding it
sudo systemctl restart myapp # or restart the program, which closes it

```

`du` adds up the files it can find by name, and `df` asks the filesystem how many blocks are in use. The difference is a file with no name that's still in use: the program writing the log still has it open, so `rm` removed the name, but the inode and its 40 gigabytes stay until the program closes it. `lsof +L1` lists exactly such files, with the process and its file descriptor. Restarting the program frees the space; on Linux you can also empty the file through `/proc`, without a restart. The lesson for next time: to shrink a log a program is writing, truncate it with `: > app.log` instead of deleting it, or better, configure log rotation, which renames the file and tells the program to reopen it.

EXERCISE 5.3 The dotfile link

Your settings are in `~/dotfiles/zshrc`. Which of these make a working `~/zshrc`: (a) `cd ~/dotfiles && ln -s zshrc ~/zshrc`, (b) `cd ~ && ln -s dotfiles/zshrc .zshrc`, (c) `cd /tmp && ln -s dotfiles/zshrc ~/zshrc`, (d) `cd ~/dotfiles && ln -s "$PWD/zshrc" ~/zshrc`?

SOLUTION

(a) is broken; (b), (c) and (d) work. Only the stored text matters, read from the link's folder, `~`. In (a) the link holds `zshrc`, so it points at `~/zshrc`, which doesn't exist. Being in `~/dotfiles` when you typed the command made no difference. (b) and (c) store `dotfiles/zshrc`, which from `~` means `~/dotfiles/zshrc`, whichever folder you were in. (d) stores an absolute path, which works from anywhere but breaks if you move the repository, while a relative link keeps working as long as the link and its target move together. Making all four confirms it. Install scripts, like chapter 8's, usually store absolute paths and recreate the links if the repository moves.

CHECKLIST

- ☐ Can you resolve any relative path to an absolute one in your head?
- ☐ Can you explain what an inode holds, and why renaming is instant and atomic?
- ☐ Can you say how hard links and symbolic links differ, and when each breaks?

- ☐ Can you write a relative symlink that points where you mean?
- ☐ Can you find a deleted file that's still holding disk space?

GO DEEPER

BOOK **The Linux Command Line** William Shotts

BOOK **Operating Systems: Three Easy Pieces, Files and Directories** Arpaci-Dusseau
& Arpaci-Dusseau

DOCS **Filesystem Hierarchy Standard** The Linux Foundation

Permissions

3 exercises

Every file has an owner, a group, and nine permission bits: read, write and execute, for the owner, for the group, and for everyone else. On a file the bits mean what they say. On a folder they mean something different, and that's where the surprises live: deleting a file needs permission on its folder, not on the file itself. This chapter decodes the bits, tries what each one allows, and covers the commands that change them, the default that sets them, and the account that ignores them.

An owner, a group and nine bits

`ls -l` starts each line with ten characters, such as `-lwxl-x---`. The first is the type: `-` for a file, `d` for a folder, `l` for a symbolic link. The other nine are three groups of three, `r` for read, `w` for write and `x` for execute, with `-` for a missing permission. The first group applies to the file's **owner**, the second to members of its **group**, and the third to everyone else, called **others**. `ls -l` also shows the owner's and group's names, and `id` shows who you are and which groups you're in.

Each group of three is also a number from 0 to 7, adding 4 for `r`, 2 for `w` and 1 for `x`, so the whole mode is three digits: `lwxl-x---` is 750. You'll meet a few constantly: 644 for ordinary files, 755 for programs and folders, 600 for private files and 700 for private folders. Exactly one group applies to you, checked in order: the owner's if you own the file, otherwise the group's if you're in its group, otherwise others'. So an owner with fewer rights than the group gets the owner's rights.

-	r	w	x	r	-	x	-	-	-	= 750
TYPE			OWNER			GROUP			OTHERS	
a file			4+2+1 = 7			4+0+1 = 5			0+0+0 = 0	

BIT	ON A FILE	ON A FOLDER
r	read its contents	list the names in it
w	change its contents	create, delete and rename entries in it
x	run it as a program	go into it, and reach files inside by name

Deleting a file changes its folder's list of names, so the folder's `w` decides who may.

Ten characters, one number. The same three letters mean different things on folders, and folder permissions decide who can delete a file.

Changing them, and the default

`chmod` changes the bits, either with a number, `chmod 640 plan.txt`, or symbolically: `u`, `g`, `o` and `a` for user, group, others and all, then `+`, `-` or `=`, then the letters. So `chmod u+x` makes a file executable for its owner, and `chmod go-r` stops anyone else reading it. `-R` applies a change to a whole tree, and a capital `X` adds execute only to folders and to files that were already executable, which is usually what you want. `chown` changes the owner, which only the administrator can do, and `chgrp` the group.

New files get their bits from the **umask**, a mask of permissions to remove from everything a program creates. Programs ask for 666 for files and 777 for folders, and the umask takes bits away. The usual umask, 022, removes write for the group and others, giving 644 and 755. A umask of 077 makes everything private, 600 and 700, which section 2 of the example shows. `umask` with no argument prints yours; set it in your shell's startup file, as chapter 8 does, to change it everywhere.

The example sets bits with `chmod` and `umask` and reads them back, then tries real operations as an ordinary user and reports only whether each was allowed. The exact wording of "Permission denied" differs between systems, but the decisions don't.

EXAMPLE

perms.py · modes decoded, chmod and umask, then reading, writing and running files, and listing, entering and deleting in folders, with the bits taken away one at a time

```
import os, pathlib, shutil, stat, subprocess, tempfile

if os.geteuid() == 0:
    raise SystemExit("Run this as an ordinary user: root may do almost anything, whatever the bits say.")

def sh(cmd, cwd):
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", cmd], cwd=cwd,
                       env={"PATH": os.environ["PATH"], "LC_ALL": "C"}, capture_output=True,
                       text=True)
    return r.returncode, r.stdout.strip()

def mode(p): return stat.filemode(os.lstat(p).st_mode)

def attempt(what, cmd, cwd):
    status, out = sh(cmd, cwd)
    if status: result = f"refused (status {status})"
    else: result = "allowed" + (f", printed {out!r}" if out else "")
    print(f"    {what:42} {result}")

print("1. Nine bits: read, write, execute, for the owner, the group, and everyone else")
for bits in (0o644, 0o755, 0o600, 0o750, 0o400):
    who = [stat.filemode(stat.S_IFREG | bits)[i:i + 3] for i in (1, 4, 7)]
    print(f"    {bits:o} {stat.filemode(stat.S_IFREG | bits)} "
          f"owner {who[0]} group {who[1]} others {who[2]}")

with tempfile.TemporaryDirectory() as tmp:
    d = pathlib.Path(tmp)
    print("2. chmod sets bits by number or by name; umask removes bits from everything you create")
    (d / "plan.txt").write_text("plan\n"); os.chmod(d / "plan.txt", 0o644)
    for change in ("u+x", "go-r", "a+r", "640"):
        os.chmod(d / "plan.txt", os.stat(d / "plan.txt").st_mode | int(change, 8))
```

```

sh(f"chmod {change} plan.txt", d)
print(f"  chmod {change:5} plan.txt  now {mode(d / 'plan.txt')}")
sh("umask 022; touch a.txt; mkdir a-dir; umask 077; touch b.txt; mkdir b-dir", d)
for name in ("a.txt", "a-dir", "b.txt", "b-dir"):
    print(f"    created with umask {'022' if name[0] == 'a' else '077'}: {name:6} {mode(d /
        name)}")

print("3. On a file, r lets you read it, w lets you change it, x lets you run it")
(d / "secret.txt").write_text("42\n"); os.chmod(d / "secret.txt", 0o200)
attempt("cat secret.txt (mode --w-----)", "cat secret.txt", d)
(d / "fixed.txt").write_text("v1\n"); os.chmod(d / "fixed.txt", 0o444)
attempt("append to fixed.txt (mode r--r--r--)", "echo v2 >> fixed.txt", d)
(d / "hello.sh").write_text("echo hello from the script\n"); os.chmod(d / "hello.sh",
    0o644)
attempt("./hello.sh (mode rw-r--r--)", "./hello.sh", d)
attempt("bash hello.sh (bash only reads it)", "bash hello.sh", d)

print("4. On a folder, r lets you list names, w lets you add and remove names, x lets you
    go in")
box = d / "box"; box.mkdir(); (box / "note.txt").write_text("inside\n")
os.chmod(box, 0o600)
attempt("echo box/* (mode rw-----: no x)", "echo box/*", d)
attempt("cat box/note.txt", "cat box/note.txt", d)
os.chmod(box, 0o300)
attempt("ls box (mode -wx-----: no r)", "ls box", d)
attempt("cat box/note.txt, knowing the name", "cat box/note.txt", d)
os.chmod(box, 0o500)
attempt("rm box/note.txt (mode r-x-----: no w)", "rm -f box/note.txt", d)
os.chmod(box, 0o700)
attempt("rm fixed.txt (r--r--r--, folder writable)", "rm -f fixed.txt", d)
os.chmod(d / "secret.txt", 0o600)

```

OUTPUT

- Nine bits: read, write, execute, for the owner, the group, and everyone else

644	-rw-r--r--	owner	rw-	group	r--	others	r--
755	-rwxr-xr-x	owner	rwx	group	r-x	others	r-x
600	-rw-----	owner	rw-	group	---	others	---
750	-rwxr-x---	owner	rwx	group	r-x	others	---
400	-r-----	owner	r--	group	---	others	---
- chmod sets bits by number or by name; umask removes bits from everything you create


```

chmod u+x  plan.txt  now -rwxr--r--
chmod go-r plan.txt  now -rwx-----
chmod a+r  plan.txt  now -rwxr--r--
chmod 640  plan.txt  now -rw-r-----
created with umask 022: a.txt  -rw-r--r--
created with umask 022: a-dir  drwxr-xr-x
created with umask 077: b.txt  -rw-----
created with umask 077: b-dir  drwx-----

```
- On a file, r lets you read it, w lets you change it, x lets you run it


```

cat secret.txt (mode --w-----)      refused (status 1)
append to fixed.txt (mode r--r--r--)  refused (status 1)
./hello.sh (mode rw-r--r--)           refused (status 126)
bash hello.sh (bash only reads it)    allowed, printed 'hello from the script'

```
- On a folder, r lets you list names, w lets you add and remove names, x lets you go in


```

echo box/* (mode rw-----: no x)     allowed, printed 'box/note.txt'
cat box/note.txt                      refused (status 1)
ls box (mode -wx-----: no r)        refused (status 1)
cat box/note.txt, knowing the name     allowed, printed 'inside'
rm box/note.txt (mode r-x-----: no w) refused (status 1)
rm fixed.txt (r--r--r--, folder writable) allowed

```

What the example shows

On a file, the bits mean what they say. With mode `--w-----`, the owner couldn't read `secret.txt`; with `r--r--r--`, appending to `fixed.txt` was refused. `./hello.sh` failed with status 126, chapter 2's "found but not executable", while `bash hello.sh` ran it, because then `bash` is the program and only needs to read the file. Execute permission is a statement of intent, not a security barrier. Anyone who can read a script can run it.

On a folder, they mean something else. `r` lets you read the folder's list of names, so `echo box/*` worked on `rw-----`, but without `x` nothing inside could be reached: `cat box/note.txt` was refused. On a Mac, even `ls box` complains in that state, because it tries to look at each entry, which needs `x`. With `-wx`, the opposite: the names can't be listed, but a file whose name you already know can be opened. `w` on a folder means changing its list of names: without it, `rm box/note.txt` was refused even though the file itself was fine. And the last line is the surprise: a read-only file was deleted, because its folder was writable. Deleting a file removes a name from a folder, as chapter 5 explained, so it's the folder's permission that counts. Plain `rm` would have asked first, at a terminal; `-f` skipped the question, not the rule.

The administrator, and three special bits

The administrator account, `root`, user number 0, bypasses almost all of these checks, which is why the example refuses to run as root: its results would be meaningless. `sudo command` runs one command as root, after asking for your password, and records that you did. Use it for system changes, installing a package or editing a file in `/etc`, and never to make an error in your own folders go away. `sudo pip install` or `sudo npm install` leaves root-owned files in your home folder that later commands, run as you, can't change, and the fix is to correct the ownership, not to use `sudo` again.

Three more bits exist. **Setuid** on a program makes it run as its owner: `passwd` is owned by root and setuid, so that you can change your own password in a file only root may write. **Setgid** on a folder, on Linux, makes new files inside it belong to the folder's group, which is how shared project folders work; macOS always does that, setgid or not. And the **sticky bit** on a folder, shown as a `t` in `drwxrwxrwt`, lets everyone create files there but each person delete only their own, which is what keeps `/tmp` civil. A Mac adds extended attributes, shown as `@` after the mode, and access control lists, shown as `+`; `xattr` and `ls -le` show them.

COMMON TRAPS

`chmod -R 777` to make an error disappear, which lets every user on the machine change everything.

`sudo` on tools that write to your home folder, leaving files you can no longer change.

Forgetting `x` on a folder, so readable files inside it can't be reached.

Thinking a read-only file is safe from deletion: its folder decides.

Private keys and tokens readable by others, which SSH and many other programs refuse to use.

EXERCISES

EXERCISE 6.1 Bits and numbers

(a) What number is `rw-r----` ? (b) What string is 755? (c) What does `chmod u=rw,go=` leave on a file that was 755? (d) With a umask of 027, what modes do new files and folders get?

SOLUTION

(a) 640: `rw-` is 4 + 2, `r--` is 4, `---` is 0. (b) `rw-r--r--`. (c) 600, `rw----` : `=` sets a class's bits exactly, and `go=` with nothing after it removes all of them. (d) 640 for files and 750 for folders: 666 without the bits in 027 is 640, and 777 without them is 750. Running `umask 027; touch f; mkdir d` confirms it, and gives the group read access while shutting everyone else out, a common setting on shared servers. Note the umask can only remove bits: no umask makes new files executable, because programs don't ask for that bit when they create ordinary files.

EXERCISE 6.2 The key that's too open

Running `ssh server` prints `WARNING: UNPROTECTED PRIVATE KEY FILE!` and Permissions 0644 for `'/home/you/.ssh/id_ed25519'` are too open, and then refuses to use the key. Why does SSH care, and what's the fix?

SOLUTION

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/id_ed25519      # private key: only you may read it
chmod 644 ~/.ssh/id_ed25519.pub  # public key: anyone may read it, that's its
                                purpose
```

With 644, every user on the machine can read your private key, and anyone who has it can log in as you wherever the key is accepted. SSH refuses on principle rather than assume nobody else has an account. The fix is to remove every permission for the group and others. The folder should be 700 too, so others can't even see which keys you have.

The same reasoning applies to any secret in a file, such as an API token, a `.env` file or a password store. Make them 600, and keep them out of repositories, as chapter 8 explains.

EXERCISE 6.3 A folder for a team

On a Linux server, everyone in the group `web` should be able to create and edit files in `/srv/site`, including each other's, and nobody else should be able to change anything. Set it up.

SOLUTION

```
sudo chgrp -R web /srv/site
sudo chmod -R g+rwX,o-w /srv/site
sudo find /srv/site -type d -exec chmod g+s {} +      # new files inherit the group
"web"
# and in each member's shell startup file:
umask 002                                           # new files are group-writable
```

The group gets read and write on everything, and execute on folders and on files that were already executable; that's what the capital `X` does, as running it confirms. Others lose write. Setgid on every folder makes new files belong to `web` rather than to their creator's own group, and the umask of 002 makes them group-writable when they're created, so a colleague can edit them without anyone running `chmod`. Without those two, the setup works for a day and then breaks file by file. For anything more detailed, such as one person who may read but not write, access control lists, through `setfacl`, are the next tool.

CHECKLIST

- ☐ Can you read a mode string, and convert between it and the octal number?
- ☐ Can you say what `r`, `w` and `x` allow on a file and on a folder?
- ☐ Can you change bits with `chmod`, both ways, and explain the umask?
- ☐ Can you explain who may delete a file, and why?
- ☐ Can you say when `sudo` is the right tool and when it makes things worse?

GO DEEPER

BOOK Learning the Shell: Permissions William Shotts

DOCS GNU Coreutils: File permissions GNU

Processes, the environment and PATH

3 exercises

*Every command runs as a **process**, started by the shell, and each process gets its own copy of the **environment**, a list of `NAME=value` strings inherited from its parent. Copies flow down to children and never back up. That's why `cd` and `export` must be part of the shell itself, and why a script can't change the folder you're in. `PATH` is one of those variables: a list of folders searched in order. It explains every "command not found", and every time the wrong version of a program runs.*

Processes

A process is a running program: its code, its memory, its open files, its current folder, and a number, the process ID, that names it. Every process except the first has a parent, the process that started it, and `ps`, `top` or the friendlier `htop` show them all. When you type a command, the shell makes a copy of itself, and the copy replaces its own program with the one you asked for, keeping its open files, its folder and its environment. That is how a child inherits everything from chapter 2's redirections. When the child exits, the shell receives its exit status, and shows the prompt again.

A command ending in `&` runs in the background while you carry on; `jobs` lists them, `fg` brings one back, and Ctrl-Z pauses the current one so that `bg` can continue it in the background. Processes are stopped with **signals**. Ctrl-C sends `SIGINT`, `kill 1234` sends `SIGTERM`, a polite request to exit that a program can catch to clean up, as chapter 4's trap did. `kill -9 1234` sends `SIGKILL`, which can't be caught and leaves no chance to clean up, so it's the last resort, not the first.

Variables and the environment

`x=1` makes a **shell variable**, which only this shell can see. `export x`, or `export x=1` in one go, marks it to be copied into the environment of every program the shell starts from then on. `env` or `printenv` lists the current environment, and `unset x` removes a variable. Putting assignments before a command, `LANG=C sort names.txt`, sets them in that one command's environment and nowhere else. By convention, environment variables have upper-case names. Some are shared by everything: `HOME`, `PATH`, `EDITOR`, `LANG` and `TZ`. Others belong to one program, like the `GIT_` variables the Git examples use to fix names and dates.

Down, never up

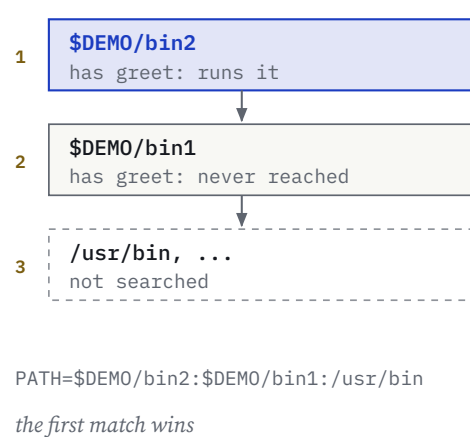
A child gets a *copy* of its parent's environment, at the moment it starts. Whatever it changes afterwards, in its variables or its current folder, changes only its copy, and disappears when it exits. Nothing in the system lets a child reach back into its parent. So `cd` can't be an ordinary program. A `cd` program would change its own folder and then exit, leaving yours exactly where it was. That's why `cd`, `export`, `source` and a few others are **builtins**, commands the shell runs itself. `type name` says what a name is: a builtin, an alias, a shell function, or a program and where it is.

The same rule decides what a script can do to your shell. Running `bash setup.sh`, or `./setup.sh`, starts a child, so its `cd` and `export` vanish with it. **Sourcing** it, `source setup.sh` or `. setup.sh`, runs its lines in your current shell, as if you'd typed them, so their effects stay. Startup files, and virtual environments' `activate` scripts, are sourced for exactly this reason.

COPIES FLOW DOWN



LOOKING UP greet



Environments are copied down at the moment a process starts, and never travel back. PATH is searched left to right, and the first match wins.

PATH

When you type a command name without a `/` in it, the shell looks in each folder listed in `PATH`, separated by colons, in order, and runs the first executable file with that name. A name containing a slash, like `./tidy` or `/usr/bin/python3`, is used as given, with no search. `type -a name` lists every match in search order, and `command -v name` prints the one that would run, which makes it the reliable check in scripts. `which` is a separate program on many systems and can disagree, because it doesn't know about your aliases and functions. To add a folder, put it at the front, where it wins, in the startup file chapter 8 sets up: `export PATH="$HOME/.local/bin:$PATH"`. Never add `.`, the current folder: any folder you `cd` into could then supply a fake `ls`.

Bash also remembers where it last found each command, so as not to search every time. If you install a program into a folder that comes earlier in `PATH` than the version you've been running, bash keeps running the old one until you type `hash -r` (in zsh, `rehash`) or open

a new shell. Changing `PATH` itself clears that memory. Version managers for Python, Node and the rest, and Python's virtual environments, all work the same way: they put their own folder at the front of `PATH`.

The example builds two folders, `bin1` and `bin2`, each with its own program called `greet`. It runs them with `PATH` in each order, passes variables to child shells, runs and sources a script, and finally starts bash in four ways, with startup files that announce themselves.

EXAMPLE

env.py · `PATH` order and type, variables exported or not, changes that don't flow back, a script run and sourced, and which startup files bash reads

```
import os, pathlib, shutil, subprocess, tempfile

def sh(cmd, cwd, env, args=()):
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", *args, "-c", cmd],
        cwd=cwd, env=env,
        capture_output=True, text=True, stdin=subprocess.DEVNULL)
    return r.stdout.replace(str(cwd), "$DEMO").rstrip("\n"), r.returncode

def show(cmd, cwd, env, args=()):
    out, status = sh(cmd, cwd, env, args)
    first, *rest = cmd.splitlines()
    print(f"    ${first}" + "".join(f"\n    > {line}" for line in rest))
    for line in out.splitlines(): print(f"        {line}")
    if status: print(f"        (exit status {status}" + (": command not found" if status == 127
        else " "))

with tempfile.TemporaryDirectory() as tmp:
    demo = pathlib.Path(tmp).resolve()
    for folder, text in (("bin1", "greet from bin1"), ("bin2", "greet from bin2")):
        (demo / folder).mkdir(); tool = demo / folder / "greet"
        tool.write_text(f"#!/bin/sh\necho '{text}'\n"); tool.chmod(0o755)
    (demo / "sub").mkdir()
    (demo / "setup.sh").write_text('export GREETING=hello\ncd sub\n')
    base = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C"}
    print("($DEMO stands for the example's temporary folder)")

    print("1. PATH is a list of folders; the shell runs the first matching program it finds")
    show('PATH="$DEMO/bin1:$DEMO/bin2:$PATH"; greet', demo, dict(base, DEMO=str(demo)))
    show('PATH="$DEMO/bin2:$DEMO/bin1:$PATH"; greet; type -a greet', demo, dict(base,
        DEMO=str(demo)))
    show("greet", demo, base)
    show("type cd export greet", demo, dict(base, PATH=f"{demo}/bin1:{base['PATH']}"))

    print("2. A child process gets a copy of the exported variables; changes never flow back
        up")
    show('x=1; bash -c \'echo "child sees x=[x]"\'', demo, base)
    show('export x=1; bash -c \'echo "child sees x=[x]"\'', demo, base)
    show('x=2 bash -c \'echo "child sees x=[x]"; echo "the shell still has x=[x]"\'', demo,
        base)
    show("export x=1; bash -c 'x=99; cd /'; echo \"x=$x, still in $PWD\"", demo, base)

    print("3. Running a script starts a child; sourcing it runs its lines in this shell")
    show('bash setup.sh; echo "GREETING=[GREETING], in $PWD"', demo, base)
    show('. ./setup.sh; echo "GREETING=[GREETING], in $PWD"', demo, base)

    print("4. Which startup files bash reads depends on how it was started")
    home = demo / "home"; home.mkdir()
    for name in (".bash_profile", ".bashrc", ".profile"):
        (home / name).write_text(f'echo "    (bash read ~/{name})"\n')
```

```

env = dict(base, HOME=str(home))
for label, args in (("a script: neither login nor interactive", []),
                    ("a login shell (bash -l)", ["-l"]), ("an interactive shell (bash -i)",
                    ["-i"]),
                    ("an interactive login shell, like a new terminal on macOS", ["-l", "-i"])):
    out = subprocess.run([shutil.which("bash"), *args, "-c", "true"], env=env, cwd=demo,
                          capture_output=True, text=True, stdin=subprocess.DEVNULL).stdout
    print(f"    {label}:\n{out.rstrip()} or '    (no startup files)'}")

```

OUTPUT

(\$DEMO stands for the example's temporary folder)

1. PATH is a list of folders; the shell runs the first matching program it finds

```

$ PATH="$DEMO/bin1:$DEMO/bin2:$PATH"; greet
greet from bin1
$ PATH="$DEMO/bin2:$DEMO/bin1:$PATH"; greet; type -a greet
greet from bin2
greet is $DEMO/bin2/greet
greet is $DEMO/bin1/greet
$ greet
(exit status 127: command not found)
$ type cd export greet
cd is a shell builtin
export is a shell builtin
greet is $DEMO/bin1/greet

```

2. A child process gets a copy of the exported variables; changes never flow back up

```

$ x=1; bash -c 'echo "child sees x=[x]"'
child sees x=[]
$ export x=1; bash -c 'echo "child sees x=[x]"'
child sees x=[1]
$ x=2 bash -c 'echo "child sees x=[x]"; echo "the shell still has x=[x]"'
child sees x=[2]
the shell still has x=[]
$ export x=1; bash -c 'x=99; cd /'; echo "x=$x, still in $PWD"
x=1, still in $DEMO

```

3. Running a script starts a child; sourcing it runs its lines in this shell

```

$ bash setup.sh; echo "GREETING=[$GREETING], in $PWD"
GREETING=[], in $DEMO
$ . ./setup.sh; echo "GREETING=[$GREETING], in $PWD"
GREETING=[hello], in $DEMO/sub

```

4. Which startup files bash reads depends on how it was started

```

a script: neither login nor interactive:
(no startup files)
a login shell (bash -l):
(bash read ~/.bash_profile)
an interactive shell (bash -i):
(bash read ~/.bashrc)
an interactive login shell, like a new terminal on macOS:
(bash read ~/.bash_profile)

```

What the example shows

The first match wins, and builtins aren't on PATH at all. With `bin1` first, `greet` ran `bin1`'s program; reversed, `bin2`'s, and `type -a` listed both in order. With neither folder on `PATH`, status 127. `type` called `cd` and `export` builtins, which is why no `PATH` change can ever break them.

Only exported variables reach a child, and nothing comes back. The unexported `x` was invisible to the child bash; exported, it arrived. `x=2 bash -c ...` gave the child a value the parent never had. The child's `x=99` and `cd /` changed its copy only. Running `setup.sh` left `GREETING` empty and the folder unchanged, and sourcing it did both.

Startup files depend on how the shell started. A script reads no startup files. A login shell, which is what you get when you log in, read `~/.bash_profile`; an interactive shell, one you type into, read `~/.bashrc`. And an interactive login shell read only `~/.bash_profile`, which is the surprise. Terminal on a Mac starts a login shell in every window, so settings put only in `.bashrc` seem to be ignored there, while most Linux terminals start non-login shells, which read only `.bashrc`. The usual fix is a line in `~/.bash_profile` that sources `~/.bashrc`. Zsh, the default on Macs since 2019, has its own set: `~/.zshenv` always, `~/.zprofile` for login shells, and `~/.zshrc` for interactive ones. Put environment settings such as `PATH` in the login file, and interactive ones, aliases and the prompt, in the rc file.

COMMON TRAPS

Expecting a script to change your shell's folder or variables: source it, or make it a function.

Setting a variable without `export` and wondering why a program can't see it.

Adding to `PATH` in a file that every nested shell reads, so it grows each time.

Trusting `which` over `type -a`, which knows about aliases, functions and order.

A stale command after installing a new version: `hash -r` or a new shell.

EXERCISES

EXERCISE 7.1 The wrong Python

You installed Python 3.13, but `python3 --version` still says 3.9. How do you find out why, and fix it?

SOLUTION

```
type -a python3          # every python3 on PATH, in the order they're searched
echo "$PATH" | tr ':' '\n' # the folders, one per line
hash -r                  # forget remembered locations, then try again
# if the new folder is after the old one, put it first, in ~/.zprofile or
# ~/.bash_profile:
export PATH="/opt/homebrew/bin:$PATH"
```

Either the new Python's folder comes later in `PATH` than the old one's, so the old one wins, or bash remembered the old location. `type -a` tells you which. In a test with an old `python3` already found once, and a new one then installed into a folder earlier on `PATH`, the shell kept running the old one, and `type` said `python3` is hashed with the old path, until `hash -r`. If the order is the problem, change `PATH` in your login startup file and open a new terminal. Don't fix it by deleting or renaming the system's Python, which other programs rely on. For projects, a virtual environment, from chapter 16 of *The Engineer's Primer*, puts the right Python first automatically.

EXERCISE 7.2 The script that wouldn't move

A script called `proj` contains `cd ~/code/"$1"`. Running `proj shop` prints nothing, and leaves you where you were. Why, and what should it be instead?

SOLUTION

```
# in ~/.zshrc or ~/.bashrc
proj() {
  cd ~/code/"$1" || return
}
```

The script ran as a child process. It changed its own folder, which it's allowed to do, then exited, and its folder went with it. Your shell's folder was never touched, because a child can't change its parent. A **function** defined in your startup file runs inside your shell, so its `cd` is your `cd`. `|| return` makes it stop with a failure status if the folder doesn't exist. An alias would work for a fixed folder, but can't take an argument in the middle. The same reasoning explains why tools that change your environment, from `nvm` to Python's `activate`, are shell functions or sourced scripts, never ordinary programs.

EXERCISE 7.3 The PATH that grew

Your `~/.zshrc` has `export PATH="$HOME/bin:$PATH"`. After a while you notice `$HOME/bin` appears in `PATH` several times. Why, how can you check, and what are two fixes?

SOLUTION

```
echo "$PATH" | tr ':' '\n' | sort | uniq -d      # print any folder that appears twice

# fix 1: in zsh, make the path array keep only the first copy of each entry
```

```
typeset -U path
```

```
# fix 2, for bash or any shell: only add the folder if it's missing  
add_path() { case ":$PATH:" in *:$1:* ) ;; *) PATH="$1:$PATH" ;; esac; }  
add_path "$HOME/bin"
```

`.zshrc` runs for every interactive shell, and each one inherits the `PATH` its parent already extended. So a shell started inside another one, in a terminal multiplexer, in an editor's terminal, or by typing `zsh`, adds the folder again. The duplicates are mostly harmless, but they make `PATH` hard to read and slow every failed lookup. The check prints each folder that appears more than once. In `zsh`, `path` is an array linked to `PATH`, and `typeset -U` makes it unique. The `case` pattern works in any shell: called three times, it added the folder once. Better still, set `PATH` in the login file, `~/.zprofile`, which runs once per login rather than once per shell.

CHECKLIST

- ☐ Can you explain what a child process inherits, and why nothing flows back?
- ☐ Can you say why `cd` is a builtin, and when to source a script instead of running it?
- ☐ Can you predict which program a name runs, and check with `type -a`?
- ☐ Can you add a folder to `PATH` safely, in the right startup file?
- ☐ Can you stop a process politely with a signal, and say when `kill -9` is justified?

GO DEEPER

COURSE The Missing Semester: Command-line Environment MIT

DOCS Bash Reference Manual: Bash Startup Files GNU

DOCS An Introduction to the Z Shell: Startup Files The Z Shell project

Dotfiles

3 exercises

Your tools read their settings from small text files in your home folder, named with a leading dot so they stay out of the way. Kept in a Git repository, with a script that links each one into place, they turn setting up a new laptop from a lost day into ten minutes, and they're the first half of this stage's deliverable. The script has to be safe to run twice, must never destroy what's already there, and the repository, which you'll make public, must never contain a secret.

What dotfiles are

Each program decides where it looks for its settings, and most look in your home folder: `~/.zshrc` and `~/.bashrc` for the shells, `~/.gitconfig` for Git, `~/.vimrc` for Vim, `~/.ssh/config` for SSH. Newer programs follow the XDG convention and keep their files under `~/.config`, so Git also reads `~/.config/git/config` and `~/.config/git/ignore`, and Neovim reads `~/.config/nvim`. A program's manual page lists its files, usually in a section called FILES, which is the place to look when you don't know where a setting lives.

What to put in them

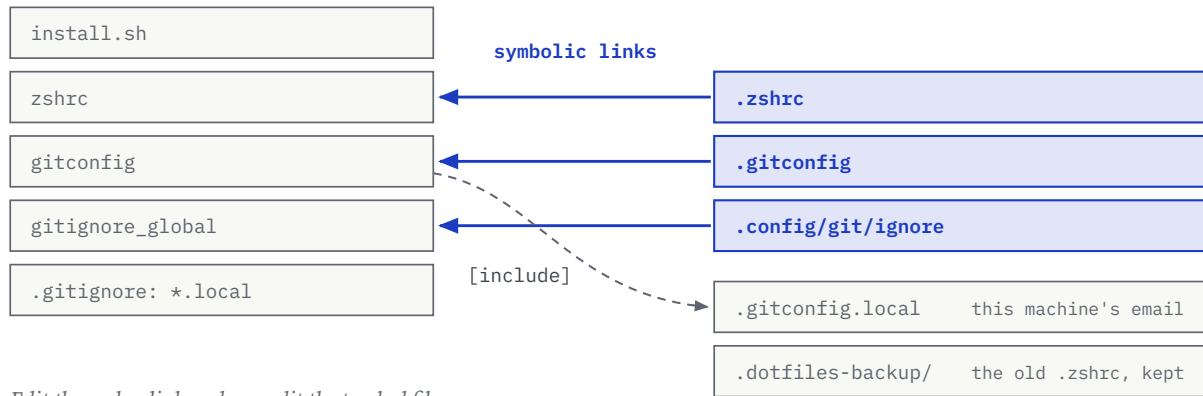
Start small and grow the files by noticing. Each time you type something long twice, or fix the same annoyance again, add a line, with a comment saying why. A useful first set covers three files. In the shell's: a `PATH` that includes `~/.local/bin`, from chapter 7; `EDITOR`; a long history that's shared between terminals; a few aliases, such as `ll` for `ls -lah`; chapter 7's `proj` function; and a prompt that shows the current Git branch. In Git's: your name, `init.defaultBranch`, `pull.ff only`, and an alias like `lg = log --graph --oneline --all`, which you'll type a thousand times in Part III. In your editor's: whatever you've changed from its defaults. Reading other people's dotfiles on GitHub is a good source of ideas, and copying a stranger's five-hundred-line configuration whole is how you end up with a shell you can't explain.

A repository, and links into it

Keep the files in one folder, such as `~/dotfiles`, under Git, named without their dots so they're visible: `zshrc`, `gitconfig`. Then make the real locations **symbolic links** into the repository, as chapter 5 described. Every program reads the repository's file through its link, so editing `~/.zshrc` edits the tracked file, and `git diff` shows what you changed. Copying the files into place instead works once, and then the copies and the repository drift apart. An install script makes the links. Tools such as GNU Stow, chezmoi and yadm do the same job with more features. Write your own first, because it's twenty lines and you'll understand every one.

~/dotfiles: PUBLIC

~: YOUR HOME FOLDER



The repository holds what you share; the home folder holds links to it, plus anything private to one machine, which the shared files include but never contain.

The example builds a small repository with an install script and three settings files, and a home folder that already has an old `.zshrc`. It installs twice, then checks the result the only way that matters: by asking Git what it now reads.

EXAMPLE

`dotfiles.py` · a dotfiles repository whose install script links each file into place, backing up what was there, run twice, then checked through git config, a global ignore file and the repository's own `.gitignore`

```
import os, pathlib, shutil, subprocess, tempfile

INSTALL = r'''#!/usr/bin/env bash
# install.sh: link each file in this repository to where its program looks for it. Safe to
# rerun.
set -euo pipefail
repo=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
backup="$HOME/.dotfiles-backup"

link() { # link() FILE-IN-REPO PATH-UNDER-HOME
    local src="$repo/$1" dst="$HOME/$2"
    if [ -L "$dst" ] && [ "$(readlink "$dst")" = "$src" ]; then
        echo "ok          ~/$2"; return
    fi
    mkdir -p "$(dirname "$dst")"
    if [ -e "$dst" ] || [ -L "$dst" ]; then      # something else is there: keep it, never delete
        it
        mkdir -p "$backup"
        mv "$dst" "$backup/${2//\\/_}"
        echo "backed up ~/$2 to ~/.dotfiles-backup/${2//\\/_}"
    fi
    ln -s "$src" "$dst"
    echo "linked      ~/$2 -> $1"
}

link zshrc .zshrc
link gitconfig .gitconfig
link gitignore_global .config/git/ignore
```

```
[ -e "$HOME/.gitconfig.local" ] || echo "note: put this machine's email in ~/.gitconfig.local"
'''
FILES = {
    "install.sh": INSTALL,
    "zshrc": 'export EDITOR=vim\nalias gs="git status --short"\n',
    "gitconfig": "[user]\n\tname = Ada Lovelace\n\tinclude]\n\t\tpath = ~/.gitconfig.local\n"
        "[init]\n\t\tdefaultBranch = main\n",
    "gitignore_global": ".DS_Store\n*.swp\n",
    ".gitignore": "*.local\n",
}

def sh(cmd, cwd, env):
    r = subprocess.run([shutil.which("bash"), "--norc", "--noprofile", "-c", cmd], cwd=cwd,
        env=env,
        capture_output=True, text=True)

    print(f"$ {cmd}")
    for line in (r.stdout + r.stderr).replace(env["HOME"], "~").splitlines(): print(f"
        {line}")

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp).resolve()
    repo, home = root / "dotfiles", root / "home"; repo.mkdir(); home.mkdir()
    for name, text in FILES.items(): (repo / name).write_text(text)
    (repo / "install.sh").chmod(0o755)
    (home / ".zshrc").write_text("# the old settings this laptop came with\n")
    env = {"PATH": os.environ["PATH"], "HOME": str(home), "LC_ALL": "C", "GIT_CONFIG_NOSYSTEM":
        "1"}

    print("1. The first run links everything, keeping a backup of what was already there")
    sh("dotfiles/install.sh", root, env)
    print("2. A second run changes nothing: running it again is always safe")
    sh("dotfiles/install.sh", root, env)
    for name in (".zshrc", ".gitconfig", ".config/git/ignore"):
        p = home / name
        print(f"    ~/{name:18} -> {pathlib.Path(os.readlink(p)).relative_to(root)}")
    print(f"    the old ~/.zshrc is kept: {(home / '.dotfiles-
        backup/.zshrc').read_text().strip()!r}")

    print("3. Git reads the linked config, and the include adds settings that never go in the
        repository")
    (home / ".gitconfig.local").write_text("[user]\n\temail = ada@example.org\n")
    sh("git config --show-origin --get user.name; git config --show-origin --get user.email",
        root, env)

    print("4. The global ignore file applies in every repository on this machine")
    sh("git init -q project && cd project && touch .DS_Store notes.md && git status --short",
        root, env)

    print("5. The repository's own .gitignore keeps per-machine files out of it")
    sh("cd dotfiles && git init -q && touch secrets.local && git add -A && git status --short",
        root, env)
```

OUTPUT

```
1. The first run links everything, keeping a backup of what was already there
$ dotfiles/install.sh
backed up    ~/.zshrc to ~/.dotfiles-backup/.zshrc
linked      ~/.zshrc -> zshrc
linked      ~/.gitconfig -> gitconfig
linked      ~/.config/git/ignore -> gitignore_global
note: put this machine's email in ~/.gitconfig.local
2. A second run changes nothing: running it again is always safe
$ dotfiles/install.sh
ok          ~/.zshrc
ok          ~/.gitconfig
ok          ~/.config/git/ignore
```

```

note: put this machine's email in ~/.gitconfig.local
~/zshrc -> dotfiles/zshrc
~/gitconfig -> dotfiles/gitconfig
~/config/git/ignore -> dotfiles/gitignore_global
the old ~/.zshrc is kept: '# the old settings this laptop came with'
3. Git reads the linked config, and the include adds settings that never go in the repository
$ git config --show-origin --get user.name; git config --show-origin --get user.email
file:~/gitconfig Ada Lovelace
file:~/gitconfig.local ada@example.org
4. The global ignore file applies in every repository on this machine
$ git init -q project && cd project && touch .DS_Store notes.md && git status --short
?? notes.md
5. The repository's own .gitignore keeps per-machine files out of it
$ cd dotfiles && git init -q && touch secrets.local && git add -A && git status --short
A .gitignore
A gitconfig
A gitignore_global
A install.sh
A zshrc

```

What the example shows

The script is safe to run any number of times. On the first run it found a real file at `~/zshrc`, moved it into `~/dotfiles-backup` and linked the repository's version in its place. The second run found every link already pointing at the right file and changed nothing. That's the check `[ "$(readlink "$dst")" = "$src" ]`. `${2//\\/_}` replaces each slash in a path with an underscore, so a backup of `.config/git/ignore` gets a flat, unambiguous name. And the first line, built from `BASH_SOURCE` and `pwd -P`, finds the repository's absolute location wherever the script is run from, which exercise 8.1 explains.

Shared settings and private ones stay apart. `git config --show-origin` showed the name coming from the linked `~/gitconfig` and the email from `~/gitconfig.local`, which the shared file pulls in with an `[include]` but which isn't in the repository. That's the pattern for anything that differs between machines or shouldn't be public: a work email, a signing key, a company's proxy. The shell's version is a last line in `zshrc`, `[ -f ~/.zshrc.local ] && source ~/.zshrc.local`. The global ignore file kept `.DS_Store` out of a brand-new repository without any setup there. In the dotfiles repository itself, `*.local` in `.gitignore` kept a stray `secrets.local` out of the commit.

Secrets, and the README

A dotfiles repository is meant to be public, so treat everything in it as published. Never commit API tokens, passwords, private SSH keys, `~/netrc` or a shell history; keep them in local files, as above, or in your system's keychain or a password manager. Before your first push, and from time to time after, search for anything that looks like a secret, with `git grep -iE 'token|secret|passw|api_?key'` or a scanner such as gitleaks. Bots search public repositories for keys within minutes of a push, so a leaked key has to be treated as stolen at once, which is what exercise 8.2 is about.

Then write the **README** for a stranger who has just cloned it. Say what's in the repository and what each file configures. Give the one command that installs it, say exactly what that command changes and how to undo it, and list the systems you've tried it on. Your future self, setting up a new machine in a hurry, is the stranger.

COMMON TRAPS

A token, key or private detail committed to a public repository.

Copying someone's entire configuration instead of growing your own a line at a time.

An install script that overwrites existing files instead of backing them up.

Links that store a relative path that only works from one folder.

Settings for one system only, such as Homebrew's paths, applied unconditionally on another; check with `uname`.

EXERCISES

EXERCISE 8.1 Run from anywhere

The install script finds its repository with `repo=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)`. What goes wrong with the simpler `repo=$(dirname "$0")`, run as `./install.sh` from inside the repository?

SOLUTION

`dirname "$0"` is the folder part of however the script was invoked: run as `./install.sh`, that's just `.`. The script then stores `./zshrc` in the link at `~/zshrc`, and as chapter 5 explained, a relative link is read from the link's own folder, so it points at `~/zshrc`, which doesn't exist. Run that way in a test, the simple version printed `linked ~/zshrc -> zshrc` for every file, and every link it made was broken. The correct line moves into the script's folder, so the path doesn't depend on how it was invoked. It then asks for the absolute physical path, with symbolic links resolved: that's `pwd -P`. `BASH_SOURCE[0]` is the script's own path even when it's sourced, which `$0` isn't. The lesson is to check that a script's effects work, here with `[ -e ~/zshrc ]`, not just that it ran without complaint.

EXERCISE 8.2 The token in the repository

You notice that last week you pushed a `zshrc` containing `export GITHUB_TOKEN=ghp_...` to your public dotfiles repository. What do you do, in what order?

SOLUTION

First, revoke the token, on GitHub's settings page, and make a new one. Assume it was copied within minutes of the push: bots watch public pushes for exactly this, and nothing you do to the repository can recall a copy. Second, check what the token could reach for anything done with it. Only then clean up. Move the token to a local file such as `~/.zshrc.local` that the repository ignores. Then remove it from history with `git filter-repo --replace-text`, which rewrites every commit that contained it, and force-push. Rewriting history is tidy but not protection: forks, clones and caches keep the old commits, which is why revoking comes first. GitHub's secret scanning and push protection catch many token formats, including GitHub's own, before or just after they're pushed. They're worth turning on, and no reason to rely on luck.

EXERCISE 8.3 An uninstall

Write `uninstall.sh`, which removes only the links that point into the repository, and puts back anything `install.sh` backed up.

SOLUTION

```
#!/usr/bin/env bash
# uninstall.sh: remove the links install.sh made, and put back what it backed up.
set -euo pipefail
repo=$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd -P)
backup="$HOME/.dotfiles-backup"
unlink_one() { # unlink_one FILE-IN-REPO PATH-UNDER-HOME
    local src="$repo/$1" dst="$HOME/$2" saved="$backup/${2//\\/_}"
    if [ -L "$dst" ] && [ "$(readlink "$dst")" = "$src" ]; then
        rm -- "$dst"; echo "removed    ~/$2"
    else
        echo "left alone ~/$2 (not our link)"; return
    fi
    if [ -e "$saved" ] || [ -L "$saved" ]; then mv -- "$saved" "$dst"; echo "restored
~/$2"; fi
}
unlink_one zshrc .zshrc
unlink_one gitconfig .gitconfig
unlink_one gitignore_global .config/git/ignore
```

It uses the same test as `install.sh`, so it only removes a link that is exactly the one the installer made. Anything else at that path, including a file someone put there since, is left alone and reported. Tested after an install, with `~/.gitconfig` then replaced by an ordinary file, it removed the two links, restored the original `~/.zshrc` from the backup, and left `~/.gitconfig` untouched. Sharing the list of files between the two scripts, for example in a file both read, would stop them drifting apart as the repository grows.

CHECKLIST

- ☐ Can you find where a program reads its settings, using its manual page?
- ☐ Can you keep your dotfiles in a repository and link them into place?
- ☐ Can you write an install script that backs up, never overwrites, and is safe to rerun?
- ☐ Can you keep per-machine and secret settings out of the repository, and include them anyway?
- ☐ Can you say what to do, in order, when a secret has been pushed?

GO DEEPER

GUIDE [Your unofficial guide to dotfiles on GitHub dotfiles.github.io](https://dotfiles.github.io)

DOCS [XDG Base Directory Specification freedesktop.org](https://freedesktop.org/spec/stand/standards/xdg)

DOCS [About secret scanning GitHub Docs](#)

PART III

Git

Five chapters on the tool every software team runs on. Git has a reputation for confusing commands, and it earns it, but under the commands is a data structure small enough to draw on a napkin: objects named by their contents, and a few movable names pointing at them. The part builds that structure by hand, then uses it to explain branches, merges, rebases and conflicts. It turns history into a debugging tool, and ends with the workflow of forks and pull requests that collaboration on GitHub runs on. It's the longest part of the stage, so give it the most time.

Git's objects

3 exercises

*Git is a store of objects, each named by a hash of its own contents, plus a few names that point into it. A **blob** is a file's contents, a **tree** is a folder listing, and a **commit** is a tree together with its parent commits, an author and a message. Once you see that, Git's behaviour stops being arbitrary. A commit is a snapshot of the whole project, not a list of changes. Identical files are stored once. And changing anything in the past changes the name of every commit after it. This chapter builds all three kinds of object by hand, and gets exactly the names Git does.*

Names made from contents

A **hash function** turns any amount of data into a short, fixed-size number, such that the same input always gives the same number, and the smallest change to the input gives a completely different one. Git uses SHA-1, whose numbers are written as 40 hexadecimal digits, and names every object by the hash of a short header, its type and size, then a zero byte, then its contents. So the name of an object is a summary of what's in it: two people who commit the same file get the same blob name without ever communicating, and if a single bit of a stored object changes, its name no longer matches and Git notices. This is called **content addressing**. Objects are kept compressed in `.git/objects`, in a folder named after the first two digits, and later packed together into packfiles to save space.

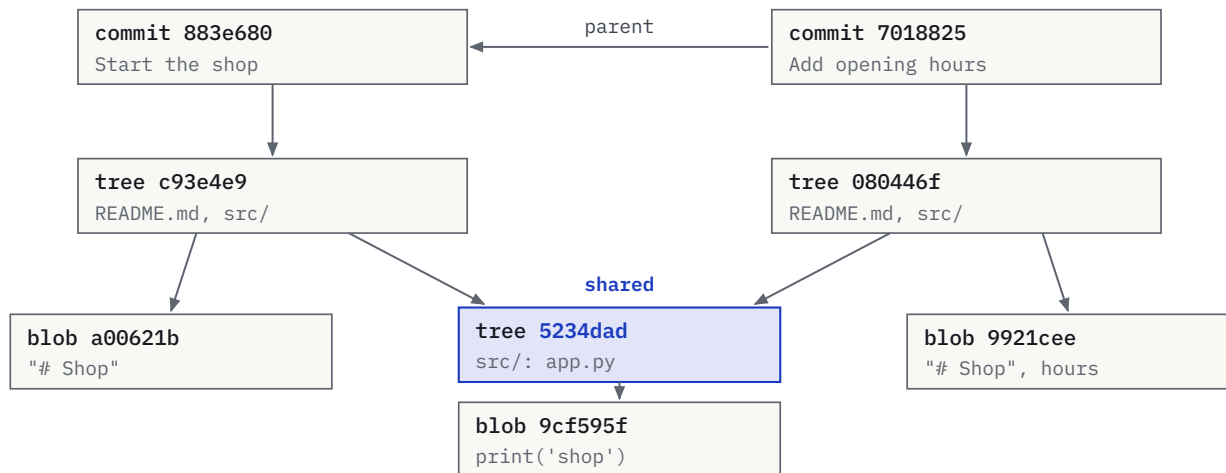
In 2017, researchers produced the first two different files with the same SHA-1 hash, at a cost of thousands of years of computing time. Git responded by using a hardened SHA-1 that detects that kind of attack, and it now also supports repositories that use SHA-256 throughout. New repositories still use SHA-1 by default.

Blobs, trees and commits

A **blob** holds a file's contents and nothing else: not its name, not its permissions. A **tree** holds a folder's listing, one entry per name. Each entry has a mode, `100644` for an ordinary file, `100755` for an executable one, `40000` for a subfolder and `120000` for a symbolic link, then the name, then the id of the blob or tree it refers to. A **commit** is a few lines of text: the id of the tree for the whole project, the ids of its parent commits, none for the first, two for a merge, then the author and committer with timestamps, a blank line, and the message. A fourth type, the annotated tag, names a commit and adds a message of its own.

Between your files and a commit sits the **index**, also called the staging area. `git add` stores the current contents of a file as a blob, and records its name and id in the index. `git commit` turns the index into trees, writes a commit pointing at the top tree, and moves the

current branch to it, which chapter 10 covers. `git status` compares three things: the last commit's tree, the index, and the files in your folder, which is why a file can be "staged" and "modified" at once.



Every commit is a full snapshot, yet storing one costs only what changed: a new blob, the trees on the path to it, and the commit. Everything else is shared by name.

The example computes object names in Python the way Git does, with `hashlib`, and checks each one against Git's own answer. Every commit gets a fixed author and time, so the 40-digit names printed here are the ones you'll get on your machine.

EXAMPLE

objects.py · a blob, a tree and a commit built by hand with SHA-1 and checked against git, then two snapshots sharing what didn't change

```

import hashlib, os, pathlib, subprocess, tempfile, zlib

# Fixed names and times, and no personal git settings, so every hash below is the same on any
# machine.
ENV = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C",
      "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull,
      "GIT_AUTHOR_NAME": "Ada Lovelace", "GIT_AUTHOR_EMAIL": "ada@example.org",
      "GIT_COMMITTER_NAME": "Ada Lovelace", "GIT_COMMITTER_EMAIL": "ada@example.org"}
clock = [1772355600] # 2026-03-01 09:00 UTC, advanced a
               minute per commit

def git(*args, cwd, data=None):
    env = dict(ENV, GIT_AUTHOR_DATE=f"@{clock[0]} +0000", GIT_COMMITTER_DATE=f"@{clock[0]}
    +0000")
    r = subprocess.run(["git", *args], cwd=cwd, env=env, input=data, capture_output=True,
                       check=True)
    return r.stdout.decode().strip()

def commit(message, cwd):
    clock[0] += 60
    git("commit", "-q", "-m", message, cwd=cwd)
    return git("rev-parse", "HEAD", cwd=cwd)

def object_id(kind, body):

```

```

    """Every git object is named by the SHA-1 of a header ("blob 6", say), a zero byte, and the
        body."""
    return hashlib.sha1(f"{{kind}} {{len(body)}}".encode() + b"\0" + body).hexdigest()

def tree_id(folder):
    """A tree lists its entries, sorted by name: mode, name, a zero byte, and the entry's id as
        20 bytes."""
    entries = []
    for p in folder.iterdir():
        if p.name == ".git": continue
        if p.is_dir(): entries.append((p.name + "/", b"40000", p.name, tree_id(p)))
        else: entries.append((p.name, b"100644", p.name, object_id("blob", p.read_bytes())))
    body = b"".join(mode + b" " + name.encode() + b"\0" + bytes.fromhex(oid)
                     for _, mode, name, oid in sorted(entries))
    return object_id("tree", body)

with tempfile.TemporaryDirectory() as tmp:
    repo = pathlib.Path(tmp)
    git("init", "-q", "-b", "main", cwd=repo)

    print("1. A blob is a file's contents, named by its hash")
    mine = object_id("blob", b"hello\n")
    print(f"    by hand:                {{mine}}")
    print(f"    git hash-object:            {{git('hash-object', '-w', '--stdin', cwd=repo,
        data=b'hello\n')}}")
    stored = repo / ".git" / "objects" / mine[:2] / mine[2:]
    print(f"    stored in .git/objects/{{mine[:2]}}/{{mine[2:8]}}..., compressed; inflated it is "
        f"{{zlib.decompress(stored.read_bytes())!r}}")

    print("2. A tree is a folder listing; a commit names a tree, its parents, an author and a
        message")
    (repo / "src").mkdir()
    (repo / "README.md").write_text("# Shop\n"); (repo / "src" /
        "app.py").write_text("print('shop')\n")
    git("add", ".", cwd=repo)
    print(f"    tree by hand:                {{tree_id(repo)}}")
    print(f"    git write-tree:              {{git('write-tree', cwd=repo)}}")
    first = commit("Start the shop", repo)
    text = (f"tree {{tree_id(repo)}}\nauthor Ada Lovelace <ada@example.org> {{clock[0]}} +0000\n"
        f"committer Ada Lovelace <ada@example.org> {{clock[0]}} +0000\n\nStart the shop\n")
    print(f"    commit by hand:              {{object_id('commit', text.encode())}}")
    print(f"    git commit:                  {{first}}")
    print("    git cat-file -p HEAD:")
    for line in git("cat-file", "-p", "HEAD", cwd=repo).splitlines(): print(f"
        {{line}}".rstrip())
    print("    git cat-file -p HEAD^{{tree}}:")
    for line in git("cat-file", "-p", "HEAD^{{tree}}", cwd=repo).splitlines(): print(f"
        {{line}}")

    print("3. A new commit is a new snapshot; whatever didn't change is shared, not copied")
    (repo / "README.md").write_text("# Shop\nOpen 9 to 5.\n"); git("add", ".", cwd=repo)
    second = commit("Add opening hours", repo)
    for name, rev in (("first", first), ("second", second)):
        print(f"    {{name:6}} commit {{rev[:10]}} tree {{git('rev-parse', rev + '^{{tree}}',
            cwd=repo)[:10]}}
            f" src/ {{git('rev-parse', rev + ':src', cwd=repo)[:10]}}
            f" README.md {{git('rev-parse', rev + ':README.md', cwd=repo)[:10]}}")
    print(f"    the second commit's parent is the first: {{git('rev-parse', 'HEAD^', cwd=repo)} ==
        first}")

    print("4. Identical contents are stored once, whatever the file is called")
    count = lambda: git("count-objects", "-v", cwd=repo).splitlines()[0].split()[1]
    before = count()
    (repo / "src" / "copy.py").write_text("print('shop')\n"); git("add", ".", cwd=repo)
    print(f"    objects stored before adding src/copy.py, a copy of app.py: {{before}}; after:
        {{count}}")

```

```
print(f"    both names point to blob {git('rev-parse', ':src/copy.py', cwd=repo)[:10]}"
      f" = {git('rev-parse', ':src/app.py', cwd=repo)[:10]}")
kinds = git("cat-file", "--batch-all-objects", "--batch-check=%(objecttype)",
            cwd=repo).split()
tally = ", ".join(f"{kinds.count(k)} {k}s" for k in ("blob", "tree", "commit"))
print(f"    objects in the repository: {tally}")
```

OUTPUT

1. A blob is a file's contents, named by its hash
 by hand: `ce013625030ba8dba906f756967f9e9ca394464a`
 git hash-object: `ce013625030ba8dba906f756967f9e9ca394464a`
 stored in .git/objects/ce/013625..., compressed; inflated it is `b'blob 6\x00hello\n'`
2. A tree is a folder listing; a commit names a tree, its parents, an author and a message
 tree by hand: `c93e4e95d9da0e7e336125c34b0ed66cfa186cd0`
 git write-tree: `c93e4e95d9da0e7e336125c34b0ed66cfa186cd0`
 commit by hand: `883e680149aecf0110fda9335267363c06bca680`
 git commit: `883e680149aecf0110fda9335267363c06bca680`
 git cat-file -p HEAD:
 tree c93e4e95d9da0e7e336125c34b0ed66cfa186cd0
 author Ada Lovelace <ada@example.org> 1772355660 +0000
 committer Ada Lovelace <ada@example.org> 1772355660 +0000

 Start the shop
 git cat-file -p HEAD^{tree}:
 100644 blob a00621bb3a9b990ee018f8d04c2d3140800d6ca6 README.md
 040000 tree 5234dadb279379214f5a218b331d5db0a198db4f src
3. A new commit is a new snapshot; whatever didn't change is shared, not copied
 first commit `883e680149 tree c93e4e95d9 src/ 5234dadb27 README.md a00621bb3a`
 second commit `70188257a5 tree 080446fce0 src/ 5234dadb27 README.md 9921ceefb6`
 the second commit's parent is the first: `True`
4. Identical contents are stored once, whatever the file is called
 objects stored before adding src/copy.py, a copy of app.py: 9; after: 9
 both names point to blob `9cf595fed1 = 9cf595fed1`
 objects in the repository: 4 blobs, 3 trees, 2 commits

What the example shows

Git's names are just hashes, and you can compute them. The blob for `hello` and a newline is `ce0136...` whether you compute it in Python or ask `git hash-object`, and the file Git stored, decompressed, is exactly the header, a zero byte and the contents. The tree computed by hand, two entries sorted by name with their modes and raw ids, matched `git write-tree`. The commit was simpler still: four lines of text and a message. Note that `cat-file -p` prints the folder's mode as `040000`, padded for display, while the stored tree says `40000`. Hashing the displayed form would give the wrong answer.

Snapshots share what didn't change. After the README changed, the second commit had a new top tree and a new README blob, but its `src` folder was the very same tree, `5234dadb27`, as the first commit's, because nothing in it changed. Adding `copy.py`, an exact copy of `app.py`, didn't add an object at all: both names point to one blob. Nine objects hold two complete snapshots: four blobs, three trees and two commits.

What follows from the model

Several things that puzzle people about Git follow directly. **A commit doesn't contain a diff**; when you ask for one, Git computes it by comparing two trees. Packfiles do store objects as differences from similar objects, but that's a storage detail, invisible to everything else. **Renames aren't recorded**: a rename is a blob that disappears from one path and appears at another, and `git log --follow` and `git diff` detect it by similarity. **Empty folders can't be committed**, because a tree lists files and subfolders, and an empty one has nothing to list, which is why projects keep a placeholder file such as `.gitkeep`. And **history can't be edited, only replaced**. A commit's name depends on its parent's name, so changing an old commit gives it a new name, which changes its child, and so on to the end, as exercise 9.3 explains.

COMMON TRAPS

Thinking a commit stores changes rather than a snapshot of every file.

Expecting Git to track renames or empty folders, which it records neither of.

Committing large binary files that change often: every version is kept for ever. Git LFS is the tool for those.

Expecting `git add` to stage later edits: it records the contents at that moment.

Editing files under `.git` by hand, instead of with the commands that keep it consistent.

EXERCISES

EXERCISE 9.1 A hash by hand

Compute the blob id for a file containing `hello` with no newline at the end, without Git, then check it. Why does `echo hello | git hash-object --stdin` give a different answer?

SOLUTION

```
import hashlib
data = b"hello"
print(hashlib.sha1(b"blob %d\0" % len(data) + data).hexdigest())  #
    b6fc4c620b67d95f953a5c1c1230aaab5db5a1b0
```

The header is the type, a space, the size in bytes as decimal digits, and a zero byte; then come the contents. `printf hello | git hash-object --stdin` agrees: `b6fc4c62...`. `echo` adds a newline, so it hashes six bytes, not five, and gets the example's `ce013625...`. One invisible

character changes the whole name, which is the point of a hash. It's also why a file whose only change is a newline at the end, or Windows line endings, counts as changed to Git.

EXERCISE 9.2 What a rename costs

In a repository with one committed file, `a.txt`, you run `git mv a.txt b.txt` and commit. Which new objects does that create?

SOLUTION

Exactly two: a new tree, which lists `b.txt` instead of `a.txt`, and a new commit pointing at it. No new blob, because the contents didn't change, and a blob has no name in it. Listing every object before and after, with `git cat-file --batch-all-objects --batch-check`, shows one tree and one commit added. In a deep folder structure, a rename also creates new trees for each folder on the path from the file up to the top, since each of their listings changes. So does an edit, which is why a commit of a one-line change in a large project is still cheap.

EXERCISE 9.3 Why history can't be changed

A branch has three commits, A, B and C. You fix a typo in A's message. What happens to B and C, and why does that matter if someone else already has them?

SOLUTION

A's message is part of A's contents, so the fixed commit has a new name, A'. B's contents include its parent's name, which is now A' instead of A, so B must be rewritten as B', with a new name, and C likewise as C'. Changing one commit renames every commit after it. That chain is what makes Git's history trustworthy: a commit's name vouches for the whole history behind it, so two repositories that agree on one name agree on everything before it. It's also what makes rewriting shared history disruptive. Anyone who already has A, B and C still has them, and from Git's point of view your A', B', C' are unrelated new commits. Their next merge will join both copies together, with the same changes appearing twice. Chapter 11's rule follows: rewrite only commits nobody else has.

- ☐ Can you explain how Git names an object, and compute a blob's name yourself?
- ☐ Can you say what a blob, a tree and a commit each contain?
- ☐ Can you explain the index, and what `git add` and `git commit` each write?
- ☐ Can you explain why identical files and unchanged folders cost nothing extra?
- ☐ Can you explain why changing an old commit changes every commit after it?

GO DEEPER

BOOK Pro Git: Git Objects Scott Chacon & Ben Straub

ARTICLE Git from the Bottom Up John Wiegley

ARTICLE SHAttered: the first SHA-1 collision CWI Amsterdam & Google

Refs, branches and HEAD

3 exercises

A branch is a small file that holds one commit's name, and `HEAD` is a file that holds the name of the current branch. Committing writes a new commit whose parent is the current one, then moves the current branch forward to it. Almost every other command that seems to rearrange history only moves these names around: switching branches, detaching `HEAD`, resetting. And nothing you've committed is lost when a name moves, because the reflog remembers everywhere each name has been.

Names for commits

Forty-digit names are for machines. People use **refs**, named pointers to commits. A branch called `main` is the file `.git/refs/heads/main`, which contains one commit's name and a newline: that's all a branch is. Tags live in `refs/tags`, and chapter 13's remote-tracking branches, like `origin/main`, in `refs/remotes`. Git sometimes packs many refs into one file, `.git/packed-refs`, for speed, and newer versions can store them in a different format altogether. So read refs with `git rev-parse main` rather than by opening files, as the example does only to show what's there.

Git understands a small language for naming commits relative to others. `HEAD~1` is the current commit's parent, and `HEAD~3` its great-grandparent, following first parents. `HEAD^2` is a merge's second parent. `main@{1}` is where `main` pointed before its last move, from the reflog. Any unambiguous prefix of a commit's name works too, so the seven-digit abbreviations Git prints are enough to type.

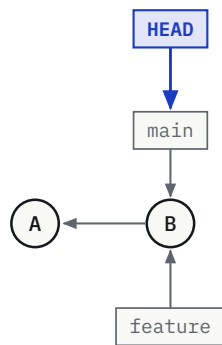
Committing moves a branch

`HEAD` normally contains `ref: refs/heads/main`, meaning "the current branch is `main`". When you commit, Git writes a commit whose parent is the commit `main` points to, then rewrites the file `refs/heads/main` to hold the new commit's name. No other branch changes. That's the whole of "being on a branch": your next commit will move that name. Making a branch writes one tiny file, so branches cost nothing, and it's normal to make one for every piece of work, however small. `git switch name` moves `HEAD` to another branch and updates your files to match its commit. The older `git checkout` does that too, along with several unrelated things, which is why newer versions of Git split it into `switch` and `restore`.

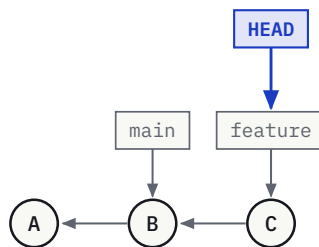
Detached HEAD

HEAD can also hold a commit's name directly, which Git calls a **detached HEAD**. You get one when you switch to a commit rather than a branch, to look at an old version, and Git puts you in one during a rebase or a bisect. You can commit there, but no branch moves, so the new commits are known only to **HEAD**. Switch to a branch, and they're no longer reachable from any name. They're not deleted, but they're easy to lose track of, which is why Git warns you when you leave. `git switch -c name` gives the commits a branch and makes them safe.

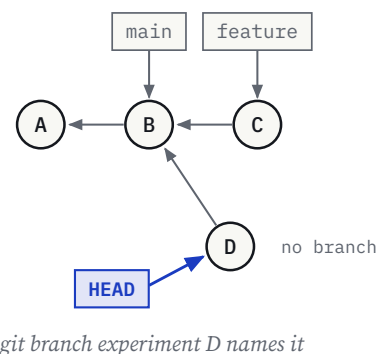
1. TWO COMMITS ON main



2. COMMIT C ON feature



3. DETACHED, COMMIT D



*Branches are labels that move when you commit; **HEAD** says which label moves. A commit with no label pointing at it is only a reflog entry away from being found again.*

The example makes the files visible. It prints the contents of **HEAD** and of the branch files as it creates a branch, commits on it, detaches **HEAD**, commits there and switches away, then finds the "lost" commit in the reflog and gives it a name. Last, it moves a branch with `reset` and moves it back.

EXAMPLE

refs.py · **HEAD** and branch files read directly while committing, branching, detaching **HEAD** and switching away, then a lost commit rescued from the reflog, and a reset undone

```

import os, pathlib, subprocess, tempfile

ENV = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C",
       "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull,
       "GIT_AUTHOR_NAME": "Ada Lovelace", "GIT_AUTHOR_EMAIL": "ada@example.org",
       "GIT_COMMITTER_NAME": "Ada Lovelace", "GIT_COMMITTER_EMAIL": "ada@example.org"}
clock = [1772355600]

def git(*args, cwd):
    env = dict(ENV, GIT_AUTHOR_DATE=f"@{clock[0]} +0000", GIT_COMMITTER_DATE=f"@{clock[0]} +0000")
    return subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True,
                          check=True).stdout.rstrip()

def commit(message, cwd):
    clock[0] += 60
    with (cwd / "log.txt").open("a") as f: f.write(message + "\n")
  
```

```

git("add", "log.txt", cwd=cwd); git("commit", "-q", "-m", message, cwd=cwd)
return git("rev-parse", "--short", "HEAD", cwd=cwd)

def files(repo, *names):
    for name in names:
        text = (repo / ".git" / name).read_text().strip()
        if not text.startswith("ref:"): text = text[:12] + "... (40 hex digits)"
        print(f"    .git/{name:18} contains: {text}")

def graph(repo):
    for line in git("log", "--all", "--graph", "--format=%h %s%d", cwd=repo).splitlines():
        print(f"        {line}".rstrip())

with tempfile.TemporaryDirectory() as tmp:
    repo = pathlib.Path(tmp); git("init", "-q", "-b", "main", cwd=repo)
    a = commit("A: first", repo); b = commit("B: second", repo)

    print("1. A branch is a file holding a commit's id; HEAD is a file naming the current
        branch")
    files(repo, "HEAD", "refs/heads/main")
    print("2. Making a branch writes one small file; nothing is copied")
    git("branch", "feature", cwd=repo); files(repo, "refs/heads/feature")
    print("3. A commit moves the current branch, and only that one")
    git("switch", "-q", "feature", cwd=repo); c = commit("C: on feature", repo)
    files(repo, "HEAD", "refs/heads/feature", "refs/heads/main"); graph(repo)

    print("4. Detached HEAD: HEAD holds a commit id instead of a branch name")
    git("switch", "-q", "--detach", b, cwd=repo); files(repo, "HEAD")
    d = commit("D: an experiment", repo)
    git("switch", "-q", "main", cwd=repo)
    print(f"    back on main, D ({d}) is on no branch, so log --all no longer shows it:");
    graph(repo)

    print("5. The reflog records every move of HEAD, so D can still be found and given a name")
    for line in git("reflog", "--format=%h %s", cwd=repo).splitlines()[:4]: print(f"
        {line}")
    git("branch", "experiment", d, cwd=repo); graph(repo)

    print("6. reset moves the current branch to another commit; the reflog can move it back")
    git("reset", "-q", "--hard", a, cwd=repo)
    print(f"    after reset --hard {a}, main is at {git('rev-parse', '--short', 'main',
        cwd=repo)}; "
        f"main's reflog: {' '.join(git('reflog', 'main', '--format=%h',
        cwd=repo).split())}")
    git("reset", "-q", "--hard", "main@{1}", cwd=repo)
    lines = len((repo / "log.txt").read_text().splitlines())
    print(f"    after reset --hard main@{1}, main is at {git('rev-parse', '--short', 'main',
        cwd=repo)} "
        f"again, and log.txt has {lines} lines")

```

OUTPUT

```

1. A branch is a file holding a commit's id; HEAD is a file naming the current branch
.git/HEAD                contains: ref: refs/heads/main
.git/refs/heads/main     contains: c7ebc43f34bf... (40 hex digits)
2. Making a branch writes one small file; nothing is copied
.git/refs/heads/feature contains: c7ebc43f34bf... (40 hex digits)
3. A commit moves the current branch, and only that one
.git/HEAD                contains: ref: refs/heads/feature
.git/refs/heads/feature contains: cb1a81f51ed7... (40 hex digits)
.git/refs/heads/main     contains: c7ebc43f34bf... (40 hex digits)
    * cb1a81f C: on feature (HEAD -> feature)
    * c7ebc43 B: second (main)
    * 87b02ea A: first
4. Detached HEAD: HEAD holds a commit id instead of a branch name
.git/HEAD                contains: c7ebc43f34bf... (40 hex digits)

```

```

back on main, D (a02cb58) is on no branch, so log --all no longer shows it:
* cb1a81f C: on feature (feature)
* c7ebc43 B: second (HEAD -> main)
* 87b02ea A: first
5. The reflog records every move of HEAD, so D can still be found and given a name
c7ebc43 checkout: moving from a02cb58cf95097efe98514c7c1793137bfdbe6c7 to main
a02cb58 commit: D: an experiment
c7ebc43 checkout: moving from feature to c7ebc43
cb1a81f commit: C: on feature
* a02cb58 D: an experiment (experiment)
| * cb1a81f C: on feature (feature)
|/
* c7ebc43 B: second (HEAD -> main)
* 87b02ea A: first
6. reset moves the current branch to another commit; the reflog can move it back
after reset --hard 87b02ea, main is at 87b02ea; main's reflog: 87b02ea, c7ebc43, 87b02ea
after reset --hard main@{1}, main is at c7ebc43 again, and log.txt has 2 lines

```

What the example shows

A branch is the name of one commit, and it moves. `refs/heads/main` held `c7ebc43f...`; making `feature` wrote a file with the same contents; committing C on `feature` changed that file and left `main`'s alone. The graph's labels, `(HEAD -> feature)` and `(main)`, are exactly those files drawn on the commits they name.

Unlabelled commits fall out of view, not out of the repository. After detaching at B and committing D, switching back to `main` left D on no branch, and `git log --all`, which starts from every ref, no longer showed it. But the reflog for `HEAD` records each move with the reason, `commit: D: an experiment` and `checkout: moving from ...`, and one `git branch` made D reachable again, joining the graph as a fork from B. Finally, `reset --hard` moved `main` back to A, which made B unreachable too. `main`'s own reflog, `87b02ea, c7ebc43, 87b02ea`, showed where it had been, and `main@{1}` put it back.

reset, and what really can be lost

`git reset target` moves the current branch, along with `HEAD`, to another commit, and its three modes say what happens to your files. `--soft` moves only the branch: the undone commit's changes stay staged, ready to be committed again, which is a way to squash several commits into one. `--mixed`, the default, also resets the index, so the changes stay in your files but unstaged. `--hard` also overwrites your files to match. That's the one dangerous command in this chapter, because changes you never committed aren't in any object, so no reflog can bring them back. Before a `--hard` reset, `git stash` saves uncommitted work as a commit you can restore. For files alone, `git restore file` discards a file's unstaged changes, and `git restore --staged file` unstages it.

Commits themselves are very hard to lose. They stay in the repository as long as a ref or a reflog entry leads to them. Reflog entries are kept for 90 days by default, or 30 for commits that are no longer reachable, and only after that can Git's garbage collection delete them. In practice, if you committed it in the last month, it's still there.

COMMON TRAPS

Committing on a detached `HEAD` and switching away without making a branch.

`git reset --hard` with uncommitted changes, the one loss no reflog can undo.

Using `reset` on commits already pushed; chapter 12's `revert` undoes them without rewriting history.

Thinking `git branch -D` deletes the commits; it deletes a name, and prints the commit it pointed to.

Reading `.git/refs` in scripts instead of asking `git rev-parse`.

EXERCISES

EXERCISE 10.1 Three kinds of reset

Your last commit added `line 2` to `f.txt`. For each of `git reset --soft HEAD~1`, `git reset HEAD~1` and `git reset --hard HEAD~1`, say where the branch points afterwards, what `git status` shows, and what `f.txt` contains.

SOLUTION

```
reset --soft HEAD~1    branch at the first commit    status: M f.txt (staged)
f.txt: line 1, line 2
reset HEAD~1           branch at the first commit    status: M f.txt (unstaged)
f.txt: line 1, line 2
reset --hard HEAD~1     branch at the first commit    status: clean
f.txt: line 1
```

All three move the branch identically; they differ in how much else they reset. In `git status --short`, the first column is the index and the second your files: an `M` in the first column means the change is staged, and in the second, that it's only in your folder. Each line was checked on a fresh repository. In all three cases the undone commit is still reachable through the reflog, so even the `--hard` reset here could be undone with `git reset --hard HEAD@{1}`. What it would destroy for good is any edit to `f.txt` that hadn't been committed.

EXERCISE 10.2 The deleted branch

You ran `git branch -D feature` on the wrong branch, which had a week of unmerged commits. Get it back.

SOLUTION

```
git branch -D feature
# Deleted branch feature (was 6f8d2ca).      <- the commit it pointed to
git branch feature 6f8d2ca                  # recreate the name; the commits were
                                           never deleted

# if that message has scrolled away, the reflog of HEAD remembers being on the branch:
git reflog | grep -m 1 'moving from feature'
git log -1 HEAD@{3}                        # inspect a candidate before recreating
                                           the branch
```

Deleting a branch deletes the 41-byte file, nothing else. Git even prints the commit it pointed to, so the quickest fix is to recreate the name at that commit. If the message is gone, `HEAD`'s reflog has an entry from each time you switched away from the branch, and the commit you were on at that moment is the branch's tip, or close to it. Inspect the candidate with `git log` before recreating the branch.

EXERCISE 10.3 Tildes and carets

On `main`, commits A then B; a branch `side` starts at A and adds S; then `side` is merged into `main`. Name the commits that `HEAD^`, `HEAD^2`, `HEAD~1`, `HEAD~2` and `HEAD^2~1` refer to.

SOLUTION

<code>HEAD^</code>	B	the merge's first parent: what main pointed to before the merge
<code>HEAD^2</code>	S	the merge's second parent: the branch that was merged in
<code>HEAD~1</code>	B	one step back, following first parents
<code>HEAD~2</code>	A	two steps back, following first parents
<code>HEAD^2~1</code>	A	the second parent, then one step back from it

`^` chooses among a commit's parents; `~` walks back through first parents. They only differ at merges, which have two. The first parent is always the branch you were on when you merged, so following `~` from the tip of `main` walks `main`'s own history and skips the details of every merged branch. That's what `git log --first-parent` shows, and why merge commits, chapter 11's subject, keep a readable summary of a project. Running `git log -1 --format=%s` on each of the five confirms the answers.

CHECKLIST

- ☐ Can you say what a branch, a tag and `HEAD` each contain?
- ☐ Can you explain what moves when you commit, and what doesn't?
- ☐ Can you recognise a detached `HEAD`, and keep commits made there?
- ☐ Can you find and rescue an unreachable commit with the reflog?
- ☐ Can you choose between `reset --soft`, `--mixed`, `--hard` and `restore`?

GO DEEPER

BOOK [Pro Git: Branches in a Nutshell](#) Scott Chacon & Ben Straub

PRACTICE [Learn Git Branching](#) Peter Cottle

BOOK [Pro Git: Reset Demystified](#) Scott Chacon & Ben Straub

Merging, rebasing and conflicts

3 exercises

*Merging combines two lines of work by comparing each with the last commit they share, the **merge base**. A change made on only one side is taken. A place changed on both sides, in different ways, is a **conflict**: Git can't know which you meant, so it marks the spot and asks you. Rebasing does something else: it replays your commits on top of the other line of work, as new commits with new names. Both can produce the same files. They differ in the history they leave behind, and in who gets hurt when they're used on commits other people already have.*

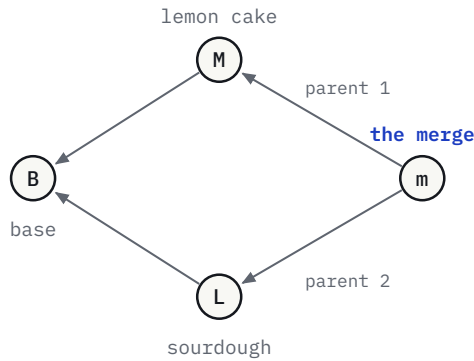
Two lines of work

Two branches that have each gained commits since they last agreed have **diverged**. The commit they last agreed on is their merge base, which `git merge-base main lunch` finds by walking back through both histories. If one side has no commits of its own, there's nothing to combine: merging just moves the branch forward to the other's tip, a **fast-forward**, and no merge commit is made. Otherwise Git performs a **three-way merge** and records the result in a **merge commit**, which has two parents and so keeps both histories.

The three-way merge

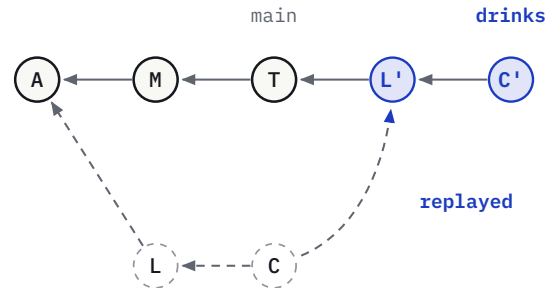
For each part of each file, Git looks at three versions: the merge base, ours (the branch you're on) and theirs (the one being merged in). The rule is simple. If neither side changed it, keep it. If only one side changed it, take that side's version: the other side's version is just the old one. If both sides made the same change, take it once. If both changed it differently, it's a conflict. Comparing with the base is what makes this work. With only the two tips, Git couldn't tell "I changed this line" from "you changed it", and every difference would be a question. Git first lines the versions up with a diff, so it works on **hunks**, runs of changed lines, rather than line numbers, and inserted or deleted lines don't throw it off. Changes that touch or sit right next to each other count as the same hunk, so two edits on neighbouring lines conflict even though they don't overlap.

MERGE: A COMMIT WITH TWO PARENTS



line 2 from lunch, line 5 from main

REBASE: THE SAME CHANGES, NEW COMMITS



the old commits: no name points at them now

new ids, the same lines added and removed

A merge joins two histories with a commit that has two parents. A rebase rewrites one history so it starts later: the same changes, as new commits.

The example works on a seven-line menu. Two branches change different lines and merge cleanly, then a third changes the same line as one of them and conflicts. A fourth moves forward without a merge commit, and two last commits are rebased onto `main`. Each command happens a minute after the last, so every id is reproducible.

EXAMPLE

`merge.py` · a clean three-way merge laid out line by line, a conflict and its markers resolved by hand, a fast-forward, and a rebase with its new commit ids

```
import os, pathlib, subprocess, tempfile

ENV = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C",
      "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull,
      "GIT_AUTHOR_NAME": "Ada Lovelace", "GIT_AUTHOR_EMAIL": "ada@example.org",
      "GIT_COMMITTER_NAME": "Ada Lovelace", "GIT_COMMITTER_EMAIL": "ada@example.org"}
clock = [1772355600]

def git(*args, cwd, check=True):
    clock[0] += 60
    the last
    env = dict(ENV, GIT_AUTHOR_DATE=f"@{clock[0]} +0000", GIT_COMMITTER_DATE=f"@{clock[0]} +0000")
    r = subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True,
                       check=check)
    return r.stdout.rstrip() if check else r

def edit(repo, branch, line, text, message):
    """Switch to a branch, replace one line of menu.txt, and commit."""
    git("switch", "-q", branch, cwd=repo)
    lines = (repo / "menu.txt").read_text().splitlines(); lines[line - 1] = text
    (repo / "menu.txt").write_text("\n".join(lines) + "\n")
    git("commit", "-q", "-am", message, cwd=repo)

def version(repo, rev): return git("show", f"{rev}:menu.txt", cwd=repo).splitlines()

def graph(repo):
    for line in git("log", "--graph", "--format=%h %s%d", "main", cwd=repo).splitlines():
        print(f"    {line}".rstrip())
```

```

with tempfile.TemporaryDirectory() as tmp:
    repo = pathlib.Path(tmp); git("init", "-q", "-b", "main", cwd=repo)
    (repo / "menu.txt").write_text("Soup\nBread\nSalad\nPasta\nCake\nTea\nWater\n")
    git("add", ".", cwd=repo); git("commit", "-q", "-m", "Base menu", cwd=repo)
    for b in ("lunch", "dinner", "drinks"): git("branch", b, cwd=repo)
    edit(repo, "lunch", 2, "Sourdough", "Lunch: sourdough")
    edit(repo, "main", 5, "Lemon cake", "Main: lemon cake")

    print("1. A merge compares each side with the merge base, the last commit both sides
          share")
    base = git("merge-base", "main", "lunch", cwd=repo)
    print(f"    merge base of main and lunch: {base[:7]} {git('log', '-1', '--format=%s', base,
          cwd=repo)}")
    b, ours, theirs = version(repo, base), version(repo, "main"), version(repo, "lunch")
    git("merge", "-q", "--no-edit", "lunch", cwd=repo)
    result = (repo / "menu.txt").read_text().splitlines()
    print(f"    {'line':5}{base[:12]}{'main (ours)':14}{lunch (theirs)':16}{rule:24}merged")
    for i, (x, o, t, m) in enumerate(zip(b, ours, theirs, result), 1):
        rule = "nobody changed it" if o == t == x else "only theirs changed" if o == x else \
              "only ours changed" if t == x else "both, the same way" if o == t else "both:
              conflict"
        print(f"    {i:<5}{x:12}{o:14}{t:16}{rule:24}{m}")

    print("2. A conflict: both sides changed the same line in different ways")
    edit(repo, "dinner", 2, "Garlic bread", "Dinner: garlic bread")
    git("switch", "-q", "main", cwd=repo)
    r = git("merge", "--no-edit", "dinner", cwd=repo, check=False)
    print(f"    git merge dinner exits with status {r.returncode}; unmerged: "
          f"{git('diff', '--name-only', '--diff-filter=U', cwd=repo)}")
    for line in (repo / "menu.txt").read_text().splitlines()[7]: print(f"        {line}")
    resolved = ["Soup", "Sourdough or garlic bread", "Salad", "Pasta", "Lemon cake", "Tea",
               "Water"]
    (repo / "menu.txt").write_text("\n".join(resolved) + "\n")
    git("add", "menu.txt", cwd=repo); git("commit", "-q", "--no-edit", cwd=repo)
    print("    resolved by hand, added, committed:"); graph(repo)

    print("3. When one side has no new commits, merging just moves the branch forward")
    git("switch", "-q", "-c", "tea", cwd=repo); edit(repo, "tea", 6, "Mint tea", "Tea: mint")
    git("switch", "-q", "main", cwd=repo); git("merge", "-q", "--ff-only", "tea", cwd=repo)
    same = git("rev-parse", "main", cwd=repo) == git("rev-parse", "tea", cwd=repo)
    print(f"    main and tea are now the same commit: {same}")

    print("4. A rebase replays commits on a new base: same changes, new commits, new ids")
    git("switch", "-q", "drinks", cwd=repo)
    for drink in ("Lemonade", "Coffee"):
        with (repo / "menu.txt").open("a") as f: f.write(drink + "\n")
        git("commit", "-q", "-am", f"Drinks: {drink.lower()}", cwd=repo)
    before = git("rev-list", "--reverse", "main..drinks", cwd=repo).split()
    git("rebase", "-q", "main", cwd=repo)
    after = git("rev-list", "--reverse", "main..drinks", cwd=repo).split()
    def patch(rev):
        diff = git("show", "--format=", "-U0", rev, cwd=repo).splitlines()
        return [l for l in diff if l[:1] in "+-" and not l.startswith(("+++", "---"))]
    for old, new in zip(before, after):
        print(f"    {git('log', '-1', '--format=%s', new, cwd=repo):18} {old[:7]} -> {new[:7]}")
        f"same change: {patch(old) == patch(new)}")
    git("switch", "-q", "main", cwd=repo); git("merge", "-q", "--ff-only", "drinks", cwd=repo)
    print("    then main fast-forwards to it, and history stays a straight line from the
          merge:")
    graph(repo)

```

OUTPUT

```

1. A merge compares each side with the merge base, the last commit both sides share
merge base of main and lunch: 4e0be3c Base menu

```

line	base	main (ours)	lunch (theirs)	rule	merged
1	Soup	Soup	Soup	nobody changed it	Soup
2	Bread	Bread	Sourdough	only theirs changed	Sourdough
3	Salad	Salad	Salad	nobody changed it	Salad
4	Pasta	Pasta	Pasta	nobody changed it	Pasta
5	Cake	Lemon cake	Cake	only ours changed	Lemon cake
6	Tea	Tea	Tea	nobody changed it	Tea
7	Water	Water	Water	nobody changed it	Water

2. A conflict: both sides changed the same line in different ways
git merge dinner exits with status 1; unmerged: menu.txt

```

Soup
<<<<<< HEAD
Sourdough
=====
Garlic bread
>>>>>> dinner
Salad

```

resolved by hand, added, committed:

```

* 23d796f Merge branch 'dinner' (HEAD -> main)
|\
| * 86a5223 Dinner: garlic bread (dinner)
* | c3ca78a Merge branch 'lunch'
|\ \
| * | a96b1aa Lunch: sourdough (lunch)
| | /
* / 3288f75 Main: lemon cake
| /
* 4e0be3c Base menu (drinks)

```

3. When one side has no new commits, merging just moves the branch forward
main and tea are now the same commit: True

4. A rebase replays commits on a new base: same changes, new commits, new ids

```

Drinks: lemonade e18dd94 -> 6248552 same change: True
Drinks: coffee c5b9e4a -> f8fe437 same change: True

```

then main fast-forwards to it, and history stays a straight line from the merge:

```

* f8fe437 Drinks: coffee (HEAD -> main, drinks)
* 6248552 Drinks: lemonade
* 1cd8cf6 Tea: mint (tea)
* 23d796f Merge branch 'dinner'
|\
| * 86a5223 Dinner: garlic bread (dinner)
* | c3ca78a Merge branch 'lunch'
|\ \
| * | a96b1aa Lunch: sourdough (lunch)
| | /
* / 3288f75 Main: lemon cake
| /
* 4e0be3c Base menu

```

What the example shows

The rule, line by line. The table in section 1 is the three-way merge made visible. Line 2 changed only on `lunch`, so it was taken from `lunch`; line 5 changed only on `main`, so it stayed; everything else agreed. Git's merged file matched the rule on every line, and Git asked nothing.

A conflict is a question Git leaves in the file. Since the merge base, `main` had changed line 2 to `Sourdough`, and `dinner` had changed it to `Garlic bread`. `git merge` exited with status 1 and listed `menu.txt` as unmerged. In the file, between `<<<<<< HEAD` and `=====` is our version, and between that and `>>>>>> dinner` theirs; everything outside the

markers merged normally. Resolving means editing the file into what it should be, here a line that keeps both ideas, then `git add` and `git commit`, which made the merge commit. The graph shows both merges, each with two parents.

A rebase makes new commits. `tea` was made from `main`'s tip, so merging it was a fast-forward: `main` moved to the same commit, with no merge commit. `drinks` had been started from the very first commit. Rebasing it onto `main` replayed its two commits on top of `main`'s tip. Each came out with a new id, because its parent and its tree changed, as chapter 9 predicted, but the lines each one added and removed were exactly the same. The old commits still exist, unreferenced, until the reflog lets them go. Afterwards `main` fast-forwarded to `drinks`, and the newest part of history is a straight line.

Resolving conflicts well

A conflict means two people had different intentions for the same lines, so resolve it by understanding both, not by picking a side. `git log --merge -p file` shows the commits on each side that touched the file. Setting `merge.conflictStyle` to `zdiff3` adds a third section to each conflict, the base version, so you can see what each side changed rather than just what each side ended with. It's the most useful Git setting few people know, and belongs in your dotfiles. Once the file is right, run the tests before committing, because a merge can be textually clean and still wrong: one side renames a function while the other adds a new call to it. If it gets confusing, `git merge --abort` or `git rebase --abort` puts everything back as it was. `git diff --check` catches leftover markers before you commit them.

Merge or rebase?

Both are good tools for different jobs. **Rebase your own work** to keep it current and tidy before you share it. Replaying your unpushed commits on top of the latest `main` means your eventual merge is simple, and an interactive rebase, `git rebase -i`, lets you reorder, combine and reword them first. **Merge to combine shared work**, because a merge adds a commit and changes none. And follow the one rule that matters: **never rebase commits that other people already have**. To them, your rebased commits are new and unrelated, and their next pull brings in both versions. `git pull --rebase` is safe, because it only replays the local commits you haven't pushed.

COMMON TRAPS

Committing conflict markers; `git diff --check` finds them.

Rebasing a branch that others have pulled, which duplicates its commits for everyone.

Resolving by taking one side wholesale without reading what the other side changed.

Long-lived branches that drift until merging them takes days; integrate often.

Forgetting that `--abort` exists, and improvising in the middle of a merge.

EXERCISES

EXERCISE 11.1 Apply the rule

For each of five lines, far enough apart that their changes don't touch, give the merged result or say it conflicts: (1) base `a`, ours `a`, theirs `b`; (2) base `x`, ours `y`, theirs `y`; (3) base `p`, ours `q`, theirs `r`; (4) all three `m`; (5) base `k`, ours deleted the line, theirs `k`.

SOLUTION

(1) `b`: only theirs changed it. (2) `y`: both made the same change, which is taken once. (3) A conflict: both changed it, differently. (4) `m`: nobody changed it. (5) The line is deleted: only ours changed it, and a deletion is a change like any other. `git merge-file`, which runs Git's three-way merge on three files, agrees on all five, with a single conflict on line 3. Case 5 surprises people. If theirs had edited line `k` while ours deleted it, that would be a conflict, because Git can't know whether the edit matters more than the deletion.

EXERCISE 11.2 The rebase that hurt

You pushed a branch, a colleague pulled it and built on it, and then you rebased the branch onto the latest `main` and force-pushed. What happens when your colleague next pulls, and how should each of you recover?

SOLUTION

Your colleague's commits sit on top of your *old* commits, A, B and C, while the branch on the server now holds new ones, A', B' and C', with the same changes and different ids. A plain `git pull` merges the two histories, so their branch now contains every change twice, once in each set of commits. Any of them that touched the same lines conflicts with its own copy. To recover, your colleague should replay only their own commits onto the new branch: `git pull --rebase` usually manages that by itself, because it recognises commits whose changes are already upstream and skips them; failing that, `git rebase --onto origin/branch C` moves exactly the commits after your old C. The lesson for you is the rule: once a branch is shared, add commits to it, or

ask first. If you do need to replace a shared branch's history, push with `--force-with-lease`, chapter 13's safer `force`, and tell everyone who has it.

EXERCISE 11.3 Three buttons

A pull request has three commits. GitHub offers to merge it with a merge commit, to squash and merge, or to rebase and merge. What does `main`'s history look like after each?

SOLUTION

Create a merge commit: the three commits, unchanged and with their ids, plus a merge commit whose second parent is the last of them. `git log --first-parent main` shows one entry for the whole pull request, and the details are one level down. **Squash and merge:** one new commit on `main` containing all three commits' changes together, with the pull request's title as its message; the original three aren't in `main`'s history at all. **Rebase and merge:** the three commits replayed onto `main` as new commits with new ids, in a straight line, with no merge commit. Squashing gives the tidiest history and loses the steps, which matters to chapter 12's `bisect` if a pull request was large. Merging keeps everything, and rebasing keeps the steps without the merge. Teams pick one and say so in their contributing guide.

CHECKLIST

- ☐ Can you find the merge base of two branches, and say when a merge will fast-forward?
- ☐ Can you apply the three-way rule to predict a merge's result?
- ☐ Can you read conflict markers, resolve them by understanding both sides, and finish or abort the merge?
- ☐ Can you explain why a rebased commit has a new id but the same change?
- ☐ Can you say when to merge, when to rebase, and which commits must never be rebased?

GO DEEPER

BOOK Pro Git: Basic Branching and Merging Scott Chacon & Ben Straub

BOOK Pro Git: Rebasing Scott Chacon & Ben Straub

History as a debugging tool

3 exercises

A repository's history isn't just a backup. It's the best debugging tool you have for the bug that begins "it used to work". `git bisect` searches history for the commit that broke something, automatically if you give it a test, and finds it among a thousand commits in ten steps. `git log -S` finds the commits that added or removed a piece of text, and `git blame` says which commit last changed each line, which leads to why. All three are only as good as the history they search: small commits, with messages that explain why.

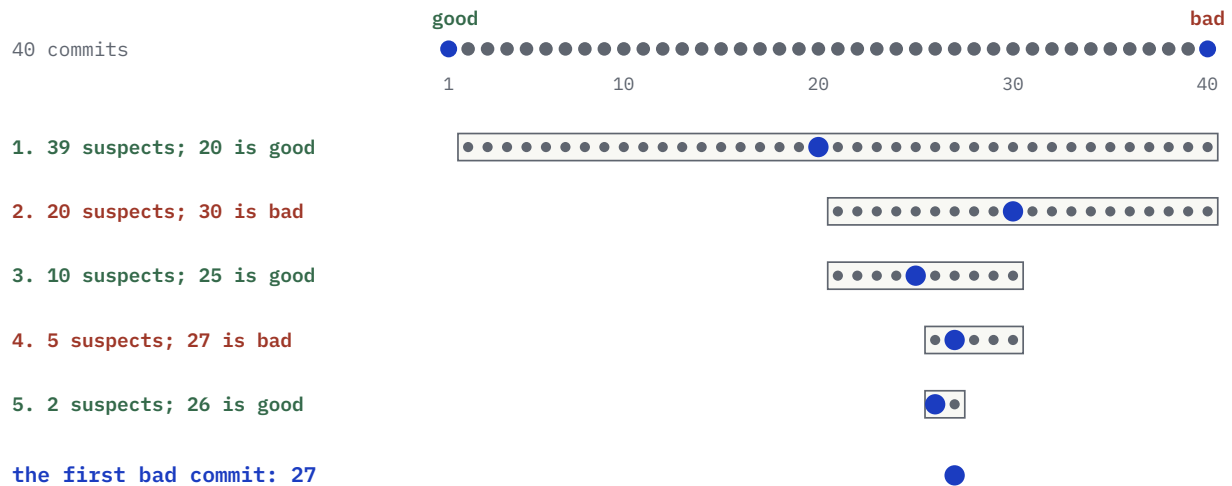
Reading history

`git log` shows commits newest first, and a handful of options turn it into a query. `--oneline` gives one line per commit, `--graph --all` draws every branch, `-p` adds each commit's changes, and `--stat` lists the files each touched. `-n 5`, `--since=2.weeks` and `-author=Ada` limit it, and `-- path` at the end limits it to commits that changed one file or folder. `git log main..feature` lists commits on `feature` that aren't on `main`, which is what a pull request contains, and `git show` displays one commit in full. Put a `lg` alias for `log --graph --oneline --all` in your dotfiles and you'll use it every day.

Bisect: binary search over history

When something worked at an old commit and doesn't now, the commit that broke it is somewhere in between, and checking every one would take as long as there are commits. `git bisect` does a binary search instead. You mark one commit good and one bad, and Git checks out the commit halfway between them, on a detached `HEAD`. You test it and say `git bisect good` or `git bisect bad`, which throws away half the suspects. Repeat, and after about $\log_2$ of the number of commits, Git names the first bad one: ten steps for a thousand commits, twenty for a million. `git bisect reset` returns you to where you started.

Better still, give bisect a command and let it do the testing: `git bisect run ./check.sh`. Its exit status decides. 0 means good, 125 means "this commit can't be tested, skip it", and any other status from 1 to 127 means bad. Recent versions of Git also stop with an error if the test can't even run, status 126 or 127, on the commit you marked good. That catches a missing or non-executable script before it can blame an innocent commit. But nothing can catch a test that runs and checks the wrong thing.



Each test halves the suspects. The search needs to know only good or bad for each commit it tests, so a script can answer for you.

The example builds a small repository with forty commits. The first adds a `slug()` function that turns a recipe's title into a web address, and commit 27, "Simplify slug()", quietly breaks it for accented letters. A test script checks one title. The example lets `bisect run` find the culprit, then asks `log -S` and `blame` the same question, and undoes the damage with `revert`.

EXAMPLE

history.py · forty commits, one that breaks a function, found by `git bisect run` with a test script, then by `log -S` and `blame`, and undone with `revert`

```
import os, pathlib, re, subprocess, sys, tempfile

ENV = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C",
      "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull,
      "GIT_AUTHOR_NAME": "Ada Lovelace", "GIT_AUTHOR_EMAIL": "ada@example.org",
      "GIT_COMMITTER_NAME": "Ada Lovelace", "GIT_COMMITTER_EMAIL": "ada@example.org"}
clock = [1772355600]

def git(*args, cwd):
    clock[0] += 60
    env = dict(ENV, GIT_AUTHOR_DATE=f"@{clock[0]} +0000", GIT_COMMITTER_DATE=f"@{clock[0]} +0000")
    return subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True,
                          check=True).stdout.rstrip()

GOOD = '''import re, unicodedata

def slug(title):
    """Make a web address from a title: "Creme Brulee, 2 Ways!" -> "creme-brulee-2-ways"."""
    plain = unicodedata.normalize("NFKD", title).encode("ascii", "ignore").decode()
    return re.sub(r"^[a-z0-9]+", "-", plain.lower()).strip("-")
...

BAD = GOOD.replace('    plain = unicodedata.normalize("NFKD", title).encode("ascii",
    "ignore").decode()\n'
                  '        return re.sub(r"^[a-z0-9]+", "-", plain.lower()).strip("-")\n',
                  '        return re.sub(r"^[a-z0-9]+", "-", title.lower()).strip("-")\n')

CHECK = ('from recipes import slug\n'
        'raise SystemExit(slug("Crème Brûlée, 2 Ways!") != "creme-brulee-2-ways")\n')
```

```

with tempfile.TemporaryDirectory() as tmp:
    root = pathlib.Path(tmp); repo = root / "site"; repo.mkdir()
    git("init", "-q", "-b", "main", cwd=repo)
    (repo / "recipes.py").write_text(GOOD); (repo / "list.txt").write_text("")
    git("add", ".", cwd=repo); git("commit", "-q", "-m", "Add slug() for recipe addresses",
        cwd=repo)
    number = {git("rev-parse", "HEAD", cwd=repo): 1}
    for n in range(2, 41): # 39 more commits; number 27 breaks
        slug()
        if n == 27:
            (repo / "recipes.py").write_text(BAD); message = "Simplify slug()"
        else:
            with (repo / "list.txt").open("a") as f: f.write(f"recipe {n}\n")
            message = f"Add recipe {n}"
            git("commit", "-q", "-am", message, cwd=repo)
            number[git("rev-parse", "HEAD", cwd=repo)] = n
    (root / "check.py").write_text(CHECK)
    (root / "check.sh").write_text(f'#!/bin/sh\nPYTHONPATH=. exec "{sys.executable}"
        ../check.py\n')
    (root / "check.sh").chmod(0o755)
    run = lambda: subprocess.run(["../check.sh"], cwd=repo, env=ENV,
        capture_output=True).returncode
    first = git("rev-list", "--max-parents=0", "HEAD", cwd=repo)
    print(f"40 commits. The test ../check.sh exits with status {run()} at HEAD: something
        broke.")

    print("1. git bisect run: a binary search over history, with the test deciding good or
        bad")
    git("bisect", "start", "HEAD", first, cwd=repo)
    git("bisect", "run", "../check.sh", cwd=repo)
    bad = git("rev-parse", "refs/bisect/bad", cwd=repo)
    steps = [l.split()[2:4] for l in git("bisect", "log", cwd=repo).splitlines()
        if l.startswith(("git bisect good ", "git bisect bad "))]
    tried = [f"{number[h]} ({verdict})" for verdict, h in steps]
    print(f"    it tested commits {' '.join(tried[:-1])} and {tried[-1]}")
    print(f"    {len(steps)} tests for 39 suspects; the first bad commit is number
        {number[bad]}: "
        f"{git('log', '-1', '--format=%h %s', bad, cwd=repo)}")
    git("bisect", "reset", cwd=repo)

    print("2. git log -S: the commits that added or removed a piece of text")
    for line in git("log", "-S", "normalize", "--format=%h %s", cwd=repo).splitlines():
        print(f"    {line}")

    print("3. git blame: which commit last changed each line")
    blame = git("blame", "--line-porcelain", "-L", "3,5", "recipes.py", cwd=repo).splitlines()
    commits = [l.split()[0] for l in blame if re.match(r"[0-9a-f]{40} \d+ \d+", l)]
    summaries = [l[len("summary "):] for l in blame if l.startswith("summary ")]
    code = [l[1:] for l in blame if l.startswith("\t")]
    for h, summary, line in zip(commits, summaries, code):
        print(f"    {h[:7]} {summary[:31]:31} | {line}")

    print("4. git revert: a new commit that undoes an old one, without rewriting history")
    git("revert", "--no-edit", bad, cwd=repo)
    print(f"    {git('log', '-1', '--format=%h %s', cwd=repo)}; the test now exits with status
        {run()}")

```

OUTPUT

```

40 commits. The test ../check.sh exits with status 1 at HEAD: something broke.
1. git bisect run: a binary search over history, with the test deciding good or bad
    it tested commits 20 (good), 30 (bad), 25 (good), 27 (bad) and 26 (good)
    5 tests for 39 suspects; the first bad commit is number 27: 8ec864a Simplify slug()
2. git log -S: the commits that added or removed a piece of text
    8ec864a Simplify slug()

```

```

dae9473 Add slug() for recipe addresses
3. git blame: which commit last changed each line
dae9473 Add slug() for recipe addresses | def slug(title):
dae9473 Add slug() for recipe addresses |     """Make a web address from a title: "Creme
    Brulee, 2 Ways!" -> "creme-brulee-2-ways"."""
8ec864a Simplify slug() |     return re.sub(r"^[a-z0-9]+", "-",
    title.lower()).strip("-")
4. git revert: a new commit that undoes an old one, without rewriting history
e53f276 Revert "Simplify slug()"; the test now exits with status 0

```

What the example shows

Bisect needed five tests. It tried commit 20, which was good, then 30, bad, 25, good, 27, bad, and 26, good, and concluded that 27 was the first bad commit. That's every possible culprit among 39 suspects, commits 2 to 40, settled in five runs of the test, with no human involved. On a real project, with a test suite that takes minutes, this is the difference between a ten-minute search and an afternoon.

Text search and blame answer different questions. `git log -S normalize` listed the two commits that changed how many times the word `normalize` appeared: the one that added it and the one that removed it. Without knowing where the bug was, that found it from a guess about what had changed. `git blame` on the function gave each line's last commit: the first two lines date from the original, and the broken `return` line from "Simplify slug()". The example reads blame's machine-readable `--line-porcelain` output, the right choice for scripts; at a terminal, plain `git blame` marks lines from the very first commit with a `^`. Finally, `git revert` made a new commit that applies the opposite of the bad one, and the test passed again. History wasn't rewritten, so it's safe even after the bad commit has been shared.

Searching and blaming well

`-S text`, the "pickaxe", finds commits that change the number of occurrences of the text, so it finds where a function was added or deleted, but not where an existing call was edited. For that, `-G regex` finds commits whose changed lines match. `git blame -w` ignores changes of whitespace, and `-C` follows lines that were moved or copied from other files. Large reformatting commits make blame useless for every line they touched. List them in a `.git-blame-ignore-revs` file, and `git blame --ignore-revs-file`, or GitHub's blame view, will look past them. Remember what blame is for: a line's commit leads to its message, its pull request and its discussion. That's *why* the line is the way it is, which is far more useful than who typed it.

History worth searching

Each of these tools depends on how history was written. When bisect finds a five-line commit that does one thing, the bug is obvious. When it finds a two-thousand-line commit titled "updates", you've learned little. So make **small commits that do one thing each**, and make sure each one builds and passes the tests, so bisect can test any of them. Write **messages that explain why**. The first line is a short summary in the imperative, around fifty

characters, as in "Simplify slug()", then a blank line, then whatever a reader will need in a year: the reason for the change, what you considered, what it fixes. `git commit` without `-m` opens your editor so there's room for that, which is one reason chapter 14 matters.

COMMON TRAPS

Huge commits that mix unrelated changes, which bisect can only point at, not explain.

Messages like "fix" or "wip", which tell blame's reader nothing.

Forgetting `git bisect reset`, and carrying on work on a detached `HEAD`.

A bisect test that checks the wrong thing, which makes bisect blame an innocent commit.

Undoing a shared commit with `reset` instead of `revert`.

EXERCISES

EXERCISE 12.1 A thousand commits

A bug appeared somewhere in the last 1,000 commits. About how many tests will bisect need? Some old commits don't build at all; write a `bisect run` script for a project tested with `make test` that copes with them.

SOLUTION

```
#!/bin/sh
# check.sh: tell git bisect whether this commit is good (0), bad (1) or untestable
(125)
make > /dev/null 2>&1 || exit 125      # doesn't build: neither good nor bad, so skip
it
make test > /dev/null 2>&1 || exit 1   # builds, tests fail: bad
exit 0                                # builds and passes: good
```

About 10, since 2^{10} is 1,024: each test halves the suspects. The script turns a build failure into 125, so bisect skips that commit and tests a neighbour instead, rather than wrongly blaming a commit that only failed to build. In a test with a skipped commit in the middle of the range, bisect still named the right first bad commit. Keep the script outside the repository, or untracked, so it doesn't change as bisect checks out old commits, and make it exit 1 for a real failure rather than any status above 127, which would stop the search.

EXERCISE 12.2 A message for the future

A commit changes the API client's timeout from `timeout=30` to `timeout=30_000`, because the library it uses changed from seconds to milliseconds in version 4. Its message is "fixed stuff". Write a better one.

SOLUTION

Give the API client's timeout in milliseconds

Version 4 of the HTTP library takes timeouts in milliseconds instead of seconds, so after the upgrade our 30-second timeout became 30 ms, and every request to the slower reports endpoint timed out.

Keep the same 30-second timeout, now written as 30_000.

The summary says what the commit does, in the imperative, short enough to read in `git log --oneline`. The body explains what nobody can recover from the change itself: why `30` became `30_000`, which would otherwise look like a typo someone might "fix" back. The symptom it cured is there for whoever searches for it, and the last line records that the behaviour was meant to stay the same. Someone who finds this line with blame next year will know not to touch it, which is the point.

EXERCISE 12.3 Which undo?

Choose the command for each: (a) you've edited a file and want it back as it was in the last commit; (b) your last commit, not yet pushed, has a typo in its message; (c) a commit from last week, already on `main`, broke production; (d) your last three local commits should have been one.

SOLUTION

(a) `git restore file`, which throws away the uncommitted edits in that file, for good, so be sure. (b) `git commit --amend`, which replaces the last commit with a corrected one, fine because nobody else has it yet. (c) `git revert <commit>`, which adds a new commit that undoes it and changes no existing history, the only safe choice for a commit others have. (d) `git reset --soft HEAD~3` then `git commit`, which moves the branch back three commits while keeping all their changes staged, ready to commit as one; `git rebase -i HEAD~3` does the same with more control. The test in each case is chapter 11's rule: has anyone else got the commit? If so, add a commit; if not, you may replace it.

CHECKLIST

- ☐ Can you query history by file, author, date and branch with `git log`?
- ☐ Can you find the commit that introduced a bug with `git bisect`, by hand and with `run`?
- ☐ Can you find when a piece of code appeared or disappeared with `git log -S`?
- ☐ Can you use `git blame` to find why a line is the way it is?
- ☐ Can you choose between `restore`, `amend`, `reset` and `revert`?

GO DEEPER

DOCS [git bisect](#) The Git project

ARTICLE [How to Write a Git Commit Message](#) Chris Beams

BOOK [Pro Git: Debugging with Git](#) Scott Chacon & Ben Straub

Remotes, forks and pull requests

3 exercises

A **remote** is another copy of the repository, usually on a server. `git fetch` copies the commits you don't have and moves your record of the remote's branches. `git push` sends your commits and asks the other side to move one of its branches, which it agrees to only if nothing would be lost. GitHub builds a way of working on top of that: forks, pull requests, review and issues. This chapter simulates GitHub with plain repositories in a folder, and follows one contribution from the first commit to the merge.

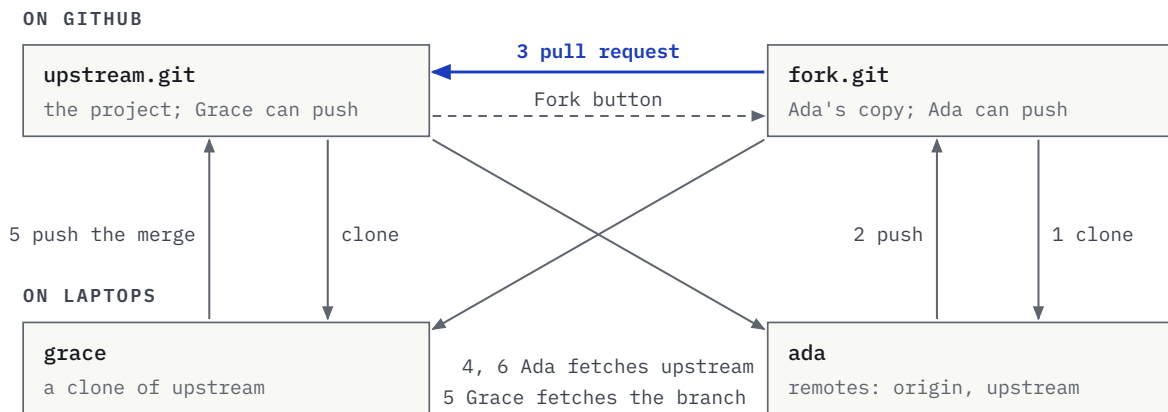
Remotes, fetch and push

When you clone a repository, Git records where it came from as a remote called `origin`, and `git remote -v` lists your remotes and their addresses. For each branch on a remote, Git keeps a **remote-tracking branch**, such as `origin/main`: a ref recording where that branch was the last time you talked to the server. `git fetch` downloads any commits you're missing and updates those refs; it never touches your own branches. So `origin/main` is a snapshot, possibly hours old, and comparisons with it are only as fresh as your last fetch. `git pull` is a fetch followed by a merge of the remote branch into yours, or a rebase onto it if you've configured that.

`git push origin feature` sends your commits and asks the remote to move its `feature` to match yours. The remote refuses unless the move is a fast-forward, meaning your branch already contains everything it has. Otherwise, commits someone else pushed would be lost. When the push is refused, fetch, combine their work with yours, and push again. `git push -u origin feature` also sets `origin/feature` as your branch's **upstream**, after which plain `git push` and `git pull` know where to go, and `git status` says how many commits you're ahead or behind.

Forks and pull requests

On GitHub, you usually can't push to a project you don't maintain. So you **fork** it, which makes your own copy of the repository on GitHub's servers, clone your fork, and push branches there. A **pull request** asks the project's maintainers to merge one of your fork's branches into their repository. Around it, GitHub runs the conversation. Reviewers comment on lines, ask for changes or approve; automated checks run the tests; and every new commit you push to the branch updates the pull request. When it's ready, a maintainer merges it with one of chapter 11's three buttons. Inside a team with write access, people skip the fork and push branches to the main repository directly, but the pull request, the review and the checks are the same. **Issues** are GitHub's bug reports and discussions. A pull request whose description says `Fixes #12` closes issue 12 when it's merged.



Ada pushes only to her fork; the pull request asks Grace to merge into the project.

A contribution is a round trip: your branch goes to your fork, the pull request asks the project to take it, and the merged result comes back to you from upstream.

The example plays both parts. Bare repositories in a temporary folder stand in for GitHub: `upstream.git` is the project, and `fork.git` Ada's copy of it, made the way the Fork button does, with a server-side clone. Grace maintains the project; Ada contributes a small fix. Each person has their own name and email, and the clock is fixed, so the commit ids are reproducible.

EXAMPLE

`remotes.py` · a project, a fork and two clones on disk: a branch pushed to the fork, a push refused, a rebase onto upstream and a push with a lease, the pull request merged, and the fork brought up to date

```
import os, pathlib, subprocess, tempfile

# Bare repositories in a temporary folder stand in for GitHub: "upstream" is the project,
# "fork" is
# the contributor's copy of it. Grace maintains the project; Ada contributes.
BASE = {"PATH": os.environ["PATH"], "HOME": "/home/learner", "LC_ALL": "C",
        "GIT_CONFIG_NOSYSTEM": "1", "GIT_CONFIG_GLOBAL": os.devnull}
PEOPLE = {"grace": ("Grace Hopper", "grace@example.org"), "ada": ("Ada Lovelace",
        "ada@example.org")}
clock = [1772355600]

def git(who, *args, cwd, check=True):
    clock[0] += 60
    name, email = PEOPLE[who]
    env = dict(BASE, GIT_AUTHOR_NAME=name, GIT_AUTHOR_EMAIL=email, GIT_COMMITTER_NAME=name,
               GIT_COMMITTER_EMAIL=email, GIT_AUTHOR_DATE=f"@{clock[0]} +0000",
               GIT_COMMITTER_DATE=f"@{clock[0]} +0000")
    r = subprocess.run(["git", *args], cwd=cwd, env=env, capture_output=True, text=True)
    if check and r.returncode: raise SystemExit(r.stderr)
    return r.stdout.rstrip() if check else r

def change(who, repo, name, text, message):
    (repo / name).write_text(text)
    git(who, "add", name, cwd=repo); git(who, "commit", "-q", "-m", message, cwd=repo)

def graph(who, repo, *revs):
```

```

hide = "--decorate-refs-exclude=refs/remotes/*/HEAD"    # newer gits add these; older ones
don't
for line in git(who, "log", "--graph", "--format=%h %an: %s%d", hide, *revs,
    cwd=repo).splitlines():
    print(f"        {line}".rstrip())

with tempfile.TemporaryDirectory() as tmp:
    gh = pathlib.Path(tmp).resolve()
    print("1. Grace creates the project; Ada forks it on GitHub and clones her fork")
    git("grace", "init", "-q", "--bare", "-b", "main", "upstream.git", cwd=gh)
    git("grace", "clone", "-q", "upstream.git", "grace", cwd=gh)
    grace, ada = gh / "grace", gh / "ada"
    git("grace", "switch", "-q", "-c", "main", cwd=grace)
    change("grace", grace, "README.md", "# Recipes\nA small recipe site.\n", "Start the
        project")
    git("grace", "push", "-q", "origin", "main", cwd=grace)
    git("ada", "clone", "-q", "--bare", "upstream.git", "fork.git", cwd=gh)    # what the
        Fork button does
    git("ada", "clone", "-q", "fork.git", "ada", cwd=gh)
    git("ada", "remote", "add", "upstream", str(gh / "upstream.git"), cwd=ada)
    for line in git("ada", "remote", "-v", cwd=ada).splitlines():
        name, rest = line.split("\t"); url, kind = rest.rsplit(" ", 1)    # paths may contain
            spaces
        print(f"    Ada's remote {name:8} is {pathlib.Path(url).name} {kind}")

    print("2. Ada works on a branch and pushes it to her fork; that's what a pull request is
        opened from")
    git("ada", "switch", "-q", "-c", "fix-typo", cwd=ada)
    change("ada", ada, "README.md", "# Recipes\nA small site for recipes.\n", "Fix the README's
        wording")
    git("ada", "push", "-q", "-u", "origin", "fix-typo", cwd=ada)
    print(f"    {git('ada', 'status', '-sb', cwd=ada).splitlines()[0]}")

    print("3. Meanwhile Grace pushes to the project; a push that would lose her commit is
        refused")
    change("grace", grace, "CONTRIBUTING.md", "Open a pull request.\n", "Add contributing
        notes")
    git("grace", "push", "-q", "origin", "main", cwd=grace)
    git("ada", "switch", "-q", "main", cwd=ada)
    change("ada", ada, "notes.txt", "todo\n", "A commit made on main by mistake")
    r = git("ada", "push", "--porcelain", "upstream", "main", cwd=ada, check=False)
    print(f"    git push upstream main: status {r.returncode}, "
        f"{[l for l in r.stdout.splitlines() if l.startswith('!')][0].split(chr(9))[-1]}")
    git("ada", "reset", "-q", "--hard", "origin/main", cwd=ada)

    print("4. Ada brings her branch up to date: fetch upstream, rebase onto it, force-push with
        a lease")
    git("ada", "fetch", "-q", "upstream", cwd=ada)
    git("ada", "switch", "-q", "fix-typo", cwd=ada); git("ada", "rebase", "-q",
        "upstream/main", cwd=ada)
    git("ada", "push", "-q", "--force-with-lease", "origin", "fix-typo", cwd=ada)
    graph("ada", ada, "fix-typo")

    print("5. Grace reviews and merges the pull request: a merge commit on the project's main")
    git("grace", "fetch", "-q", str(gh / "fork.git"), "fix-typo", cwd=grace)
    git("grace", "merge", "-q", "--no-ff", "-m", "Merge pull request #1 from ada/fix-typo",
        "FETCH_HEAD",
        cwd=grace)
    git("grace", "push", "-q", "origin", "main", cwd=grace)
    graph("grace", grace, "main")

    print("6. Ada syncs: update her main from upstream, push it to her fork, delete the merged
        branch")
    git("ada", "switch", "-q", "main", cwd=ada)
    git("ada", "pull", "-q", "--ff-only", "upstream", "main", cwd=ada)
    git("ada", "push", "-q", "origin", "main", ":fix-typo", cwd=ada)    # ":name" deletes a

```

```

remote branch
git("ada", "branch", "-q", "-D", "fix-typo", cwd=ada)
same = {git("ada", "rev-parse", r, cwd=gh / repo) for repo, r in
        (("upstream.git", "main"), ("fork.git", "main"), ("ada", "main"), ("grace",
        "main"))}
refs = git("ada", "for-each-ref", "--format=%(refname)", "refs/heads", "refs/remotes",
        cwd=ada).split()
branches = [r.split("/", 2)[2] for r in refs if not r.endswith("/HEAD")]
print(f"    all four copies of main are the same commit: {len(same) == 1}")
print(f"    Ada's branches and remote-tracking branches: {' ', '.join(branches)}")

```

OUTPUT

1. Grace creates the project; Ada forks it on GitHub and clones her fork
 Ada's remote origin is fork.git (fetch)
 Ada's remote origin is fork.git (push)
 Ada's remote upstream is upstream.git (fetch)
 Ada's remote upstream is upstream.git (push)
2. Ada works on a branch and pushes it to her fork; that's what a pull request is opened from
 ## fix-typo...origin/fix-typo
3. Meanwhile Grace pushes to the project; a push that would lose her commit is refused
 git push upstream main: status 1, [rejected] (fetch first)
4. Ada brings her branch up to date: fetch upstream, rebase onto it, force-push with a lease
 * caddb90 Ada Lovelace: Fix the README's wording (HEAD -> fix-typo, origin/fix-typo)
 * 5202fa0 Grace Hopper: Add contributing notes (upstream/main)
 * 35e30f9 Grace Hopper: Start the project (origin/main, main)
5. Grace reviews and merges the pull request: a merge commit on the project's main
 * b997fe3 Grace Hopper: Merge pull request #1 from ada/fix-typo (HEAD -> main, origin/main)
 |\
 | * caddb90 Ada Lovelace: Fix the README's wording
 |/
 * 5202fa0 Grace Hopper: Add contributing notes
 * 35e30f9 Grace Hopper: Start the project
6. Ada syncs: update her main from upstream, push it to her fork, delete the merged branch
 all four copies of main are the same commit: True
 Ada's branches and remote-tracking branches: main, origin/main, upstream/main

What the example shows

Two remotes, two roles. Ada's clone has `origin`, her fork, where she can push, and `upstream`, the project, which she reads from. After `push -u`, `git status -sb` showed `fix-typo...origin/fix-typo`: the branch and its upstream, with nothing ahead or behind.

Pushes that would lose work are refused. While Ada worked, Grace pushed a commit to the project. Ada then tried to push a stray commit on `main` straight to `upstream`, and was refused with `[rejected] (fetch first)`: the remote had a commit she'd never seen, and accepting her push would have discarded it. The example uses `--porcelain` to print that refusal in a stable, machine-readable form. Ada threw the stray commit away with `reset -hard origin/main`. To bring her branch up to date, she fetched `upstream` and rebased `fix-typo` onto `upstream/main`. Rebasing rewrote her one commit, so the push to her fork needed `--force-with-lease`, as exercise 13.2 explains.

The merge and the sync. Grace fetched the branch from Ada's fork, merged it with `--no-ff`, which always makes a merge commit, and used the message GitHub writes, `Merge pull request #1 from ada/fix-typo`. Then Ada updated her own `main` with `pull --ff-only`,

pushed it to her fork, and deleted the merged branch there with the refspec `:fix-typo`, the same as `git push origin --delete fix-typo`. All four copies of `main` ended on the same commit.

A good pull request

Pull requests are read by busy people, so make them easy to review. Keep each one **small and about one thing**: a reviewer can check two hundred lines carefully, and two thousand only superficially. Write a description that says what changed and why, how you tested it, and what you're unsure about, with a screenshot for anything visible and a link to the issue it fixes. Open it as a **draft** when you want early feedback. When reviews come, answer every comment, either with a change or with a reason, and push fixes as new commits so reviewers can see what changed since they last looked. When you review someone else's, be specific and kind, and say which comments must be addressed and which are suggestions.

To talk to GitHub at all, you need to authenticate. Over SSH, make a key with `ssh-keygen -t ed25519` and add the public half to your GitHub account; chapter 6 explained the permissions the private half needs. Over HTTPS, use a credential manager or a token, never your password. GitHub's command-line tool, `gh`, does all of this and more: `gh auth login`, `gh repo fork`, `gh pr create` and `gh pr checkout` turn most of this chapter into single commands.

COMMON TRAPS

`git push --force` over commits someone else added; use `--force-with-lease`.

Working on your fork's `main` instead of a branch, so it drifts from the project's.

One enormous pull request that mixes a refactoring, a feature and a fix.

Treating `origin/main` as `current` without fetching first.

Pulling with a merge by habit, filling history with "Merge branch 'main'" commits; set `pull.ff only` or `pull.rebase true`.

EXERCISES

EXERCISE 13.1 What pull does

You and a colleague both committed to `main` since you last pulled. Describe what `git pull` does with the default settings, with `pull.ff only`, and with `pull.rebase true`. Which would you put in your dotfiles?

SOLUTION

All three start by fetching. With no setting, recent versions of Git refuse and ask you to choose. Older versions, and `pull.rebase false`, merge `origin/main` into your `main`, making a merge commit that joins two lines of the same branch; on a busy team, history fills with them. With `pull.ff only`, the pull succeeds only when it's a fast-forward, and otherwise stops so you can decide. With `pull.rebase true`, your local commits are replayed on top of the colleague's, giving a straight line, which is safe because those commits haven't been pushed yet. A good default for your dotfiles is `pull.ff only`, which never does anything surprising, plus `git pull --rebase` when you know you want it. Many people set `pull.rebase true` instead, which is also a sound choice.

EXERCISE 13.2 A force with a lease

Your pull request's branch is on your fork. A reviewer pushes a small fix-up commit to it, which GitHub allows when maintainers may edit. Not having fetched, you amend your own commit and push with `--force`. What happens? What would `--force-with-lease` have done?

SOLUTION

`--force` replaces the remote branch with yours, whatever it held, so the reviewer's commit vanishes from the branch. Nothing warns either of you. `--force-with-lease` replaces it only if the remote branch is still where your `origin/` ref says it was, the last state you saw. Here it had moved, so the push was refused with `[rejected] (stale info)` and the fix-up survived. Both outcomes were checked with repositories on disk. Then fetch, look at what changed, rebase onto it, and push again. Note the lease is only as good as your remote-tracking ref: if you fetch and push without looking, it's satisfied. Adding `--force-if-includes` also checks that your branch actually contains what you fetched. Neither is needed for ordinary pushes, only after rewriting history.

EXERCISE 13.3 The description

Write the pull request description for Ada's change to the README.

SOLUTION

Reword the README's description of the project

The README described the project as "A small recipe site.", which can be read as a site for small recipes. "A small site for recipes" says what we mean: the site is small, not the recipes.

Only README.md changes; there's no code or test change.

Fixes #3

The title is a summary a maintainer can scan in a list of pull requests. The body says what changed and why, in a sentence each, because even a one-line change needs its reason. It also tells the reviewer what not to look for, here no code or tests, which saves them checking. The last line links the issue, which GitHub will close when the change is merged. For a larger change, add how you tested it, what you're unsure about, and screenshots of anything visible. The same discipline as chapter 12's commit messages, one level up.

CHECKLIST

- ☐ Can you explain what `fetch`, `pull` and `push` each change, and what remote-tracking branches are?
- ☐ Can you explain why a push is refused, and what to do about it?
- ☐ Can you contribute to a project through a fork: branch, push, pull request, update, sync?
- ☐ Can you write a pull request that's easy to review, and respond to a review well?
- ☐ Can you set up SSH keys or a token to push to GitHub?

GO DEEPER

BOOK [Pro Git: Contributing to a Project on GitHub](#) Scott Chacon & Ben Straub

DOCS [About pull requests](#) GitHub Docs

GUIDE [Code Review Developer Guide](#) Google

PART IV

Learning the craft

Three chapters on the skills underneath every other one. The editor where you'll spend more working hours than anywhere else, and how to make it disappear from your attention. Error messages and documentation, which answer most questions before you ask anyone, and how to ask well when they don't. And the science of learning itself: why rereading feels productive and isn't, and what to do instead. The last chapter can be read at any time, and pays off most if you read it first.

One editor, to fluency

3 exercises

Your editor is where you'll spend more working hours than anywhere else, so learning one deeply pays back every day. Fluency means moving without the mouse, opening any file or symbol in a keystroke, editing many places at once, and letting the editor understand your code well enough to rename a function safely. Pick one editor and practise it deliberately, the way chapter 16 describes. Under its best features are simple algorithms, and this chapter builds three of them: fuzzy file finding, search across files, and a rename that knows what each name refers to.

Pick one, and go deep

The choice matters much less than the depth. Visual Studio Code is the most widely used, free, and extensible for any language. JetBrains' editors, such as PyCharm, understand code most deeply out of the box. Vim and its descendant Neovim are modal editors, controlled entirely from the keyboard, steep to learn and very fast once learned, and available on every server. Emacs is a world of its own. Choose one, use it for everything, and give it a year. Whatever you choose, learn enough Vim to survive: it's what opens when Git wants a commit message on a machine you've never configured. `i` starts typing, Escape stops, `:wq` saves and quits, `:q!` quits without saving. Then set `EDITOR` in your dotfiles to the editor you chose.

What fluency means

Fluency is a list of things you can do without thinking. You can open any file by typing a few letters of its name, jump to any function or class by name, go to a definition and back again, and list every reference to a name. You can search the whole project, with a regular expression and a filter by file type, and replace across it after reviewing each change. You can put a cursor on many lines at once, or select every occurrence of a word and type once, rename a symbol everywhere it's used, and format a file on save. You can use the built-in terminal, the Git view to stage single hunks and see blame, and the debugger. The **command palette**, which finds any command by typing part of its name and shows its shortcut, is the key that unlocks the rest: when you don't know how to do something, ask the palette, and note the shortcut it shows you.

Learn the actions in small batches, by practice, not by reading lists. Each week, choose three to five things you do often and slowly, learn their keys, and make yourself use them even when it's slower at first; after a week they're habits. Vim has a built-in thirty-minute course, `vimtutor`. VS Code publishes a one-page shortcut reference for each system. Time a common task, such as finding and renaming a function, before and after a week of practice, and you'll see why the investment pays.

query: mdl

s	r	c	/	s	h	o	p	/	m	o	d	e	l	s	/	d	a	t	a	_	l	o	a	d	e	r	.	p	y		score
																															28.1
										8.1						9.8															10.2

t	e	s	t	s	/	t	e	s	t	_	m	o	d	e	l	s	.	p	y		score										
																															15.5
											9.9		2.8		2.8																

each letter scores 1, plus 8 at the start of a word, plus 2 in the file name;
plus 8 right after the previous match, minus 0.2 for each letter skipped

Fuzzy finding rewards letters that start words, which is why typing the initials of a path finds a file faster than typing its name.

Under the features

Fuzzy finding scores every file path against the letters you've typed, in order, with gaps allowed, and shows the best first. Letters that start a word, after a `/`, `_`, `-` or `.`, are worth most, letters right after the previous match a lot, and letters in the file name rather than the folders a little more; letters skipped in between cost a little. The example finds the best alignment of the query in each path, the way `fzf` and VS Code do. **Search across files** is a scan of every line of every file, as chapter 3's `grep -rn`; `ripgrep` and the editors that use it make it fast by searching files in parallel and skipping those Git ignores. **Rename** is the one that needs real understanding. To rename a function, an editor has to know which uses of the name refer to that function, and which are a different variable, a string, or an attribute that happens to share it.

EXAMPLE

editor.py · a fuzzy file finder that scores letters at word starts, a search across files, and three ways to rename a function, only one of them correct

```
import ast, re

# 1. "Go to file": find the files whose paths contain the query's letters in order, best first.
FILES = ["README.md", "pyproject.toml", "src/shop/__init__.py", "src/shop/models/order.py",
         "src/shop/models/data_loader.py", "src/shop/views/menu.py",
         "src/shop/views/order_detail.py",
         "src/shop/utils/money.py", "tests/test_models.py", "tests/test_order_detail.py",
         "docs/deploy.md"]

def fuzzy(query, path):
    """The best score for finding the query's letters in order in the path, and where they are.
    A letter scores more at the start of a word, right after the previous match, or in the file
    name;
    letters skipped between matches cost a little."""
    q, p, n = query.lower(), path.lower(), len(path)
    name_start = path.rfind("/") + 1
    def value(j): return 1 + 8 * (j == 0 or p[j - 1] in "/_-.") + 2 * (j >= name_start)
    best = {}
    at p[j]
    for i, ch in enumerate(q):
```

```

        for j in range(n):
            if p[j] != ch: continue
            if i == 0: best[(i, j)] = (value(j) - 0.1 * j, [j])
            else:
                options = [(s + value(j) + (8 if k == j - 1 else -0.2 * (j - k - 1)), at + [j])
                           for (pi, k), (s, at) in best.items() if pi == i - 1 and k < j]
                if options: best[(i, j)] = max(options)
        finals = [v for (i, _), v in best.items() if i == len(q) - 1]
        return max(finals) if finals else None

def marked(path, at): return "".join(f"[{c}]" if i in at else c for i, c in enumerate(path))

for query in ("mdl", "tmod", "odet", "ord"):
    ranked = sorted(((fuzzy(query, f), f) for f in FILES if fuzzy(query, f)), key=lambda t: -
                    t[0][0])
    print(f"go to file '{query}':")
    for (score, at), f in ranked[:3]: print(f"    {score:5.1f}    {marked(f, at)}")

# 2. Search in files: every line that matches, with its file and line number, like rg or grep -rn
PROJECT = {"src/shop/models/order.py": "class Order:\n    total = 0\n",
           "src/shop/views/menu.py": "def show(menu):\n    # TODO: prices\n    return menu\n",
           "src/shop/utills/money.py": "def cents(x):\n    return round(x * 100) # TODO: negative?\n"}
print("search for TODO in every file:")
for path, text in sorted(PROJECT.items()):
    for n, line in enumerate(text.splitlines(), 1):
        if re.search(r"TODO", line): print(f"    {path}:{n}: {line.strip()}")

# 3. Rename: text replacement versus a rename that knows what each name refers to
SOURCE = '''def total(items):
    return sum(items)

class Receipt:
    total = 0                                # a different thing with the same name

def report(carts):
    totals = [total(cart) for cart in carts]
    print("total:", total(totals))
'''

def rename_function(source, old, new):
    """Rename the module-level function 'old': its definition, and each use that refers to it."""
    tree = ast.parse(source); spots = []
    def visit(node, shadowed):
        if isinstance(node, ast.FunctionDef) and node.name == old and not shadowed:
            line = source.splitlines()[node.lineno - 1]
            spots.append((node.lineno, line.index(old, node.col_offset)))
        if isinstance(node, ast.Name) and node.id == old and not shadowed:
            spots.append((node.lineno, node.col_offset))
        if isinstance(node, (ast.ClassDef, ast.FunctionDef)):
            # a name assigned inside is a new thing
            assigned = {t.id for n in ast.walk(node) for t in getattr(n, "targets", [])
                       if isinstance(t, ast.Name)}
            args = {a.arg for a in node.args.args} if isinstance(node, ast.FunctionDef) else set()
            shadowed = shadowed or old in assigned | args
            for child in node.body: visit(child, shadowed)
        else:
            for child in ast.iter_child_nodes(node): visit(child, shadowed)
    for child in tree.body: visit(child, False)
    lines = source.splitlines(keepends=True)
    for row, col in sorted(spots, reverse=True):
        lines[row - 1] = lines[row - 1][:col] + new + lines[row - 1][col + len(old):]
    return "".join(lines)

```

```
def changed(before, after):
    return [f"  line {n}: {b.strip()} -> {a.strip()}"
            for n, (b, a) in enumerate(zip(before.splitlines(), after.splitlines()), 1) if a !=
            b]

for label, after in (("replace every 'total'", SOURCE.replace("total", "grand_total")),
                    ("replace whole words only, \\bttotal\\b", re.sub(r"\\bttotal\\b",
                    "grand_total", SOURCE)),
                    ("rename the function, as an editor's Rename Symbol does",
                    rename_function(SOURCE, "total", "grand_total"))):
    print(f"{label}:"); print("\n".join(changed(SOURCE, after)))
compile(after, "after.py", "exec")
```

OUTPUT

```
go to file 'mdl':
 28.1  src/shop/[m]odels/[d]ata_[l]oader.py
 15.5  tests/test_[m]o[d]e[l]s.py
  9.7  src/shop/[m]o[d]e[l]s/order.py
go to file 'tmod':
 42.6  tests/[t]est_[m][o][d]els.py
go to file 'odet':
 41.9  tests/test_[o]rder_[d][e][t]ail.py
 41.5  src/shop/views/[o]rder_[d][e][t]ail.py
 20.0  src/shop/m[o][d][e]ls/da[t]a_loader.py
go to file 'ord':
 31.9  tests/test_[o][r][d]er_detail.py
 31.5  src/shop/views/[o][r][d]er_detail.py
 31.4  src/shop/models/[o][r][d]er.py
search for TODO in every file:
src/shop/utils/money.py:2: return round(x * 100)  # TODO: negative?
src/shop/views/menu.py:2: # TODO: prices
replace every 'total':
line 1: def total(items): -> def grand_total(items):
line 5: total = 0                                # a different thing with the same name ->
      grand_total = 0                            # a different thing with the same name
line 8: totals = [total(cart) for cart in carts] -> grand_totals = [grand_total(cart) for
      cart in carts]
line 9: print("total:", total(totals)) -> print("grand_total:", grand_total(grand_totals))
replace whole words only, \\bttotal\\b:
line 1: def total(items): -> def grand_total(items):
line 5: total = 0                                # a different thing with the same name ->
      grand_total = 0                            # a different thing with the same name
line 8: totals = [total(cart) for cart in carts] -> totals = [grand_total(cart) for cart
      in carts]
line 9: print("total:", total(totals)) -> print("grand_total:", grand_total(totals))
rename the function, as an editor's Rename Symbol does:
line 1: def total(items): -> def grand_total(items):
line 8: totals = [total(cart) for cart in carts] -> totals = [grand_total(cart) for cart
      in carts]
line 9: print("total:", total(totals)) -> print("total:", grand_total(totals))
```

What the example shows

Initials find files. `mdl` picked `src/shop/models/data_loader.py` because its three letters start three words; the same letters in the middle of `test_models.py` scored about half as much. `tmod` found the one test file with `t`, `m`, `o`, `d` at word starts. And `ord` gave a near tie between three files, which is the honest answer to an ambiguous query: type one more letter. Exercise 14.3 tunes the scorer to break that tie the way most people would want.

Text replacement doesn't understand code. Replacing every `total` with `grand_total` changed all four lines that contain it, so it renamed the class attribute `Receipt.total`, which is a different thing. It also turned the variable `totals` into `grand_totals` and changed the printed text `"total:"`. Whole-word replacement, `\btotal\b`, spared `totals` but still broke the other two. The rename that walked Python's syntax tree changed exactly three places: the definition and the two calls. It knew that `total = 0` inside the class body defines something new, and that a string is not a name. Every version still compiles, which is the danger: the wrong renames produce working code that does the wrong thing.

Language servers

Editors used to need separate support for every language. The **Language Server Protocol**, published by Microsoft in 2016, split the job in two. A language server, such as `pyright` or `basedpyright` for Python, `rust-analyzer` for Rust or `gopls` for Go, understands one language: it parses your project, resolves names, and answers questions. Any editor that speaks the protocol can ask them: complete this, where is this defined, what refers to it, rename it, what's wrong with this line. That's why Neovim and VS Code can offer the same rename, and why setting up the language server is the most valuable configuration step for any language you use. Add a formatter, such as `ruff` or `black` for Python or `Prettier` for the web, and a linter that runs as you type, and the editor catches whole classes of mistakes before you run anything. Put the project's settings in the repository, for instance in an `.editorconfig` file for indentation and line endings, so everyone's editor agrees.

COMMON TRAPS

Switching editors every few months, and never getting past the basics of any.

Renaming with find-and-replace across a project, instead of the editor's rename.

Hunting through menus instead of asking the command palette.

Not knowing how to leave Vim when Git opens it for a commit message.

Installing dozens of extensions at once; add one when you've felt the need for it.

EXERCISES

EXERCISE 14.1 Your ten actions

List the ten things you do most often in your editor, find the keyboard shortcut for each, and plan how to make them habits over two weeks.

SOLUTION

VS Code, default keys (Cmd on a Mac, Ctrl on Windows and Linux)	
go to file	Cmd/Ctrl+P
command palette	Cmd/Ctrl+Shift+P
go to definition	F12
find all references	Shift+F12
rename symbol	F2
search in all files	Cmd/Ctrl+Shift+F
select next occurrence	Cmd/Ctrl+D
extra cursor	Option/Alt+Click
show or hide the terminal	Ctrl+`
comment out a line	Cmd/Ctrl+/*

That's a typical list for VS Code; your ten will depend on your work, which is why making the list yourself matters. Watch yourself for a day and note what you do with the mouse or do slowly. For the plan, take five in the first week and five in the second, write them on a card next to your screen, and when you catch yourself reaching for the mouse, undo and do it with the keys. Each takes a few days to become automatic. Every editor lets you change its keys, but learn the defaults first, because they're what tutorials, colleagues and other machines use. In Vim, the equivalents of the first few come from plugins such as `fzf` or Telescope, and from the language server: `gd` to go to a definition, `gr` for references in many setups.

EXERCISE 14.2 The right tool for each edit

Which editor feature fits each task? (a) Rename a function used in 40 files. (b) Add a comma to the end of 12 consecutive lines. (c) Change `http://` to `https://` in every YAML file in the project. (d) Swap the two arguments of every call to `move(a, b)`.

SOLUTION

(a) **Rename symbol**, F2 in VS Code, which uses the language server to change only real references, as the example showed. (b) **Multiple cursors**: put a cursor on each of the twelve lines, by column selection or by adding cursors line by line, press End, type the comma once. (c) **Search and replace across files**, with the file filter set to `*.yaml,* .yaml`. Review each match before replacing, because some may be URLs that shouldn't change, such as a local server without HTTPS. Commit just before, so `git diff` shows exactly what changed. (d) This one has no perfect tool. A regular-expression replacement such as `move\((\w+), (\w+)\)` to `move($2, $1)` handles simple arguments, but breaks on arguments that are themselves calls with commas inside. Some languages' tools offer a "change signature" refactoring that does it properly. Whichever you use, run the tests afterwards. The general rule is the example's: the more an edit depends on meaning rather than text, the more you want a tool that understands the code.

EXERCISE 14.3 Break the tie

For the query `ord`, the example's scorer ranked `tests/test_order_detail.py` just above `src/shop/models/order.py`. Change the scoring so that a match at the very start of the file name wins, and shorter paths win near-ties.

SOLUTION

```
def better(query, path):
    hit = fuzzy(query, path)                # the example's scorer
    if not hit: return None
    score, at = hit
    if at[0] == path.rfind("/") + 1: score += 10 # the query starts the file name
    return score - 0.1 * len(path), at        # and shorter paths win near-ties
```

With those two changes, `ord` ranks `src/shop/models/order.py` first, then `views/order_detail.py`, then the test file, which is what someone typing `ord` most likely wants. The other queries keep their order: `mdl` still finds `data_loader.py`, and `tmod` the test file. Real editors combine more signals still, such as files you opened recently and files near the one you're in. Ranking is a design decision, not a fact, and the right test is whether the file you wanted appears first, on the queries people really type.

CHECKLIST

- ☐ Can you open any file and jump to any symbol in your editor without the mouse?
- ☐ Can you go to a definition, find all references, and rename a symbol safely?
- ☐ Can you search and replace across a project with a regular expression and a file filter?
- ☐ Can you edit many places at once with multiple cursors?
- ☐ Can you exit Vim, and have you set `EDITOR` to the editor you chose?

GO DEEPER

COURSE The Missing Semester: Editors (Vim) MIT

DOCS VS Code: Basic Editing Microsoft

Error messages, docs and good questions

3 exercises

*Most answers are already in front of you: in the error message you skimmed, the manual page, the `--help` output and the project's documentation. Reading them is a skill. Read tracebacks from the bottom up, decode a synopsis, and search the documentation before the web. When you do need help, a **minimal reproduction** does half the work: making one often reveals the bug, and when it doesn't, it gets your question answered fast. This chapter reads a traceback properly, then shrinks a failing input automatically to the two lines that matter.*

Read the whole message

An error message is a program's best attempt to tell you what went wrong and where, and it's usually right. Read it slowly, all of it, before doing anything else. It names a file, a line, a value, or the thing it expected and what it found instead. When there are many errors, start with the first: compilers and test runners often report one real problem followed by a cascade of consequences. If you search for a message, search for the exact words in quotes, without the parts that are yours, such as your file names, paths and values, which nobody else's copy of the message contains.

Tracebacks, from the bottom

When a Python program fails, it prints a **traceback**: the chain of calls that was in progress, with the outermost first and the innermost, where the error happened, last, followed by a final line naming the exception and its message. So read it from the bottom. The last line says *what* went wrong. The frame just above it says *where*: the file, line number and function, and the line of code itself. The frames above that say *how the program got there*. The bug is usually in the lowest frame that belongs to your own code rather than to a library, because libraries mostly fail when given something they shouldn't have been. A traceback containing "During handling of the above exception, another exception occurred" shows two errors, and the first is usually the cause. Other languages differ in layout: Java, for one, prints the exception first and then the frames, innermost first. Learn your language's convention once.

```

Traceback (most recent call last):
  File "errors.py", line 35, in <module>
    average_prices("\n".join(menu))

  File "errors.py", line 11, in average_prices
    result[category] = average(prices)

  File "errors.py", line 20, in average
    ... round(sum(numbers) / len(numbers), 2)

ZeroDivisionError: division by zero

```

3. how it got there

2. where it happened

1. what went wrong

read upwards

"Most recent call last" means the answer is at the bottom. Read upwards until you reach your own code, and look there first.

Manuals, help and documentation

Every command-line tool documents itself. `man ls` opens the manual page. Its sections are always in the same order: NAME, SYNOPSIS, DESCRIPTION and the options, then often EXAMPLES, FILES, ENVIRONMENT and SEE ALSO. Search inside it with `/`, press `n` for the next match, and `q` to quit. `man -k word` searches the one-line descriptions of every page. Manuals are grouped in numbered sections, so `man 1 crontab` is the command and `man 5 crontab` the file format. Most programs also answer `--help` with a summary, and the community tldr pages give the five commands people actually type. For languages and libraries, read the official documentation, for the version you have. Python's is excellent, and `help(thing)` shows it from inside Python.

A **synopsis** is a compact grammar for a command's arguments. Square brackets mark something optional, a bar separates alternatives, and an ellipsis means "one or more". `cp [-R [-H | -L | -P]] [-f | -i | -n] source_file ... target_directory` says that `-R` is optional, and that `-H`, `-L` or `-P` may follow it. It also says that at most one of `-f`, `-i` and `-n` belongs in a command, and that copying several files requires a folder at the end.

The example takes a short function that averages prices by category from a menu file, and a real menu of about a hundred lines on which it crashes. It reads the traceback from structured data rather than printing it, because the exact layout of tracebacks changes between versions of Python. Then it hands the failing input to **delta debugging**, Andreas Zeller's algorithm for shrinking a failing input automatically. The algorithm repeatedly removes chunks of the input and keeps any smaller version that still fails, until removing anything at all makes the failure go away.

EXAMPLE

errors.py · a function that crashes on a hundred-line menu, its traceback read from the bottom up, the input shrunk by delta debugging to the two lines that matter, and the bug report that makes possible

```

import random, traceback

def average_prices(text):

```

```

    """The average price in each category of a menu, from lines like '[drinks]' and 'tea
    2.50'."""
    result, category, prices = {}, None, []
    for line in text.splitlines():
        line = line.strip()
        if not line or line.startswith("#"): continue
        if line.startswith("["):
            if category is not None:
                result[category] = average(prices)
                category, prices = line.strip("[]"), []
            else:
                name, price = line.rsplit(" ", 1)
                prices.append(float(price))
        if category is not None and prices:
            result[category] = average(prices)
    return result

def average(numbers): return round(sum(numbers) / len(numbers), 2)

def make_menu():
    """A menu of about a hundred lines; one category, part-way down, has nothing in it yet."""
    rng = random.Random(15); lines = ["# The Corner Cafe menu"]
    for category in ["breakfast", "soups", "salads", "mains", "specials", "sides", "cakes",
                    "ice cream", "tea", "coffee", "juice", "wine"]:
        lines += ["", f"[{category}]"]
        if category == "specials": lines.append("# coming soon"); continue
        lines += [f"{category} item {i} {rng.randint(150, 1800) / 100:.2f}" for i in range(1,
        8)]
    return lines + ["# prices include service"]

menu = make_menu()
print(f"1. The program fails on the real menu, {len(menu)} lines long")
try:
    average_prices("\n".join(menu))
except Exception as error:
    frames = traceback.extract_tb(error.__traceback__)
    print(f"    the last line of the traceback says what went wrong: {type(error).__name__}:
    {error}")
    print("    the lines above it say where; from the bottom up, the innermost call first:")
    for f in reversed(frames):
        name = f.name if f.name.startswith("<") else f.name + "()"
        print(f"        line {f.lineno:2} in {name:16} {f.line}")

def fails(lines):
    try: average_prices("\n".join(lines)); return False
    except ZeroDivisionError: return True

def shrink(lines, fails):
    """Delta debugging (Zeller, 2002): keep dropping chunks of the input while it still
    fails."""
    n, tests, sizes = 2, 0, [len(lines)]
    while len(lines) >= 2:
        size = -(-len(lines) // n)
        chunks = [lines[i:i + size] for i in range(0, len(lines), size)]
        for i in range(len(chunks)):
            rest = [line for j, c in enumerate(chunks) if j != i for line in c]
            tests += 1
            if fails(rest):
                lines, n = rest, max(n - 1, 2); sizes.append(len(lines)); break
        else:
            if n >= len(lines): break
            n = min(2 * n, len(lines))
    return lines, tests, sizes

print("2. Shrinking the input automatically, keeping only what's needed to fail")
small, tests, sizes = shrink(menu, fails)

```

```

print(f"    {tests} test runs; the failing input went from {' -> '.join(map(str, sizes))}
      lines:")
for line in small: print(f"        {line}")
print(f"    with the first line alone it works: {not fails(small[:1])}; "
      f"with the second alone: {not fails(small[1:])}")

print("3. The bug report this makes possible")
print(f"""    average_prices() raises ZeroDivisionError when a category has no items and another
      follows.
To reproduce: average_prices("{small[0]}\n{small[1]}")
Expected:     {} or {'{small[0].strip('[]')}: None}, not a crash
Actually:     ZeroDivisionError: division by zero, in average(), called from line 11
Tried:        the same with an empty category at the end of the menu works""")

```

OUTPUT

1. The program fails on the real menu, 104 lines long
the last line of the traceback says what went wrong: `ZeroDivisionError: division by zero`
the lines above it say where; from the bottom up, the innermost call first:

```

line 20 in average()      def average(numbers): return round(sum(numbers) /
len(numbers), 2)
line 11 in average_prices() result[category] = average(prices)
line 35 in <module>       average_prices("\n".join(menu))

```
2. Shrinking the input automatically, keeping only what's needed to fail
30 test runs; the failing input went from 104 -> 52 -> 26 -> 19 -> 14 -> 10 -> 8 -> 6 -> 4 -> 3 -> 2 lines:

```

[specials]
[sides]

```

with the first line alone it works: `True`; with the second alone: `True`
3. The bug report this makes possible
`average_prices()` raises `ZeroDivisionError` when a category has no items and another follows.
To reproduce: `average_prices("[specials]\n[sides]")`
Expected: `{}` or `{'specials': None}`, not a crash
Actually: `ZeroDivisionError: division by zero, in average(), called from line 11`
Tried: the same with an empty category at the end of the menu works

What the example shows

The traceback pointed straight at the cause. Its last line said `ZeroDivisionError: division by zero`; the frame above, line 20 in `average()`, showed the division by `len(numbers)`, so the list was empty. The frame above that, line 11 in `average_prices()`, showed when `average` is called: when a new category starts and the previous one is finished. Together they say "a category had no items". Reading only the first line of the traceback would have told you nothing.

Shrinking found the real conditions. Delta debugging needed 30 runs to cut 104 lines down to two, `[specials]` and `[sides]`. Neither line fails alone: an empty category followed by another category is the bug, while an empty category at the very end is handled correctly. That's a fact you'd likely never have found by staring at the hundred-line menu, and it's exactly what the fix needs. The bug report in section 3 says in five lines what a confused paragraph and an attached file would otherwise have left the reader to work out. You can shrink by hand the same way, deleting halves of the input or the code and keeping the failure, as exercise 15.3 does.

Asking a good question

When you've read the message and the documentation and you're still stuck, ask, and make the question easy to answer. Give a **minimal, reproducible example**: the smallest complete program and input that show the problem, which someone else can run and see fail. Say what you expected and what happened instead, with the exact error copied as text, never a screenshot. Name the versions of everything involved, and list what you've already tried and ruled out. Making that example is itself a debugging technique: in shrinking the problem, you'll often find it. So will explaining it out loud to anyone, or anything, which is why programmers talk of rubber-duck debugging. The same rules apply when the one you ask is an AI assistant. Give it the minimal example and the exact error, and then check its answer, because you have to be able to explain every line you ship.

COMMON TRAPS

Reading the first line of a traceback instead of the last.

Searching for an error with your own paths and names in it, which no one else's copy contains.

Screenshots of code or errors, which can't be searched, copied or run.

"It doesn't work", without what you expected, what happened, and what you tried.

Asking before reading, when the answer is in the message or the manual.

EXERCISES

EXERCISE 15.1 Read a synopsis

A Mac's `cp` manual begins `cp [-R [-H | -L | -P]] [-f | -i | -n] [-aclNpSsvXx] source_file target_file`. According to it: (a) can you write `cp -L a b`? (b) `cp -f -n a b`? (c) `cp -R` notes backup? (d) What do you need to copy three files at once?

SOLUTION

(a) Not on this line: `-L` appears only inside the brackets that begin with `-R`, so it's meant for recursive copies, where it says to follow symbolic links. (b) No: `[-f | -i | -n]` allows at most one of the three. (c) Yes: `-R` is optional, and it's the option for copying folders. (d) The page's second form, `source_file ... target_directory`: any number of files, then an existing folder. One more lesson comes from trying them. The Mac's `cp` accepted both `cp -H a b` and `cp -f -n a b` without complaint. Its description explains why: `-H`, `-L` and `-P` are ignored unless `-R` is given, and when several of `-f`, `-i` and `-n`

are given, the last one wins. A synopsis states the intended grammar, and tools are often more forgiving than it. So read the description too, and never rely on undocumented leniency in a script.

EXERCISE 15.2 Rewrite the question

Rewrite this as a question someone could answer: "My Python script doesn't work!! I get an error about division. Attached is a screenshot and my menu file. Please help ASAP."

SOLUTION

```
average_prices() raises ZeroDivisionError for an empty category followed by another
```

With Python 3.13, this function, from our menu tool, crashes on a menu where a category has no items yet:

```
average_prices("[specials]\n[sides]")
```

Expected: no entry for "specials" (or None), and no crash.

Actual: ZeroDivisionError: division by zero, in average(), called from average_prices() at the line that closes the previous category.

An empty category at the end of the menu works. I think the final check "if category is not None and prices" is missing where a new category starts, but I'm not sure whether skipping the category or recording None is the expected behaviour. Which do the callers rely on?

The title states the problem in a line, so the right person recognises it. The two-line reproduction replaces the screenshot and the whole menu file, and anyone can paste it and see the error. Expected and actual are stated separately, with the exact message as text. The version is given. And the last paragraph shows what you've worked out and asks the one thing you can't decide alone, which is the question that gets answered. "ASAP" is gone: urgency is yours, and a good question is the fastest way to an answer.

EXERCISE 15.3 Shrink it by hand

A 40-line script that loads a configuration file and processes some records fails with `KeyError: 'region'`, deep inside a library call. Describe how to cut it down to a minimal reproduction.

SOLUTION

Work like delta debugging, by hand, and run the script after every change to make sure it still fails the same way. First make the failure independent of everything outside the script: replace the configuration

file and the records with literal values copied into the code, so that it fails on any machine. Then delete half of what's left, the code or the data, and run it. If it still fails, keep the deletion; if not, put it back and try the other half, or a smaller piece. Replace each function call you don't suspect by the value it returns. Stop when every remaining line is needed for the failure. Usually you finish with three or four lines and a single record, often one with no `region` field. By then you can see whether the library expects the field, which its documentation will say, or your code should have supplied it. Commit the starting point first, so you can always get back to it.

CHECKLIST

- ☐ Can you read a traceback from the bottom, and find the frame to look at first?
- ☐ Can you find an answer in a manual page, using its sections and its search?
- ☐ Can you decode a synopsis's brackets, bars and ellipses?
- ☐ Can you shrink a failing program or input to a minimal reproduction?
- ☐ Can you write a question with a reproduction, expected and actual results, and what you tried?

GO DEEPER

GUIDE [How to create a Minimal, Reproducible Example Stack Overflow](#)

BOOK [The Debugging Book: Reducing Failure-Inducing Inputs Andreas Zeller](#)

ESSAY [How to Report Bugs Effectively Simon Tatham](#)

How to learn

3 exercises

How you study matters as much as how long. Rereading and highlighting feel productive and produce little. Decades of experiments point to three things that work far better. Testing yourself beats reviewing. Spreading reviews out over days and weeks beats massing them. And practising a skill at the edge of your ability, with feedback, beats repeating what you can already do. This chapter explains why, uses a small model of forgetting to show what spacing buys, and builds the scheduler flashcard programs use.

Why rereading fails

Reading a chapter a second time feels like learning, because everything is familiar: you recognise each idea as it comes. But recognising is not recalling. The test that matters is whether you can produce the idea when you need it, with the book closed, and familiarity says little about that. Psychologists call the mistake an illusion of fluency, and it's why students who reread and highlight are often confident and wrong about what they know. The remedy is to make study feel more like the test. That's harder, and the difficulty is the point: the effort of retrieving something is what strengthens it.

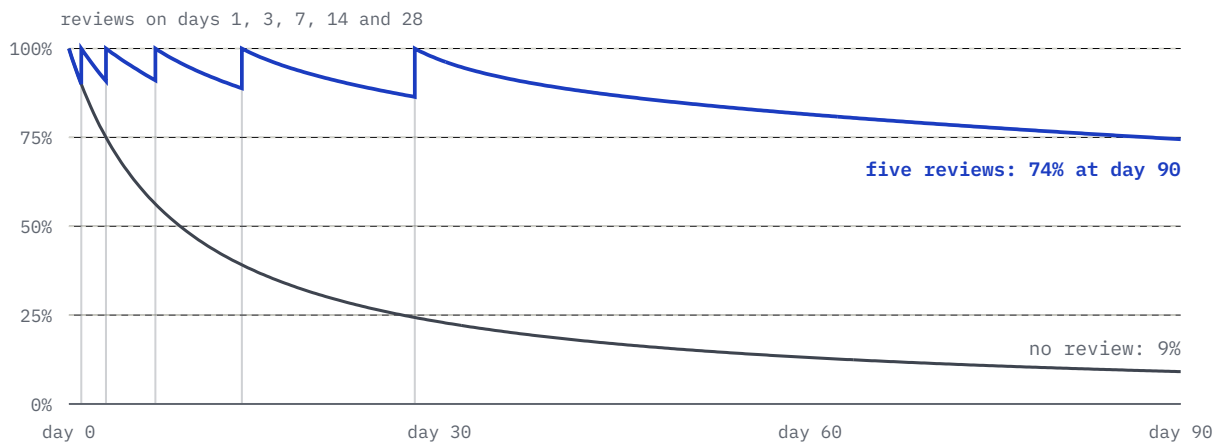
Retrieval practice

In a well-known experiment by Henry Roediger and Jeffrey Karpicke, published in 2006, students studied short prose passages. Some reread a passage several times; others read it once and then tried to write down what they remembered. Five minutes later, the rereaders remembered slightly more. Two days and a week later, the students who had tested themselves remembered much more. Later experiments found retrieval practice beating rereading, note-taking and concept mapping across subjects and ages. The practical forms are simple. Close the book and write what you remember, then check. Use flashcards that ask a question. Explain an idea aloud as if teaching it. And, throughout this volume, predict what a command will print before you run it, which is retrieval with immediate feedback.

Spacing

In the 1880s, Hermann Ebbinghaus learned lists of nonsense syllables and measured how much he'd forgotten after various delays. The result was the first **forgetting curve**: steep at first, then flattening. He also found that reviewing a list again, after a delay, made the next curve shallower. A century of later work confirmed it as the **spacing effect**. A 2006 review by Nicholas Cepeda and colleagues, pooling hundreds of experiments, found that the same total study time, spread over separate sessions, is remembered far better than when it's

massed into one. The best gap grows with how long you want to remember. To keep something for a year, review it at intervals of weeks, not minutes. Cramming works, after a fashion, for tomorrow's test, and leaves almost nothing a month later.



Each review resets the curve and flattens it. Five reviews in the first month leave most of what was learned still recallable two months later; none leaves almost nothing.

The example turns those findings into a small model. Its forgetting curve has the shape modern schedulers use, and a review strengthens a memory more the closer it was to being forgotten. It compares five plans for 200 facts learned on the same day, each reviewed only during the first month, and then follows one fact under the scheduling rule that flashcard programs use. Its numbers come from invented constants, so read them for their shape, not as measurements.

EXAMPLE

learning.py · a small model of forgetting, used to compare no review, cramming, fixed spacing, Leitner boxes and a 90 percent rule, then one fact's growing gaps between reviews

```
import random
from dataclasses import dataclass

# A small model of memory, of the kind spaced-repetition programs are built on. Its constants
# are
# chosen to show the shape of the effect; the experiments described in the chapter are the
# evidence.
# - Each fact has a stability S: the number of days until your chance of recalling it falls
# to 90%.
#   t days after you last saw it, the chance is 1 / (1 + t / (9 * S)). A new fact has S = 1.
# - Recalling it makes S larger, by more the nearer you were to forgetting: S * (1 + 15 * (1
# - chance)).
#   A review of something you can recall easily adds almost nothing.
# - Failing to recall it sets S back to 1: you look up the answer and start again.
def chance(days, stability): return 1 / (1 + days / (9 * stability))

@dataclass
class Fact:
    S: float = 1.0
```

```

seen: float = 0.0
reviews: int = 0
box: int = 1

def review(self, day, rng):
    p = chance(day - self.seen, self.S)
    recalled = rng.random() < p
    self.S = self.S * (1 + 15 * (1 - p)) if recalled else 1.0
    self.seen, self.reviews = day, self.reviews + 1
    return recalled

def cram(fact, rng):
    # four more passes through the cards,
    that evening
    for k in range(1, 5): fact.review(k / 48, rng)

def fixed(fact, rng):
    # the same five days for every fact
    for day in (1, 3, 7, 14, 28): fact.review(day, rng)

def leitner(fact, rng):
    """Leitner boxes: a fact in box k comes back 2**(k-1) days later; recalling it moves it up
    a box,
    forgetting it sends it back to box 1. Hard facts are seen often, easy ones rarely."""
    day = 1
    while day <= 30:
        fact.box = min(fact.box + 1, 5) if fact.review(day, rng) else 1
        day += 2 ** (fact.box - 1)

def at_ninety(fact, rng):
    """What modern schedulers do: bring each fact back just as its predicted recall falls to
    90%."""
    day = fact.seen + fact.S
    while day <= 30:
        fact.review(day, rng); day = fact.seen + fact.S

print("200 facts learned on day 0, then reviewed only during the first 30 days")
print(f"    {'plan':38}{'reviews per fact':>17}{'recall, day 60':>16}{'recall, day 90':>16}")
for label, plan in (("no review", None), ("cram: 4 more passes on day 0", cram),
                    ("fixed: days 1, 3, 7, 14 and 28", fixed), ("Leitner boxes", leitner),
                    ("each fact when recall reaches 90%", at_ninety)):
    rng = random.Random(16); facts = [Fact() for _ in range(200)]
    for f in facts:
        if plan: plan(f, rng)
    reviews = sum(f.reviews for f in facts) / len(facts)
    later = [sum(chance(day - f.seen, f.S) for f in facts) / len(facts) for day in (60, 90)]
    print(f"    {label:38}{reviews:>17.1f}{later[0]:>16.0%}{later[1]:>16.0%}")

print("one fact under the 90% rule: the gaps between reviews grow as it becomes stable")
f, rng = Fact(), random.Random(3); days = []
while f.seen + f.S <= 365:
    f.review(f.seen + f.S, rng); days.append(round(f.seen))
print("    reviewed on days " + ", ".join(map(str, days)) + f"; by then it would last {f.S:.0f}
    days")

```

OUTPUT

```

200 facts learned on day 0, then reviewed only during the first 30 days
plan                                reviews per fact  recall, day 60  recall, day 90
no review                          0.0              13%            9%
cram: 4 more passes on day 0        4.0              15%            10%
fixed: days 1, 3, 7, 14 and 28      5.0              81%            74%
Leitner boxes                       4.9              83%            74%
each fact when recall reaches 90%   4.8              86%            78%
one fact under the 90% rule: the gaps between reviews grow as it becomes stable
reviewed on days 1, 4, 10, 25, 64, 162; by then it would last 244 days

```

What the example shows

Cramming adds almost nothing. Four more passes on the evening of learning raised recall two months later from 13% to 15%. In the model, as in life, reviewing what you can recall easily barely strengthens it, because there's no effort of retrieval. The same number of reviews spread across the month, on days 1, 3, 7, 14 and 28, raised it to 81%.

Scheduling by need does slightly better, with less work. Leitner boxes, Sebastian Leitner's 1970s system of moving cards between boxes reviewed at doubling intervals, gave 83% from 4.9 reviews per fact. Reviewing each fact just as its predicted recall falls to 90%, which is what modern schedulers do, gave 86% from 4.8. They spend reviews where they're needed, often on facts about to be forgotten and rarely on secure ones. Under that rule, one fact's gaps between reviews grew from one day to three, six, fifteen, thirty-nine and ninety-eight: six reviews in half a year left it stable for the best part of a year. This is the real promise of spaced repetition: remembering something for years costs a few minutes in total.

Flashcards that work

A spaced-repetition program does the scheduling for you. **Anki** is free, open source and runs everywhere, and since 2023 includes the FSRS scheduler, which fits a memory model to your own review history and targets 90% recall by default, the rule the example used. The program is the easy part; the cards decide whether it works. Make each card ask one thing, as a question you have to answer from memory, in your own words. Ask why and when, not just what: "Why must `cd` be a shell builtin?" teaches more than "What is `cd`?". Use cloze deletions, blanks in a sentence, for syntax, such as "`find . -name '*.py' -_____rm {} +`". Add cards as you learn, review every day, about fifteen minutes, and delete cards that annoy you. For this stage, good material is anything you've looked up twice: options, Git concepts, regular-expression syntax, exit statuses.

Deliberate practice

Facts are only part of it; most of this stage is skills, and skills are learned by practice of a particular kind. Anders Ericsson's studies of expert performers, beginning with violinists in 1993, distinguished **deliberate practice** from doing the activity. It has a specific goal just beyond what you can do now, immediate feedback, full attention, and repetition with refinement. Playing through pieces you already know isn't it, and neither is typing commands you already know. Learn Git Branching is deliberate practice for Git: each level is a precise goal, the graph shows you instantly whether you reached it, and you can repeat a level until it's fluent. So is solving an exercise in this volume a second time, a week later, from memory.

Two more findings are worth knowing. **Interleaving**, mixing different kinds of problems in one session instead of practising one kind at a time, feels worse and works better, because each problem makes you first decide which tool it needs. Doug Rohrer and Kelli Taylor showed it with mathematics problems in 2007. And Robert Bjork's **desirable difficulties**

sums up the chapter: the conditions that make practice feel harder, testing, spacing, interleaving, are the ones that make learning last. If studying feels easy, check that you're not just rereading.

COMMON TRAPS

Rereading and highlighting as the main way of studying.

Cramming, which serves tomorrow and forgets next month.

Watching tutorials without doing the exercises, which builds recognition, not skill.

Flashcards that are too big, "Explain Git", or that only ask you to recognise an answer.

Practising only what you're already good at, because it feels like progress.

EXERCISES

EXERCISE 16.1 Ten cards

Write ten flashcards for chapters 1 to 3 of this volume, and say what makes each a good card.

SOLUTION

```
Q: What does a program receive from the line showargs $f when f="my notes.txt"?
A: Two arguments, "my" and "notes.txt": the unquoted expansion is split.
Q: Why does sort data.txt > data.txt leave an empty file?
A: The shell truncates data.txt for the redirection before sort starts reading it.
Q: In what order are these applied: cmd > out.txt 2>&1 ? Where does stderr go?
A: Left to right: 1 to out.txt, then 2 to where 1 points: both go to out.txt.
Q: What's a pipeline's exit status by default, and what changes it?
A: The last command's; set -o pipefail makes it fail if any command failed.
Q: What does exit status 127 mean? And 126? And 130?
A: Command not found; found but not executable; killed by SIGINT (128 + 2).
Q: Why does uniq need sorted input?
A: It only merges duplicate lines that are adjacent.
Q: Fill in: find . -name '*.tmp' -_____ | xargs -0 rm
A: print0
Q: In a regular expression, what does . match? How do you match a real dot?
A: Any one character; \. (or grep -F for plain text).
Q: In awk, what are $1, $NF and NR?
A: The first field, the last field, and the current line's number.
Q: When does set -e not stop a script after a failing command?
A: When the command's status is being tested: in an if, a while, or beside && or ||.
```

Each card asks one thing, and most ask why or when, not just what, so that recalling the answer exercises understanding. They're questions you must answer from memory, not statements to recognise. The answers are short enough to check in a second. The fill-in card trains a detail that's easy to forget and expensive to get wrong. Write your own cards in your own words, rather than copying these: making the card is the first act of retrieval, and you'll remember your phrasing better than anyone else's.

EXERCISE 16.2 A week of deliberate practice

Plan a week of deliberate practice to make Part III's Git operations fluent, at about thirty minutes a day.

SOLUTION

Day 1: Learn Git Branching's introduction levels, then repeat any level you needed hints for until you can do it without them. Day 2: in a scratch repository, recreate from memory the graph in chapter 10's example, a branch, a detached commit, a rescue from the reflog, checking with `git log --graph --all` after each step. Day 3: the "moving work around" levels, rebase and cherry-pick, and then make a conflict on purpose and resolve it twice, once by merging and once by rebasing. Day 4: interleave, with a mixed set of five small tasks, such as undoing a pushed commit, splitting a commit, recovering a deleted branch, bisecting a planted bug and fixing a commit message. Choosing the right command for each is half the exercise. Day 5: time yourself on day 4's set, then do it again without looking anything up. Each session has a specific goal, immediate feedback from the graph or the test, and a difficulty just past where you are. When a task becomes easy, replace it with a harder one. And put the facts you had to look up on flashcards.

EXERCISE 16.3 Earlier or later?

The example reviewed each fact when its predicted recall fell to 90%. Predict what happens to the number of reviews, and to recall at day 90, if the rule is 97% instead, or 80%.

SOLUTION

review when recall falls to per review	reviews per fact	recall at day 90	strength gained
97%	11.6	83%	x1.45
90%	4.8	78%	x2.50
80%	2.8	65%	x4.00

Reviewing earlier, at 97%, gives somewhat better recall, but at more than twice the work, because a review of something you recall easily strengthens it only a little. Reviewing later, at 80%, saves reviews, because each one strengthens more, but more facts are forgotten in between, and a forgotten fact starts again from one day. The figures come from running the example's model with each rule, so they show the trade-off rather than predict your memory. Real schedulers face the same trade-off, which is why Anki's FSRS lets you choose the recall you want and defaults to 90%, a reasonable balance between effort and forgetting.

CHECKLIST

- ☐ Can you explain why rereading feels effective and isn't?
- ☐ Can you use retrieval practice in your own study: closed-book recall, questions, prediction?
- ☐ Can you explain the spacing effect, and why cramming fails a month later?
- ☐ Can you write good flashcards, and keep a daily review habit?
- ☐ Can you design deliberate practice for a skill, with a goal, feedback and rising difficulty?

GO DEEPER

BOOK [Make It Stick: The Science of Successful Learning](#) Brown, Roediger & McDaniel

ESSAY [Augmenting Long-term Memory](#) Michael Nielsen

DOCS [Anki Manual: Background](#) Ankitects

The stage asks for two things, both public and both with a README written for a stranger: a repository of your dotfiles, and one shell script that automates a chore you genuinely repeat. Chapters 8 and 4 build a small working version of each; the lists below say what the full versions need. Both will outlive the stage. You'll install the first on every machine you use, and the second is the start of a habit of automating what bores you.

1. A public dotfiles repository

Your settings for the tools you use every day, in a repository anyone can read, with a script that installs them. Chapter 8 is the model.

- 01 Settings for your shell, Git and editor, grown a line at a time, each with a comment saying why it's there.
- 02 An install script that links every file into place, backs up anything it would replace, never deletes, and is safe to run again, plus an uninstall that reverses it.
- 03 Per-machine and private settings in local files that the shared ones include, ignored by Git, and no secret anywhere in the history: check with `git grep` or a scanner before you publish.
- 04 Tested on a clean account or a fresh container, with anything specific to macOS or Linux behind a check of `uname`.
- 05 A README that says what each file configures, gives the one command that installs it, says what that command changes and how to undo it, and lists the systems you've tried it on.

2. A script for a chore you repeat

Pick something you really do by hand every week: tidying downloads, backing up a folder, renaming photos, cleaning old branches, starting your day's tools. Chapter 4 is the model.

- 01 Strict mode, a usage message for `-h` and for wrong arguments, options parsed with `getopts`, exit statuses 0, 1 and 2, and every message on stderr.
- 02 A dry run that shows exactly what would happen: nothing overwritten or

...any run that ensures no safety issue could happen, meaning no file is deleted without a check; safe to run twice; and a lock and a trap if two runs could ever overlap.

- 03 Every expansion quoted, no warnings from ShellCheck, and either bash 3.2 compatible or a clear statement of the version it needs.
 - 04 Tests that run it in a temporary folder and check the results, including what happens with bad arguments, a missing folder and names with spaces.
 - 05 Installed as a command through your dotfiles, documented in a README, and if the chore is regular, run on a schedule with `cron` on Linux or `launchd` on a Mac.
-

For both: keep them under Git from the first day, with small commits and messages that explain why, as chapter 12 describes. Make at least one change through a pull request, from a branch, even in your own repository, so chapter 13's workflow is something you've done rather than read about. Then ask someone else to install your dotfiles and run your script from the README alone, and fix whatever confused them.

THE SIXTEEN RULES

One per chapter, each linked back to it. Read them before you start a new script or a new repository, and again when a tool surprises you.

-
- 01 Quote every expansion, and know exactly which list of arguments a line becomes.
-
- 02 Results to stdout, messages to stderr, and check the status of every step, including inside a pipeline.
-
- 03 Ask questions of text with small tools in a pipeline: quote the patterns, and sort before you count.
-
- 04 Strict mode, a dry run, and never overwrite: a script must be safe to run twice.
-
- 05 A name points to an inode, and a symbolic link points to a name.
-
- 06 The folder decides who may delete; never fix a permission error with `777` or `sudo .`
-
- 07 The environment flows down, never up; ask `type -a` what a name will run.
-
- 08 Keep your settings in a public repository, linked into place, and your secrets out of it.
-
- 09 A commit is a snapshot named by its contents; changing the past renames everything after it.
-
- 10 Branches are labels and the reflog remembers them; only uncommitted work can really be lost.
-
- 11 Merge shared work; rebase only commits nobody else has.
-
- 12 Make small commits that explain why, and bisect whatever used to work.
-
- 13 Work on branches, push with a lease, and keep pull requests small and explained.

-
- 14 Learn one editor deeply, and rename with a tool that understands the code.
-
- 15 Read the whole message, from the bottom, and ask with a minimal reproduction.
-
- 16 Test yourself, space your reviews, and practise at the edge of your ability.
-

READING LIST

The courses, books and documentation behind this volume, and the chapters each one deepens. If you do only one thing, work through The Missing Semester with its exercises. For Git, read Pro Git's first three chapters and its chapter on internals, and play Learn Git Branching until rebasing is boring.

WORK	AUTHOR	DEEPENS
The Missing Semester of Your CS Education	Athalye, Gjengset & Gonzalez Ortiz, MIT	Ch. 1–4, 7, 14
Pro Git	Scott Chacon & Ben Straub	Ch. 9–13
Learn Git Branching	Peter Cottle	Ch. 10–11, 16
The Linux Command Line	William Shotts	Ch. 1–7
The Bash Guide and Bash Pitfalls	Greg's Wiki	Ch. 1–4
The AWK Programming Language, second edition	Aho, Kernighan & Weinberger	Ch. 3
ShellCheck	Vidar Holen	Ch. 4
Git from the Bottom Up	John Wiegley	Ch. 9–10
Practical Vim	Drew Neil	Ch. 14
How to Report Bugs Effectively	Simon Tatham	Ch. 15
The Debugging Book	Andreas Zeller	Ch. 12, 15
Make It Stick	Brown, Roediger & McDaniel	Ch. 16
Augmenting Long-term Memory	Michael Nielsen	Ch. 16

Compiled October 2026 as the companion to stage 00, Tools, the shell, and how to learn, of The Engineer's Syllabus, alongside *The Engineer's Primer, Workbench, Compass, Mill, Loom, Exchange, Press, Ledger, Craft, Shipyard, Fleet, Watch, Citadel* and *Foundry*. It follows the order of topics in the stage, and draws on the works in the reading list and on the original sources for its named results. These are Ebbinghaus (1885) on forgetting; Leitner (1972) on boxes of flashcards; McIlroy (1986) on the six-command pipeline; Ericsson, Krampe and Tesch-Römer (1993) on deliberate practice; Zeller and Hildebrandt (2002) on delta debugging; Cepeda and colleagues (2006) on spacing; Roediger and Karpicke (2006) on retrieval; Rohrer and Taylor (2007) on interleaving; and Stevens and colleagues (2017) on the first SHA-1 collision. Tools are described as of late 2026.

Every example was executed before publication with Python 3.14 on macOS 26, whose shell tools are bash 3.2, BSD's versions of the text tools, and Git 2.50. They're written for Python 3.12 or later and to print the same output on Linux with the GNU tools: they sort whatever the filesystem would order arbitrarily, avoid output whose spacing differs between systems, give every commit a fixed author and time, ignore your own Git settings, and use only features shared by bash 3.2 and bash 5. They were not run on Linux for this edition, so a difference there is a bug worth reporting. Each was also run a second time in a deliberately hostile environment, with a different time zone and temporary folder, a home folder with its own Git and shell settings, and `BASH_ENV` and `CDPATH` set, and printed exactly the same. Tested here: how bash turns a line into arguments; redirections, pipes, exit statuses and `set -e`; questions answered from a web server's log; a tidying script with a dry run, a lock and a trap; hard links, symbolic links and a deleted file still open; permission bits tried one at a time; `PATH`, exported variables and bash's startup files; a dotfiles repository installed twice; Git's objects built by hand; refs, a detached `HEAD` and the reflog; a clean merge, a conflict and a rebase; a bug found by `bisect`, `log -S` and `blame`; a contribution through a fork; a fuzzy finder and a scope-aware rename; a traceback read and an input shrunk by delta debugging; and a model of forgetting.