

FROM ZERO TO CONFIDENT

Linux Command Line for Developers

Navigate, Automate, and Run Servers

Files, text, permissions, processes, SSH, bash scripting, and a real server — explained step by step

BASH · GREP · AWK · SSH · SYSTEMD · CRON · RSYNC

THENEXBOOKS

Linux Command Line for Developers

From Zero to Confident
Navigate, Automate, and Run Real Servers from the Terminal

First Edition

THENEXBOOKS

Linux Command Line for Developers
From Zero to Confident

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code and script examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Commands that change system configuration, delete files, or affect remote servers should be tested in a safe environment first. Distributions and tools evolve; always consult the manual pages and official documentation for the versions you use.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Ubuntu, Debian, Fedora, Red Hat, macOS, Windows, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Emberline, the running example in this book, is fictional. Any resemblance to real organizations or services is coincidental. Host names and IP addresses in examples use reserved documentation ranges.

Preface

Sooner or later, every developer ends up in a terminal. A build breaks on a CI runner, a production server runs out of disk space, a log file holds the one line that explains a bug, a container needs a quick look inside, or a repetitive task begs to be automated. Graphical tools help, but the command line is where developers get closest to how their systems actually work — and where a few well-chosen commands can replace an hour of clicking.

Yet many developers only ever learn the terminal in fragments: a copied command here, a half-remembered option there, and a nagging fear of typing something that cannot be undone. That fear is reasonable. The shell is powerful, terse, and unforgiving. But it is also logical, consistent, and learnable, and once its core ideas click — files and paths, streams and pipes, permissions, processes, and a small number of composable tools — it becomes one of the most productive places you can work.

This book takes you from zero to confident. We start with the very first commands and build up, step by step, to text processing, permissions, processes, networking, SSH, shell scripting, automation, and finally running a real application on a Linux server. Along the way we explain what the system is doing and how to avoid the mistakes that hurt on real machines.

Who This Book Is For

- **Developers new to Linux** who work on Windows or macOS and need to become comfortable in a Linux shell, WSL, containers, or cloud servers.
- **Developers who know a few commands** and want to understand the system well enough to work quickly and safely.
- **Back-end, DevOps-curious, and full-stack developers** who deploy, debug, and automate on Linux servers.
- **Students and career changers** who want a solid, practical foundation for professional work.

What You Will Learn

- How to navigate the file system and manage files, directories, and links with confidence.
- How to find, filter, and transform text and data with pipes, grep, sort, sed, awk, and jq.
- How users, permissions, processes, packages, services, logs, and disks work on a Linux system.
- Networking from the terminal: diagnosing connections, testing APIs with curl, SSH, and syncing files with rsync.
- Bash scripting from fundamentals to robust, production-quality scripts, plus scheduling with cron and systemd timers.
- Customizing your shell, daily development workflows, security, monitoring, troubleshooting, and a complete server case study.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up your environment and teaches navigation, file management, and how to get help. **Part II — Working with Text and Data** (Chapters 6–10) covers viewing files, pipes and redirection, searching, and the classic text-processing tools. **Part III — Understanding the System** (Chapters 11–15) explains permissions, processes, packages, services, logs, and storage. **Part IV — Networking and Remote Work** (Chapters 16–20) covers networking, HTTP, SSH, file transfer, and terminal productivity. **Part V — Shell Scripting and Automation** (Chapters 21–25) teaches bash scripting, robust scripts, scheduling, and shell customization. **Part VI — Developer Workflows and Operations** (Chapters 26–28) brings it all together in daily development, operations, and a complete case study. The appendices provide a command cheat sheet, script templates, a server checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Emberline

Throughout the book we work on **Emberline**, a fictional community radio and podcast platform run by a small team of developers. Emberline has a web API and front end, audio uploads that are transcoded with ffmpeg, a PostgreSQL database, log files, nightly backups, and scheduled jobs that publish podcast feeds. The team develops on laptops and runs everything on a Linux server called `emberline-prod`. Every chapter's commands have a realistic job to do there — from searching logs to recovering a full disk.

Conventions

Commands are shown after a `$` prompt, followed by their output; long output is trimmed with `...`. Examples assume a recent Ubuntu LTS release and GNU tools; where macOS or other distributions differ, we say so.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A command or mistake that can lose data or break a system.

Contents

Preface

PART I — GETTING STARTED

1 Why the Command Line: Shells, Terminals, and Linux Distributions

2 Setting Up Your Environment: Linux, WSL, macOS, Virtual Machines, and Containers

3 Navigating the File System: pwd, cd, ls, and the Directory Tree

4 Working with Files and Directories: cp, mv, rm, mkdir, touch, and Links

5 Getting Help: man, --help, info, tldr, and Reading Documentation

PART II — WORKING WITH TEXT AND DATA

6 Viewing Files: cat, less, head, tail, and Following Logs

7 Pipes, Redirection, and Standard Streams

8 Finding Things: find, locate, grep, and ripgrep

9 The Text Processing Toolkit: cut, sort, uniq, wc, tr, paste, join, and xargs

10 sed, awk, and jq: Transforming Text and JSON

PART III — UNDERSTANDING THE SYSTEM

11 Users, Groups, Permissions, and sudo

12 Processes and Jobs: ps, top, kill, Signals, and Background Work

13 Installing Software: apt, dnf, Snap, Flatpak, and Building from Source

14 Services and Logs with systemd and journalctl

15 Disks, File Systems, Archives, and Compression

PART IV — NETWORKING AND REMOTE WORK

16 Networking Basics: ip, ping, ss, dig, and Troubleshooting Connections

17 HTTP from the Terminal: curl, wget, and Testing APIs

18 SSH: Keys, Config, Agents, Tunnels, and Secure Remote Access

19 Moving and Syncing Files: scp, sftp, rsync, and Backups

20 Terminal Productivity: History, Aliases, Keybindings, and tmux

PART V — SHELL SCRIPTING AND AUTOMATION

21 Bash Fundamentals: Variables, Quoting, Expansion, and Exit Codes

22 Control Flow, Functions, and Arrays in Bash

23 Writing Robust Scripts: Strict Mode, Arguments, Traps, and ShellCheck

24 Automation: cron, systemd Timers, and Scheduled Jobs

25 Customizing Your Shell: Dotfiles, Prompts, PATH, and Environment Variables

PART VI — DEVELOPER WORKFLOWS AND OPERATIONS

26 The Command Line in Daily Development: Editors, Git, Build Tools, and Containers

27 Security, Monitoring, and Troubleshooting a Linux Server

28 Case Study: Running Emberline on a Linux Server from Zero

APPENDICES

A Command Cheat Sheet

B Script Templates and Reference

C Server Checklist

D Glossary

PART I

Getting Started

What the command line is, how to get a Linux shell on any computer, how to move around the file system, manage files safely, and find answers when you are stuck.

1 · Why the Command Line
2 · Setting Up Your Environment
3 · Navigating the File System
4 · Working with Files and Directories
5 · Getting Help

CHAPTER 1 · PART I — GETTING STARTED

Why the Command Line: Shells, Terminals, and Linux Distributions

Before you type a single command, it helps to know what you are talking to. This chapter explains why the command line still matters to developers, untangles the words terminal, shell, and console, tours the Linux family of distributions, and walks you through your very first commands.

Why Developers Still Need the Command Line

Modern development tools are remarkably friendly. Editors have buttons for running tests, cloud dashboards have forms for deploying services, and version-control clients show your history as a colorful graph. So it is fair to ask: why learn a text interface that looks, at first glance, like something from decades ago?

The honest answer is that the command line is not a legacy interface that graphical tools replaced. It is the layer that most of those graphical tools are built on, and it is the interface you are left with when the friendly layers are not available. When you connect to a production server over SSH, there is no desktop. When a container fails to start in a CI pipeline, the only evidence is text in a log. When a build tool misbehaves, its documentation will tell you which command-line flag to add. Sooner or later, every developer ends up in a terminal, and the ones who are comfortable there solve problems faster and with less stress.

There are several concrete reasons the command line keeps earning its place:

- **Composability.** Small commands can be chained together so the output of one becomes the input of the next. A question like "which ten IP addresses made the most requests to our API yesterday?" can be answered with one line that combines a handful of tools, with no custom program required.
- **Repeatability.** A command you typed once can be saved, shared in a chat message, pasted into documentation, or turned into a script. Clicking through a sequence of dialogs is much harder to reproduce exactly.
- **Automation.** Anything you can do by typing, a script can do for you at 3 a.m. Backups, deployments, report generation, and cleanups all start as commands a human ran by hand.
- **Remote work.** Servers, virtual machines, and containers are usually managed over a text connection. Knowing the command line means you can work on a machine on the other side of the world as easily as on your laptop.

Meet Emberline

Throughout this book we follow a single fictional project so that examples build on each other. **Emberline** is a community radio and podcast platform run by a small team of developers. It has a Node.js API, a static front end served by nginx, audio uploads that are transcoded with ffmpeg, a PostgreSQL database, log files, nightly backups, and scheduled jobs that publish podcast RSS feeds for shows such as "Night Owl Sessions", "Local Voices", and "Morning Commute".

The team develops on laptops running Linux, macOS, or Windows with WSL, and keeps the project in `~/projects/emberline/`. In production, Emberline runs on an Ubuntu LTS server called `emberline-prod`, with the application installed under `/srv/emberline/` and logs written to `/var/log/emberline/`. You do not need to understand any of that yet. By the end of the book you will have seen every one of those pieces from the command line, and in Chapter 28 you will put a server like this together from nothing.

```
~/projects/emberline/
├── api/        Node.js API source
├── web/        static front end
├── scripts/    backup.sh, deploy.sh, transcode.sh, ...
├── config/     configuration files
├── docs/       team documentation
├── data/       sample data such as episodes.csv
└── logs/       local development logs
```

Terminal, Shell, and Console: Untangling the Words

People use the words terminal, shell, console, and command line almost interchangeably, and in casual conversation that is fine. But the distinctions are real, and understanding them will save you confusion later, for example when you wonder why changing a font is done in one place while changing your prompt is done somewhere else.

The terminal

Historically, a **terminal** was a physical device: a keyboard and a screen (or earlier, a printer) connected by a cable to a large shared computer. The terminal itself did no computing. It sent the characters you typed to the computer and displayed whatever characters came back.

Today we use **terminal emulators**: ordinary graphical applications that imitate those old devices. GNOME Terminal, Konsole, Windows Terminal, macOS Terminal, iTerm2, Alacritty, kitty, and WezTerm are all terminal emulators. Their job is to draw text in a window, handle fonts, colors, tabs, scrollback, and copy-and-paste, and pass your keystrokes along to a program running inside them. When people say "open a terminal", they mean "start a terminal emulator".

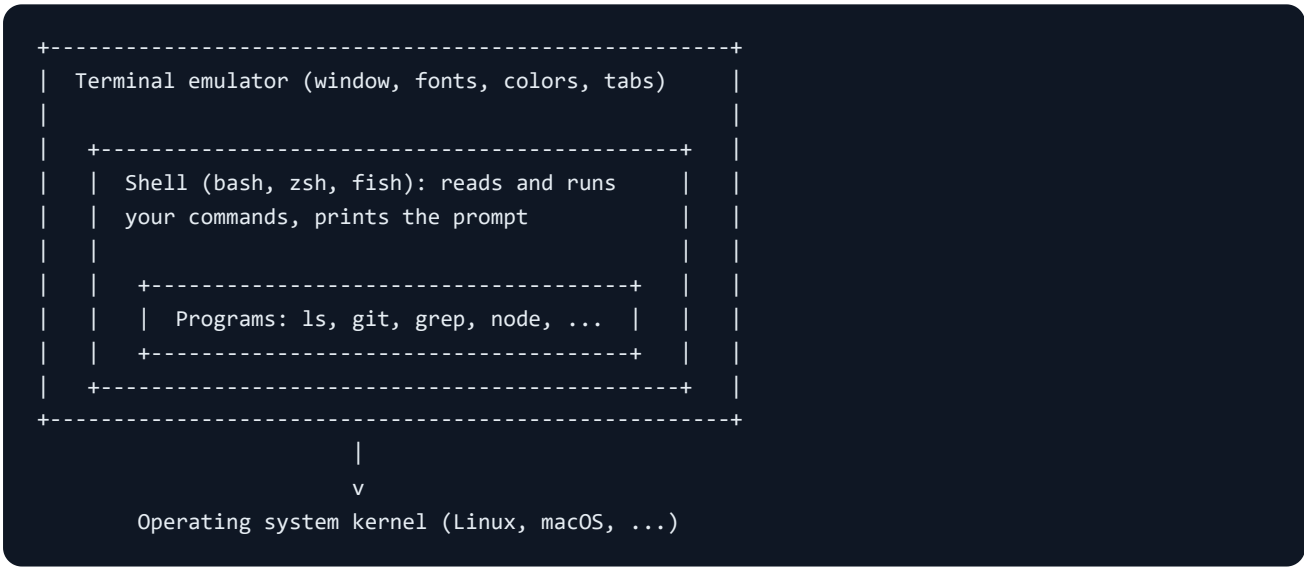
The shell

The **shell** is the program running inside the terminal that actually understands your commands. It prints a prompt, reads the line you type, works out what you meant (expanding wildcards, looking up variables, finding the program you named), runs that program, and then shows the prompt again. The shell is also a full programming language: it has variables, loops, conditionals, and functions, which is why you can write shell scripts.

The separation matters. The terminal decides how text *looks*; the shell decides what text *means*. If you want a bigger font, you change a terminal setting. If you want a new alias or a different prompt, you change a shell setting. You can run the same shell in many different terminal emulators, and you can run different shells in the same terminal.

The console

Console originally meant the terminal physically attached to a machine, the one an operator used to control it. On Linux today it usually means the text-mode screens you get without any graphical desktop, such as the screen you see when a server boots, or the virtual consoles you can reach on many desktop Linux systems with `Ctrl+Alt+F3` and similar key combinations. Cloud providers also offer a "serial console" or "web console" that gives you this kind of low-level access when networking is broken. For everyday purposes, a console is just a terminal that is especially close to the machine.



Meet the Shells: bash, zsh, and fish

There is not just one shell. Over the decades many have been written, and a few are in common use today. The good news is that the basics you will learn in the first half of this book work the same in all of the popular ones.

bash

bash, the "Bourne Again SHell", is the GNU project's shell and the default interactive shell on most Linux distributions. It is also the shell you will find on almost every server you log in to, and the most common target for shell scripts. bash is compatible with the older Bourne shell (`sh`) and the POSIX shell standard, while adding many conveniences such as command history, tab completion, and arrays. This book uses bash for all its examples and scripts, because it is the shell you can rely on being there.

zsh

zsh (the Z shell) is very similar to bash for everyday use but adds more powerful completion, globbing, and theming features. It has been the default shell for new user accounts on macOS for several years, so if you use a Mac you are probably running zsh already. Many developers also install it on Linux, often together with a configuration framework such as Oh My Zsh. Almost every interactive example in this book works unchanged in zsh; where there is a difference that matters, we point it out.

fish

fish, the "friendly interactive shell", focuses on a pleasant experience out of the box: syntax highlighting as you type, suggestions based on your history, and readable configuration. Its scripting language is deliberately *not* compatible with bash, which means scripts written for bash will not run under fish, and some command-line idioms look different. fish is a fine optional extra for interactive use once you know the basics, but learn bash first, because that is what servers and scripts expect.

sh and other shells

To see which shell you are using, and which shells are installed, try these:

```
$ echo "$SHELL"
/bin/bash
$ ps -p $$
    PID TTY          TIME CMD
   4817 pts/0    00:00:00 bash
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/usr/bin/sh
/bin/dash
/usr/bin/dash
```

`$SHELL` holds your *login* shell, the one configured for your account. `$$` is the process ID of the shell you are typing into right now, so `ps -p $$` shows the shell that is actually running, which can differ if you started another shell by hand. `/etc/shells` lists the shells the system accepts as login shells.

> **NOTE**
>
> On macOS the first command will usually print `/bin/zsh`. That is fine for following along. When this book shows a bash-specific feature, you can type `bash` to start a bash session inside your zsh session, and `exit` to return.

The Unix Philosophy

Linux belongs to the Unix family of operating systems, and it inherits a set of design ideas that explain why the command line works the way it does. These ideas are often summarized as the **Unix philosophy**. There are many phrasings, but the core is simple:

- **Write programs that do one thing and do it well.** `sort` sorts lines. `wc` counts them. `grep` selects lines that match a pattern. None of them tries to do everything.
- **Write programs that work together.** Tools read text from their input and write text to their output, so you can connect them.
- **Use text as a universal interface.** Because almost everything is plain text (configuration files, logs, command output), the same tools work on all of it.

The mechanism that connects programs is the **pipe**, written with the `|` character. It sends the output of the command on its left into the input of the command on its right. You will study pipes properly in Chapter 7, but here is a taste using a small sample of Emberline's episode data:

```
$ cat data/episodes.csv
id,show,title,duration_sec,published_at
101,Night Owl Sessions,Late Trains and Lo-Fi,3120,2026-03-02
102,Local Voices,The Bakery on Mill Street,1845,2026-03-04
103,Morning Commute,Traffic and Tea,1290,2026-03-05
104,Night Owl Sessions,Rain on the Roof,3380,2026-03-09
105,Local Voices,Harbor Stories,2210,2026-03-11
$ cut -d, -f2 data/episodes.csv | tail -n +2 | sort | uniq -c | sort -rn
      2 Night Owl Sessions
      2 Local Voices
      1 Morning Commute
```

That one line uses five separate programs. `cut` extracts the second comma-separated field (the show name), `tail -n +2` drops the header line, `sort` groups identical names together, `uniq -c` counts each group, and the final `sort -rn` orders the counts from largest to smallest. None of those tools knows anything about podcasts or CSV headers. Combined, they answer a question nobody wrote a program for. Do not worry about the details yet; Chapters 7 and 9 cover every piece. The point is the style of thinking: break a problem into small steps, and let small tools handle each one.

Linux: Kernel and Distributions

Strictly speaking, **Linux** is a *kernel*: the core of the operating system that manages memory, schedules processes, talks to hardware through drivers, and provides the file system and networking layers. The kernel on its own is not something you can log in to. To get a usable system, you need a lot more: a boot loader, an init system (on most distributions today that is systemd), a shell, the core utilities such as `ls` and `cp`, libraries, a package manager, and often a graphical desktop.

A **distribution** (or "distro") is a complete operating system built around the Linux kernel, with all those pieces chosen, configured, packaged, and tested together. Different distributions make different choices about release schedules, default software, package formats, and how much they change once a release is out. That is why there are so many of them, and why instructions sometimes differ slightly between them.

Many of the everyday commands you will use come from the GNU project, for example the GNU coreutils package that provides `ls`, `cp`, `mv`, and dozens more. That is why you will sometimes see the whole system called "GNU/Linux". This book targets these GNU tools. macOS and the BSD systems have their own versions of the same commands, which usually behave the same way for basic use but differ in some options; we flag the important differences as we go.

The major families

You do not need to memorize distributions, but it helps to recognize the main families, because members of a family share a package manager and many conventions.

Family	Examples	Package tool	Typical use
Debian	Debian, Ubuntu, Linux Mint	`apt` (.deb)	Servers, desktops, cloud
Red Hat	Fedora, RHEL, Rocky, AlmaLinux	`dnf` (.rpm)	Enterprise servers, desktops
Arch	Arch Linux, Manjaro	`pacman`	Enthusiast desktops
Alpine	Alpine Linux	`apk`	Small containers

Debian and Ubuntu. Debian is one of the oldest community distributions and is known for stability. Ubuntu is based on Debian and publishes long-term support (LTS) releases that receive updates for several years, which makes them very popular on servers and in the cloud. Emberline's production server runs a recent Ubuntu LTS release, so most server examples in this book use Ubuntu conventions.

Fedora and RHEL. Fedora is a community distribution that moves quickly and adopts new technology early. Red Hat Enterprise Linux (RHEL) is a commercial distribution aimed at long-lived enterprise deployments, and Rocky Linux and AlmaLinux are community rebuilds compatible with it. They use `dnf` to manage RPM packages and enable SELinux by default.

Arch. Arch Linux is a "rolling release": instead of big version upgrades, packages are updated continuously. It is minimal by design and well known for its excellent wiki, which is useful reading even if you never run Arch.

Alpine. Alpine Linux is tiny. It uses the musl C library and BusyBox instead of the GNU versions of many tools, which keeps images small and makes it a common base for containers. The trade-off is that some commands have fewer options than their GNU equivalents, and software compiled for other distributions may not run on it unchanged.

To find out which distribution you are running, read `/etc/os-release`, a small text file present on essentially every modern distribution:

```
$ cat /etc/os-release
PRETTY_NAME="Ubuntu 24.04 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
...
```

Your output will differ depending on your distribution and version. Note the `ID_LIKE` line: it tells you which family the system belongs to, which is a quick way to know whether to reach for `apt` or `dnf`. Chapter 13 covers package managers in depth.

> **TIP**
>
> When you search the web for help with a command, include your distribution family in the query, for example "ubuntu" or "fedora". Instructions for installing software and configuring services often differ between families, while instructions for everyday commands like `ls` and `grep` usually do not.

Anatomy of a Command

Almost every command you type follows the same basic shape. Once you can see that shape, even unfamiliar commands become readable.

```
$ ls -l -h /var/log
  |  |  |  |
  |  |  |  +--- argument: what to operate on
  |  |  +------ option: -h, human-readable sizes
  |  +--------- option: -l, long listing format
  +------------ command: the program to run
```

The command

The first word is the **command**: the name of a program (such as `ls`, which lists files), a shell built-in (such as `cd`), an alias, or a function. When you type a program name, the shell searches a list of directories stored in the `PATH` variable to find the matching executable file. You can ask where a command comes from with `type`:

```
$ type ls
ls is aliased to `ls --color=auto'
$ type cd
cd is a shell builtin
$ type date
date is /usr/bin/date
$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo
```

This output shows three different kinds of command. `ls` is an alias, a shortcut that many distributions define so that `ls` automatically adds color. `cd` is built into the shell itself. `date` is a separate program stored in `/usr/bin`. `type -a` shows every match, which reveals that `echo` exists both as a built-in and as a program; the built-in wins. (Two paths appear because on many modern distributions `/bin` is simply a link to `/usr/bin`.) Chapter 25 explains `PATH` in detail.

Options

Options, also called flags or switches, change how a command behaves. They usually start with a dash. There are two main styles:

- **Short options** are a single dash followed by a single letter, like `-l` or `-h`. Most programs let you combine them, so `ls -l -h -a` can be written `ls -lha`.
- **Long options** are two dashes followed by a word, like `--human-readable` or `--all`. They are easier to read in scripts and documentation, and GNU tools provide long names for most of their short options.

Some options take a value. For short options the value usually follows as the next word (`head -n 5`) or immediately after the letter (`head -n5`). For long options it usually follows an equals sign (`--lines=5`) or a space. The following commands are equivalent:

```
$ ls -l -a -h
$ ls -lah
$ ls --all --human-readable -l
$ head -n 5 data/episodes.csv
$ head --lines=5 data/episodes.csv
```

Arguments

Arguments (sometimes called operands) are the things the command operates on: file names, directory names, URLs, text strings, and so on. Some commands need none, some need exactly one, and many accept several. Arguments are separated by spaces, which is why a file name that contains spaces must be quoted or escaped:

```
$ ls "Night Owl Sessions"
episode-101.mp3  episode-104.mp3
$ ls Night\ Owl\ Sessions
episode-101.mp3  episode-104.mp3
$ ls Night Owl Sessions
ls: cannot access 'Night': No such file or directory
ls: cannot access 'Owl': No such file or directory
ls: cannot access 'Sessions': No such file or directory
```

The last attempt shows what happens without quotes: the shell splits the line into three separate arguments, and `ls` looks for three files that do not exist. This is one of the most common beginner mistakes, and it is also a common source of bugs in scripts. Quoting is covered thoroughly in Chapter 21. For now, remember that quotes group words into a single argument, and that it is wise to avoid spaces in file names you create yourself.

Reading the Prompt

When the shell is ready for input, it prints a **prompt**. A typical default prompt on Ubuntu looks like this:

```
alex@laptop:~/projects/emberline$
```

Reading from left to right, this tells you the user name (`alex`), the host name of the machine (`laptop`), and the current directory (`~/projects/emberline`, where `~` is shorthand for your home directory). The final `$` means you are an ordinary user. If you see `#` instead, you are working as `root`, the all-powerful administrator account, and you should slow down, because the system will not stop you from damaging it. Chapter 11 explains root and `sudo`.

The host name in the prompt is surprisingly important. When you have terminals open to your laptop and to the production server at the same time, the prompt is your main clue about where a command is about to run:

```
alex@laptop:~/projects/emberline$
deploy@emberline-prod:/srv/emberline$
```

Prompts are fully customizable, and yours might look different: zsh on macOS shows something like `alex@MacBook-Pro emberline %`, and many developers use themes that add Git branch names, colors, and icons. Chapter 25 shows how to build your own. In this book we show only `$` before each command to keep examples short, and `#` on the rare occasions a root shell is involved. Do not type the `$` itself.

> **WARNING**
>
> Never paste a command into a terminal without reading it first, and be especially careful with commands copied from web pages, chat messages, or tutorials that begin with `sudo` or that you see running as root (`#` prompt). A single line can delete files, change permissions, or download and run software. If you do not understand what a command does, look it up (Chapter 5 shows how) before you press Enter.

Your First Commands

Enough theory. Open a terminal (Chapter 2 describes every way to get one if you do not have a Linux shell yet) and try the following commands. They are all harmless: they only report information and change nothing.

whoami: who am I logged in as?

```
$ whoami
alex
```

`whoami` prints the name of the user you are currently acting as. That sounds trivial, but on servers it is a genuinely useful check. After switching users with `sudo -i` or `su`, or after connecting to a machine with a different account, `whoami` confirms whose permissions your next command will run with. The related command `id` shows more detail, including the groups you belong to:

```
$ id
uid=1000(alex) gid=1000(alex) groups=1000(alex),4(adm),27(sudo),999(docker)
```

hostname: which machine is this?

```
$ hostname
laptop
$ hostname -I
192.0.2.24 172.17.0.1
```

`hostname` prints the machine's name. On Linux, `hostname -I` (capital I) lists the IP addresses assigned to the machine, which is handy when you need to connect to it from elsewhere; this option is specific to the Linux version of `hostname`. Chapter 16 introduces the more complete `ip` command. On the

production server, the same command would print `emberline-prod`, which is exactly what you want to see before running anything with consequences.

date: what time is it here?

```
$ date
Tue Mar 10 14:32:07 UTC 2026
$ date +%Y-%m-%d
2026-03-10
$ date -u +"%Y-%m-%dT%H:%M:%SZ"
2026-03-10T14:32:07Z
$ date -d "yesterday" +%F
2026-03-09
```

Plain `date` prints the current date and time in the system's time zone. The argument that starts with a plus sign is a **format string**: `%Y` is the four-digit year, `%m` the month, `%d` the day, `%H:%M:%S` the time, and `%F` is a shortcut for `%Y-%m-%d`. `-u` reports the time in UTC, which is what servers usually log in. Formats like these are how Emberline's backup script will later name its archive files, for example `emberline-2026-03-10.tar.gz`, so that they sort in date order.

The last example uses `-d` to describe a different date in plain English. That option belongs to GNU `date`. On macOS, the BSD version of `date` uses `-v` for date arithmetic instead (for example `date -v-1d`), and does not understand `-d "yesterday"`. This is the first of several small GNU-versus-BSD differences you will meet.

uname: what system is this?

```
$ uname
Linux
$ uname -r
6.8.0-45-generic
$ uname -m
x86_64
$ uname -a
Linux laptop 6.8.0-45-generic #45-Ubuntu SMP ... x86_64 x86_64 x86_64 GNU/Linux
```

`uname` reports information about the kernel. On its own it prints the kernel name (`Linux`, or `Darwin` on macOS). `-r` prints the kernel release, `-m` the machine hardware name (`x86_64` for most PCs and cloud servers, `aarch64` or `arm64` for ARM machines such as Apple Silicon Macs and many newer cloud instances), and `-a` prints everything at once. The architecture matters when you download precompiled software or build container images: a binary built for `x86_64` will not run natively on `aarch64`.

echo: say something back

```
$ echo Hello, Emberline
Hello, Emberline
$ echo "Today is $(date +%A)"
Today is Tuesday
$ echo "My home directory is $HOME"
My home directory is /home/alex
$ echo 'My home directory is $HOME'
My home directory is $HOME
```

`echo` prints its arguments, separated by spaces, followed by a newline. It is the command-line equivalent of a print statement, and you will use it constantly: to display messages in scripts, to check the value of variables, and to preview what a command will do before running it for real. The examples also show two shell features at work. `$(date +%A)` is **command substitution**: the shell runs the inner command and pastes its output into the line. `$HOME` is a **variable** that holds the path of your home directory. Notice that double quotes allow the shell to expand variables, while single quotes keep the text exactly as written. That difference is one of the most important rules in shell programming, and Chapter 21 explains it fully.

When Things Go Wrong

Part of becoming comfortable at the command line is learning to read the messages the shell and programs print when something goes wrong. They look intimidating at first, but they are usually precise. Here are the ones you will see most often in your first weeks:

```
$ whoamii
whoamii: command not found
$ sl
Command 'sl' not found, but can be installed with:
sudo apt install sl
$ ls /nonexistent
ls: cannot access '/nonexistent': No such file or directory
$ ls --colour-blind
ls: unrecognized option '--colour-blind'
Try 'ls --help' for more information.
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

- **command not found** means the shell could not find a program with that name in your `PATH`. Check the spelling first. On Ubuntu, a helper often suggests a package that provides the command; on other systems you will simply see the short message.
- **No such file or directory** means a path you gave does not exist. Check for typos, capitalization, and whether you are in the directory you think you are in (`pwd`).