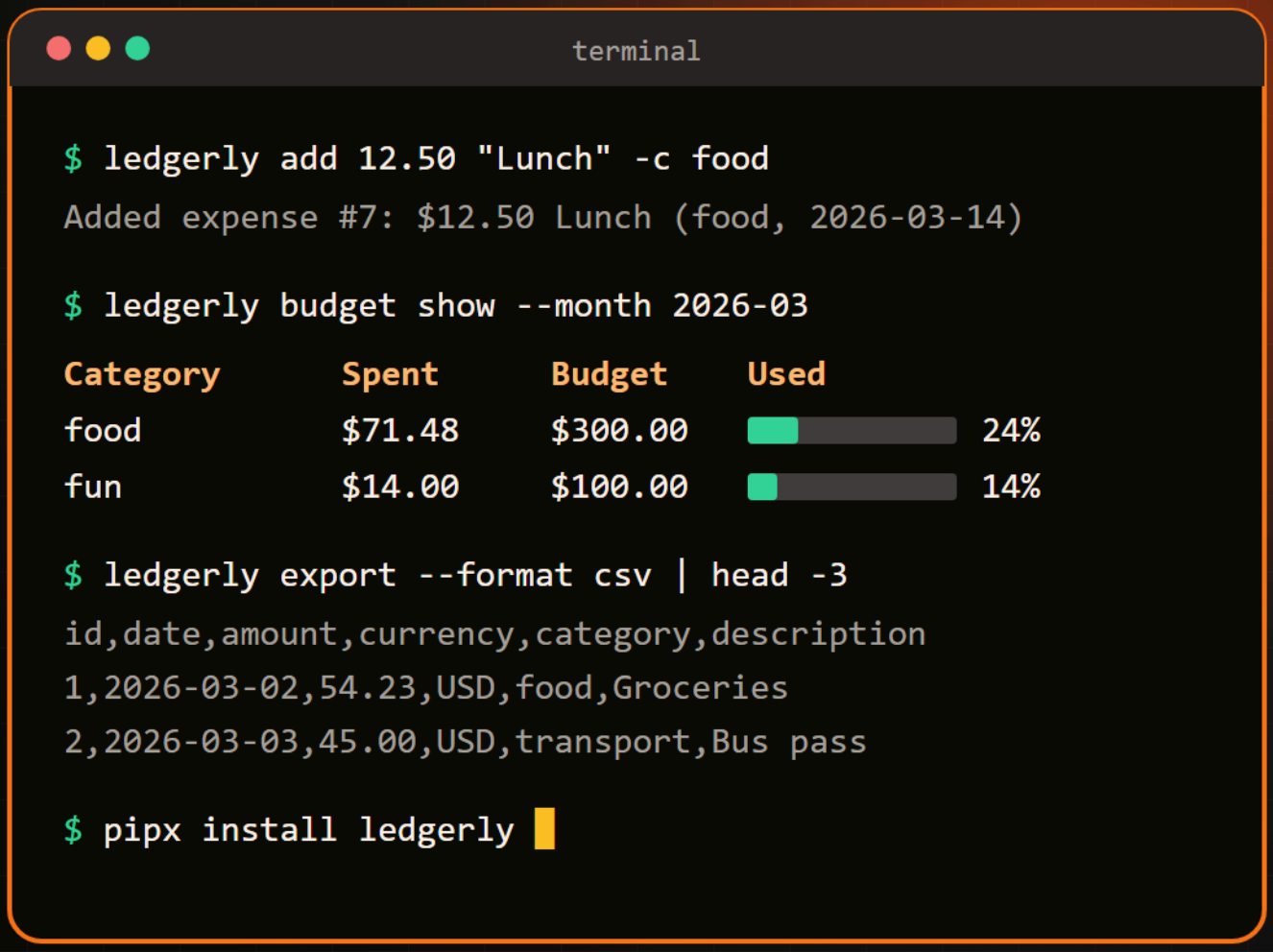


A PRACTICAL GUIDE

Build Your First Command-Line Application

Design, build, test, and ship real CLI tools with Python

argparse · Typer · Rich · SQLite · pytest
Packaging · pipx · PyPI · Cross-platform

DESIGN · BUILD · TEST · SHIP

THENEXBOOKS

Build Your First Command-Line Application

A Practical Guide to Designing, Building, Testing, and Shipping CLI Tools with Python

First Edition

THENEXBOOKS

Build Your First Command-Line Application
A Practical Guide to Designing, Building, Testing, and Shipping CLI Tools with Python

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 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. Python and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. Windows and PowerShell are trademarks of Microsoft Corporation. macOS is a trademark of Apple Inc. Linux is a registered trademark of Linus Torvalds. GitHub is a trademark of GitHub, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by the Python Software Foundation or the maintainers of any library mentioned.

Ledgerly, the application built in this book, is a teaching example. The web service `rates.example.com` used in examples does not exist; `example.com` domains are reserved for documentation.

Preface

Most programmers learn to write scripts long before they learn to write tools. A script solves your problem once, on your machine, when you remember how to run it. A tool solves a problem repeatedly, for other people too: it has a clear name, helpful options, sensible error messages, a way to install it, and documentation that answers questions before they are asked. The distance between the two is not talent. It is a set of practices that nobody tends to teach in one place.

Command-line applications are the best place to learn those practices. They have no graphical interface to design, so you can focus on the things that make any software good: a clear interface, reliable data handling, honest error reporting, tests, and a clean way to ship. They are also genuinely useful. Developers, data analysts, system administrators, and scientists use command-line tools every day because they are fast, scriptable, and easy to combine with other tools.

This book takes you from a ten-line Python script to a real, installable command-line application with subcommands, colorful output, a local database, configuration files, tests, and a published release. We build one tool — an expense tracker called Ledgerly — step by step, and every chapter explains *why* each decision is made, not just what to type.

Who This Book Is For

- **Beginner and intermediate Python programmers** who can write functions and loops and want to build something complete.
- **Students and self-taught developers** looking for a practical project that teaches professional habits.
- **Data analysts, scientists, and engineers** whose scripts have grown into tools that colleagues now depend on.
- **Developers from other languages** who want a clear path to shipping Python command-line tools.

You need basic Python: variables, functions, lists and dictionaries, loops, conditions, and a little experience with classes. Everything else is explained as it appears.

What You Will Learn

- How terminals, shells, arguments, standard streams, and exit codes work — and how your program fits in.
- How to design a command-line interface before writing code: commands, options, flags, and help text.
- Argument parsing with `argparse`, then modern CLIs with Click and Typer.
- Interactive prompts, colorful tables and progress bars with Rich, and machine-readable output.

- Working with files and paths safely, storing data in JSON and SQLite, and managing configuration.
- Pipes, import and export, friendly errors, logging, validation of dates and money, and testing with pytest.
- Help text, shell completion, calling web APIs, performance, cross-platform behavior, and plugins.
- Packaging with `pyproject.toml`, distributing with pipx and PyPI, and maintaining your tool with CI and changelogs.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains how command-line programs work, sets up your environment, reviews the Python you need, and plans Ledgerly. **Part II — Arguments, Commands, and Output** (Chapters 6–10) builds the command-line interface with argparse and then Typer, and makes the output genuinely helpful. **Part III — Working with Data** (Chapters 11–15) adds files, JSON, SQLite, configuration, and pipes. **Part IV — Making It Robust** (Chapters 16–19) covers errors, logging, validation, and testing. **Part V — Polish and Power Features** (Chapters 20–24) adds documentation, completion, network access, performance work, cross-platform support, and plugins. **Part VI — Shipping Your Tool** (Chapters 25–28) packages, publishes, and maintains Ledgerly, ending with a complete case study. The appendices provide a cheat sheet, templates, a design and release checklist, and a glossary.

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

The Running Example: Ledgerly

Ledgerly is a personal expense tracker you use from the terminal. You record what you spend with `ledgerly add 12.50 "Lunch" --category food`, list and search your expenses, set monthly budgets, see reports, import and export CSV files, and convert currencies. It starts in Chapter 3 as a tiny script that appends a line to a text file. By Chapter 28 it is a tested, documented, packaged application with a SQLite database, configuration files, colorful output, and a 1.0 release anyone can install with a single command.

Conventions

Examples use Python 3.12; most code also works on Python 3.10 and later. Where Windows (PowerShell) and macOS or Linux (bash or zsh) commands differ, we show both. Terminal sessions start with a `$` prompt for macOS and Linux or `PS>` for PowerShell; you type what follows the prompt.

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

> **NOTE**
>
> Background information, platform differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can lose data or confuse your users.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `src/ledgerly/cli.py`, tell you where the code lives.

Contents

Preface

PART I — FOUNDATIONS

1 What Is a Command-Line Application? Terminals, Shells, and Why CLIs Matter

2 Setting Up: Python, a Terminal, an Editor, and Virtual Environments

3 Your First Script: Output, Input, Arguments, and Exit Codes

4 Python Essentials for CLI Builders: Functions, Modules, Dataclasses, and Exceptions

5 Planning Ledgerly: Requirements, Commands, and Project Layout

PART II — ARGUMENTS, COMMANDS, AND OUTPUT

6 Parsing Arguments with argparse

7 Subcommands and Command Design

8 Modern CLI Frameworks: Click and Typer

9 Interactive Input: Prompts, Confirmations, and Passwords

10 Output That Helps: Formatting, Color, Tables, and Progress with Rich

PART III — WORKING WITH DATA

11 Files and Paths: pathlib, Safe Writes, and Where to Store Data

12 Storing Data in JSON, CSV, and TOML

13 Reliable Local Storage with SQLite

14 Configuration: Config Files, Environment Variables, and Precedence

15 Pipes and Streams: stdin, stdout, stderr, Import, and Export

PART IV — MAKING IT ROBUST

16 Errors, Exit Codes, and Friendly Messages

17 Logging, Verbosity, and Debug Modes

18 Validation and Edge Cases: Dates, Money, Unicode, and Bad Input

19 Testing Your CLI with pytest

PART V — POLISH AND POWER FEATURES

20 Help Text, Documentation, and Shell Completion

21 Talking to the Network: Calling Web APIs Safely

22 Performance: Startup Time, Large Files, and Streaming

23 Cross-Platform CLIs: Windows, macOS, and Linux

24 Extensibility: Plugins and Hooks

PART VI — SHIPPING YOUR TOOL

25 Packaging with pyproject.toml and Entry Points

26 Distributing: pipx, PyPI, and Standalone Executables

27 Versioning, Changelogs, CI, and Maintaining Your CLI

28 Case Study: Ledgerly from First Commit to Release 1.0

APPENDICES

A CLI Cheat Sheet

B Templates and Reference

C CLI Design and Release Checklist

D Glossary

PART I

Foundations

How terminals, shells, arguments, and exit codes work; a working Python setup; the language features every CLI builder relies on; and a plan for Ledgerly before the first real line of code.

1 · What Is a Command-Line Application?
2 · Setting Up
3 · Your First Script
4 · Python Essentials for CLI Builders
5 · Planning Ledgerly

CHAPTER 1 · PART I — FOUNDATIONS

What Is a Command-Line Application? Terminals, Shells, and Why CLIs Matter

Before we write a single line of code, we need a clear picture of what a command-line application actually is, where it runs, and why so many developers reach for one first. This chapter gives you that picture, introduces the vocabulary we will use for the rest of the book, and shows you the tool we are going to build together: Ledgerly.

A Program You Talk To in Words

Most of the software you use every day has a graphical user interface, or GUI. You click buttons, drag windows, tick checkboxes, and scroll through lists. A **command-line application**, usually shortened to **CLI** (command-line interface), works differently. Instead of pointing and clicking, you type a line of text that names the program and describes what you want it to do. You press Enter, the program runs, prints its results as text, and then hands control back to you so you can type the next command.

Here is a tiny example. On almost any computer with Python installed, you can open a terminal window and type the following line:

Terminal

```
$ python --version
Python 3.12.4
```

That single line contains everything a CLI conversation needs. The word `python` names the program. The text `--version` is an instruction to that program. The response, `Python 3.12.4`, is the program's answer, printed as plain text. (Your version number will almost certainly be different; that is fine.) The `$` at the start of the line is not something you type. It is the **prompt**, a marker that the shell prints to say "I am ready for your next command." We use `$` in this book for macOS and Linux sessions and `PS>` for Windows PowerShell sessions when the two differ.

This style of interaction is older than the mouse, but it is far from obsolete. Every major operating system still ships with a command line, every cloud provider offers a command-line tool, and nearly every programming language comes with a compiler, package manager, or test runner that you drive by typing commands. By the end of this book you will have designed, built, tested, packaged, and published your own CLI, and you will understand the choices behind every part of it.

What a CLI is not

A CLI is not the same as a full-screen text program such as a terminal text editor; those are called **TUIs** (text user interfaces) and behave more like GUIs drawn with characters. A classic CLI's rhythm is "command in, results out, done." Because it finishes and returns a result, other programs can run it, read its output, and decide what to do next. That is why command-line tools become the building blocks of automation.

Terminal, Shell, and Console: Three Words, Three Jobs

People often use the words *terminal*, *shell*, and *console* as if they meant the same thing. In casual conversation that is usually harmless, but when something goes wrong it pays to know which piece is responsible for what. There are really three layers involved every time you type a command.

The terminal: the window

The **terminal** (more precisely, the *terminal emulator*) is the application that draws the window, shows text, handles your keyboard, and deals with things like fonts, colors, scrolling, copy and paste, and tabs. On Windows the modern choice is **Windows Terminal**. On macOS the built-in app is simply called **Terminal**, and many developers also use third-party alternatives. Linux desktops ship with their own terminal apps, such as GNOME Terminal or Konsole. Code editors like Visual Studio Code also include an integrated terminal panel.

The terminal does not understand commands. It only knows how to display characters and send your keystrokes somewhere. That "somewhere" is the shell.

The shell: the interpreter

The **shell** is a program that reads the line you type, interprets it, finds the program you named, starts it, and passes along the rest of what you typed. Shells also provide their own features: variables, wildcards such as `*.csv`, history, tab completion, pipes, and small scripting languages with `if` statements and loops.

The shells you are most likely to meet are:

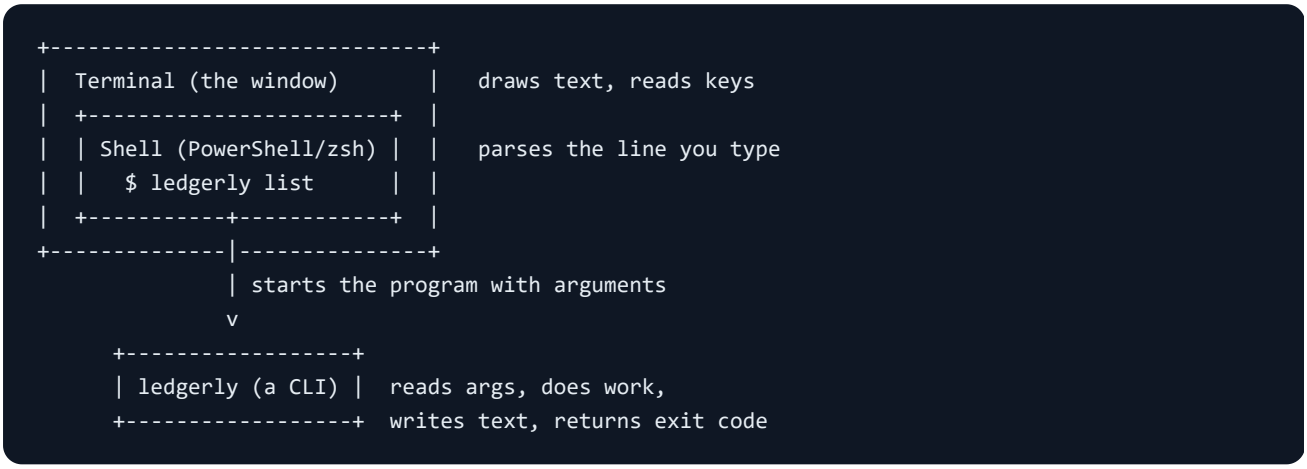
- **PowerShell** on Windows. Modern Windows machines include Windows PowerShell 5.1, and the newer cross-platform PowerShell 7 can be installed alongside it. PowerShell passes objects between commands internally, but when it runs an external program such as Python it exchanges plain text just like other shells.
- **zsh**, the default shell on current versions of macOS.
- **bash**, the long-standing default on most Linux distributions and still very common everywhere.
- **Command Prompt** (`cmd.exe`), the older Windows shell. It still works, but we will use PowerShell for Windows examples in this book.

Different shells have slightly different rules for quoting, variables, and wildcards. You will see this whenever we show both a PowerShell and a bash version of the same command. The good news is that the program you build does not care which shell started it. Python receives the same list of arguments either way, once the shell has done its interpretation.

The console: the historical term

The word **console** originally meant the physical keyboard and screen attached to a large computer. Today it is used loosely, and on Windows it also names the subsystem text-mode programs talk to. Treat it as a synonym for "the text window where a CLI runs."

How the pieces fit together



> **NOTE**
>
> When we say "open a terminal" in this book, we mean: start your terminal application, which will in turn start your default shell and show you a prompt. Chapter 2 walks through doing that on Windows, macOS, and Linux.

Why Developers Love Command-Line Tools

If GUIs are so friendly, why do experienced developers spend so much time typing commands? There are several practical reasons, and understanding them will shape how we design Ledgerly.

They are scriptable

Anything you can type, you can save in a file and run again. If you find yourself running the same five commands every morning, you can put them in a script and run one command instead. You can schedule that script to run automatically at night. You can run it on a server that has no screen at all. A GUI action, by contrast, usually requires a human to click the right place at the right time. That difference is enormous once you have repetitive work to do.

They are composable

Because CLIs read and write plain text, you can connect them together. The output of one program becomes the input of the next. The connector is called a **pipe**, written with the vertical bar character `|`. For example, on macOS or Linux you can list the files in a folder and count them:

Terminal (macOS/Linux)

```
$ ls | wc -l
12
```

Neither `ls` nor `wc` knows anything about the other. Each does one small job, and the shell wires them together. PowerShell supports pipes too, and it adds its own twist by passing structured objects between its built-in commands:

Terminal (PowerShell)

```
PS> Get-ChildItem | Measure-Object

Count    : 12
...
```

Later in the book, Ledgerly will take advantage of composition. You will be able to export expenses as CSV and pipe them to another tool, or feed a file of expenses into `ledgerly import`. Chapter 15 is devoted to pipes and streams.

They are fast

For someone who knows the commands, typing is often quicker than navigating menus. You also get shell history for free: press the Up arrow and the previous command reappears, ready to edit and run again.

They are precise and repeatable

A command line is an exact record of what you asked for. `ledgerly report monthly --month 2026-03` can go into documentation, a bug report, or an automated test without losing any detail, unlike "open the app, click Reports, pick March."

They are cheap to build and easy to maintain

A useful CLI can be a single Python file. You do not need to design screens or learn a GUI toolkit, so you can focus on the problem you actually want to solve.

> **TIP**
>
> A CLI and a GUI are not rivals. Many popular applications have both, and often the GUI is built on top of the same core code that powers the CLI. If you structure your code well, with the "real work" separated from the part that talks to the user, adding another interface later becomes much easier. We will follow that separation in Ledgerly from the start.

Anatomy of a Command

Let us dissect a command so that we share the same vocabulary. Here is one that we will be able to run on Ledgerly once it is finished:

Terminal

```
$ ledgerly add 12.50 "Lunch" --category food --date 2026-03-14
Added expense #7: $12.50 Lunch (food, 2026-03-14)
```

The shell splits that line into a list of separate pieces called **arguments** (sometimes *tokens* or *words*). Quotation marks group words that contain spaces, so `"Lunch with Sam"` would become a single argument. Here is how each piece is named:

The parts of a command

```
ledgerly  add  12.50  "Lunch"  --category food  --date 2026-03-14
   |       |     |       |          |                 |
program    |  argument argument   option + value   option + value
       subcommand
```

Part	Example	What it does
Program	`ledgerly`	The executable the shell starts.
Subcommand	`add`	Chooses which action the program performs. Also called a *command* or *verb*.
Positional argument	`12.50`, `"Lunch"`	Required values identified by their position in the line.
Option	`--category food`	A named setting that takes a value. Usually optional, with a sensible default.
Flag	`--yes`, `-v`	An option with no value. Its presence turns a behavior on.

Programs and subcommands

Simple tools do one thing, so the program name alone says what happens. Tools that do several related things, such as version control systems or package managers, use **subcommands** to choose among actions. You have probably seen this style: a version control tool has subcommands for committing, branching, and showing history, and a Python package installer has subcommands for

installing, uninstalling, and listing packages. Some tools even nest subcommands in groups. Ledgerly will do this with `ledgerly budget set` and `ledgerly budget show`, where `budget` is a group and `set` and `show` are its commands.

Positional arguments

A **positional argument** is identified purely by where it appears. In `ledgerly add 12.50 "Lunch"`, the program knows the first value after `add` is the amount and the second is the description. Positional arguments are ideal for the one or two values a command cannot work without. Too many of them make a command hard to remember, because users must recall the exact order.

Options and flags

An **option** is identified by a name, so its position does not matter. By convention, long option names start with two hyphens (`--category`) and short aliases start with one hyphen and use a single letter (`-c`). The value can usually be written either as a separate argument (`--category food`) or joined with an equals sign (`--category=food`). The libraries we use later accept both forms.

A **flag** is an option that takes no value, such as `--yes` to skip a confirmation question, or `--verbose` to print extra detail. Some flags can be repeated to increase their effect. Ledgerly will support `-v` for more detail and `-vv` for even more, a common convention you will see in many tools.

Two special options appear in nearly every well-behaved CLI: `--help`, which prints a description of how to use the command, and `--version`, which prints the program's version. Users expect them, and we will make sure Ledgerly supports both.

> **WARNING**
>
> Quoting rules differ between shells. In bash and zsh, text in single quotes is passed through exactly as written, while double quotes still allow variables like `$HOME` to expand. PowerShell also treats `$` specially inside double quotes, and it has its own rules for special characters. When a description contains spaces, dollar signs, or quotes, wrap it in single quotes to be safe in both families of shells, for example `ledgerly add 5 'Tip for $5 coffee'`.

Output, Errors, and Exit Codes

A CLI communicates back to you and to other programs through three channels. Understanding them early will save you from design mistakes later.

- **Standard output** (stdout) is where the program writes its normal results: the list of expenses, the report, the confirmation message.
- **Standard error** (stderr) is where the program writes error messages, warnings, and diagnostic chatter. It usually appears in the same terminal window, but it is a separate stream that can be redirected independently.

- **The exit code** (also called exit status or return code) is a small number the program returns when it finishes. By long convention, `0` means success and anything else means something went wrong.

Why separate stdout and stderr? Imagine you run `ledgerly export --format csv` and redirect the output into a file. If an error message were mixed into stdout, it would end up inside your CSV file and corrupt it. By sending errors to stderr, the program keeps data clean while still telling you what went wrong on screen. Ledgerly follows this rule strictly: **data goes to stdout, errors go to stderr**.

Exit codes matter because scripts use them to decide what to do next. You can check the exit code of the last command yourself:

Terminal (macOS/Linux)

```
$ python -c "import sys; sys.exit(3)"
$ echo $?
3
```

Terminal (PowerShell)

```
PS> python -c "import sys; sys.exit(3)"
PS> $LASTEXITCODE
3
```

Ledgerly will use a small, consistent set of exit codes, which we will implement properly in Chapter 16:

Code	Meaning	Example
`0`	Success	An expense was added.
`1`	Runtime error	Expense #99 does not exist; the database file is unreadable.
`2`	Usage error	A required argument is missing or an option value is invalid.
`130`	Interrupted	The user pressed Ctrl+C.

The value `2` for usage errors is the convention followed by Python's `argparse` module and by the Click library, so using it makes Ledgerly behave like the tools your users already know. The value `130` comes from the Unix convention of 128 plus the signal number for an interrupt (signal 2).

The Unix Philosophy in Plain Words

Many of the habits that make command-line tools pleasant to use come from a set of ideas known as the **Unix philosophy**. It grew out of the early Unix operating system, but it applies to CLIs on every platform, including Windows. You do not need to memorize any official wording. In everyday language, it boils down to a few principles.

Do one thing well

A tool should have a clear purpose. Ledgerly tracks personal expenses. It does not also send email, manage passwords, or edit photos. When a tool has a focused job, it is easier to learn, easier to test, and easier to combine with other tools. Focus does not mean tiny: Ledgerly will have a dozen commands. But all of them serve the same purpose.

Work together through text

Programs should read and write text in predictable formats so that other programs can use their output. This is why Ledgerly will offer `--format csv` and `--format json` for listing and exporting. A human-friendly table is great on screen, but a script wants something it can parse without guesswork.

Be quiet when things go well

A good tool does not flood the screen with chatter. If a command succeeds, a short confirmation is plenty. If there is nothing to report, it is fine to report nothing. Extra detail should be available on request, which is exactly what `--verbose` is for, and suppressible entirely with `--quiet`.

Fail loudly and clearly

When something goes wrong, say so on stderr, say what the user can do about it, and return a non-zero exit code. A message like `Error: expense #99 not found. Run 'ledgerly list' to see IDs.` is far more useful than a long Python traceback. We will build a friendly error system in Chapter 16.

Make the common case easy

The most frequent action should need the least typing. For Ledgerly, adding an expense is the most common action, so `ledgerly add 4.75 "Coffee with Sam"` should work with sensible defaults: today's date, the default currency, and a default category. Less common details can be supplied through options when needed.

> **TIP**
>
> Whenever you are unsure how your CLI should behave, look at how established tools you already use handle the same situation. Consistency with widespread conventions (like `--help`, `--version`, `-v` for verbose, and exit code 0 for success) is itself a feature, because your users arrive already knowing how to use it.

Well-Known CLIs and What They Teach Us

You have probably used a few command-line tools already, even if you did not think of them as "applications." Looking at them through a designer's eyes teaches useful lessons.

Version control tools like git show how a large tool can stay organized with subcommands. There are many commands, but they share global options and a consistent style, and each has its own help page. When you forget how something works, the tool itself can remind you. Ledgerly borrows this structure: one program, many focused subcommands, and help available at every level.

Package managers like pip show how a CLI can report progress, explain errors in plain language, and work equally well for a person watching and for an unattended automated build. Ledgerly will aim for that same dual personality.

Python itself is a CLI: `python -m venv .venv` has options, arguments, and exit codes, as you will see in Chapter 2.

Across all of them, the same qualities appear: predictable structure, built-in help, meaningful exit codes, clean separation between data and messages, and output that works for both people and programs. Those qualities are our design goals.

Why Python for Your First CLI

You can build command-line tools in almost any language. We chose Python for this book for several practical reasons.

- **You already know the basics.** This book assumes you are comfortable with variables, functions, lists, dictionaries, loops, and simple classes. That is enough to start building immediately.
- **The standard library is excellent for CLIs.** Python ships with `argparse` for parsing arguments, `pathlib` for file paths, `sqlite3` for a real database, `csv` and `json` for data formats, `logging` for diagnostics, and `tomllib` for reading configuration files. You get a lot before installing anything.
- **The ecosystem is mature.** Libraries such as **Typer** (built on **Click**) turn ordinary Python functions into polished commands, and **Rich** makes colored tables and progress bars easy. We will use them from Chapter 8 onward.
- **Testing and packaging are well supported** with **pytest** and `pyproject.toml`, and one codebase runs on Windows, macOS, and Linux.

Python does have trade-offs: a Python CLI needs an interpreter, and it starts more slowly than a compiled program. Chapter 22 shows how to measure and improve startup time, and Chapter 26 covers distribution, including standalone executables.

We target **Python 3.12** in this book. Everything also works on Python 3.11, and nearly everything works on 3.10; where a feature needs a newer version (such as `tomllib`, added in 3.11), we say so.

Meet Ledgerly: The Tool We Will Build

Learning from isolated snippets only takes you so far. The fastest way to understand how a real tool comes together is to build one from start to finish, making the same decisions a professional would. So throughout this book we build **Ledgerly**, a personal expense tracker that lives in your terminal.

Ledgerly lets you record what you spend, see where your money goes, set monthly budgets for categories, and move data in and out as CSV or JSON. It is small enough to understand completely, yet it touches nearly every topic a real CLI needs: argument parsing, subcommands, validation, storage, configuration, colors and tables, errors, logging, testing, networking, and packaging.

Here is a preview of a session with the finished tool. Do not worry about how any of it works yet; that is what the rest of the book is for.

Terminal

```
$ ledgerly add 54.23 "Groceries" --category food --date 2026-03-02
Added expense #1: $54.23 Groceries (food, 2026-03-02)
$ ledgerly add 45.00 "Bus pass" -c transport -d 2026-03-03
Added expense #2: $45.00 Bus pass (transport, 2026-03-03)
$ ledgerly budget set food 300
Budget for food set to $300.00 per month
$ ledgerly list --month 2026-03 --limit 2
 ID  Date        Category   Amount  Description
  1  2026-03-02  food       $54.23  Groceries
  2  2026-03-03  transport  $45.00  Bus pass
$ ledgerly show 99
Error: expense #99 not found.
$ echo $?
1
```

Notice a few design choices already visible in that session. Commands are verbs (`add`, `list`, `show`). Common options have short aliases (`-c`, `-d`). Success messages are brief and confirm exactly what happened. The error message is short, clear, and comes with a non-zero exit code. Money is always shown with two decimal places. These details will not happen by accident; we will design each of them deliberately.

The complete command set

By the end of the book, Ledgerly will support the following commands. You will see this list again when we plan the tool in Chapter 5, so there is no need to memorize it now.

Command	Purpose
`ledgerly add`	Record a new expense with amount and description.
`ledgerly list`	Show expenses, filtered by month or category.
`ledgerly show` / `edit` / `delete`	Work with a single expense by its ID.
`ledgerly budget set` / `show`	Set and check monthly category budgets.

`ledgerly report monthly` / `categories`	Summaries of spending over time.
`ledgerly import` / `export`	Move data in and out as CSV or JSON.
`ledgerly config show` / `set`	View and change settings.
`ledgerly convert`	Convert an amount between currencies.

Ledgerly will also offer global options that work with every command: `--version`, `--verbose` (`-v`, repeatable), `--quiet` (`-q`), `--no-color`, and `--db PATH` to point at a different database file.

How money is handled

One design decision is so important that we will state it now and repeat it often: **Ledgerly stores money as whole numbers of cents, never as floating-point numbers.** The amount $54.23 is stored as the integer `5423`. Floating-point numbers cannot represent most decimal fractions exactly, which leads to small rounding errors that add up in financial totals. Integers do not have that problem. You will see this idea in Chapter 4 when we design the `Expense` data model and again in Chapter 18 when we parse amounts carefully.

The Journey Ahead

We will grow Ledgerly the way real software grows: in small, working steps. Each part of the book adds a layer.

The Ledgerly roadmap

```
Part I    Foundations        10-line script  ->  a planned project layout
Part II   Arguments & Output argparse        ->  Typer commands, Rich tables
Part III  Working with Data  JSON file       ->  SQLite, config, import/export
Part IV   Robustness         friendly errors, logging, validation, pytest
Part V    Polish & Power     help, completion, web API, speed, plugins
Part VI   Shipping           packaging, PyPI, CI, release 1.0
```

Part I ends with a clear plan and project layout for Ledgerly, and in Chapter 3 you will write the very first version: a ten-line script that appends expenses to a text file. At every stage, Ledgerly works. You will never have to wait ten chapters before you can run something. Each chapter ends with exercises in a "Try It Yourself" section and a short list of key takeaways.

Type the code yourself rather than copying it; the small mistakes you make and fix along the way are some of the most valuable lessons.

> **NOTE**
>
> Terminal output in this book is shown realistically but trimmed for space. Your version numbers, file paths, and timestamps will differ. Where output depends on your operating system, we show both versions or point out the difference.