





```
for card in soup.select(
  "li.product-card"):
  sku = card["data-sku"]
  price = parse_price(
    card.select_one(".amount").text)
```

1 robots.txt + fetch

2 parse + clean

3 validate + store

4 detect changes

200 OK · Crawl-delay: 1 · 429 → Retry-After: 5 · 30 products · 0 errors

A RESPONSIBLE, PRACTICAL GUIDE

# Web Scraping and Data Collection with Python

## Requests, httpx, Beautiful Soup, APIs, and responsible, reliable data pipelines

HTTP · Sessions · Retries · Caching · CSS & XPath · Crawling
Logins · JSON APIs · Async · SQLite · Testing · Scheduling

FETCH · PARSE · CLEAN · STORE · MONITOR

THENEXBOOKS

# Web Scraping and Data Collection with Python

*Requests, httpx, Beautiful Soup, APIs, and Responsible, Reliable Data Pipelines*
*From your first request to a scheduled, tested pipeline that respects the sites it visits*

First Edition

THENEXBOOKS

**Web Scraping and Data Collection with Python**
Requests, httpx, Beautiful Soup, APIs, and Responsible, Reliable Data Pipelines

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, including the Cobblestone Market demo site and the cobblescrape package, 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. Nothing in this book is legal advice. Laws, website terms, and platform rules differ between countries and change over time; before collecting data from any website you do not own, read its terms, prefer an official API, ask for permission where appropriate, and consult a qualified lawyer for real projects. Software libraries change between versions; consult their documentation for the version you use.

Python is a registered trademark of the Python Software Foundation. Windows is a trademark of the Microsoft group of companies. Other product and project names mentioned in this book, including requests, httpx, aiohttp, Beautiful Soup, lxml, pandas, DuckDB, SQLite, Playwright, Selenium, and Scrapy, are trademarks or names of their respective owners or projects. Their use is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of them.

Cobblestone Market, Hollowbrook Analytics, and every maker, product, review, reviewer, and account used as the running example in this book are fictional. The Cobblestone Market website is a demo program included with the book that runs only on your own computer. Email addresses use reserved example domains.

# Preface

Sooner or later, almost every analyst and developer needs data that lives on a web page. A product team wants to track prices across a catalog, a researcher needs a few thousand public records that no one has packaged as a download, an operations team wants to know the moment a listing changes. Copying and pasting works for ten rows. It does not work for ten thousand rows every Monday morning.

Web scraping is the craft of turning web pages into clean, structured data with code. Python is the most popular language for it, with excellent tools: requests and httpx to fetch pages, Beautiful Soup and lxml to read HTML, asyncio to work concurrently, and pandas, SQLite, and DuckDB to store and explore the results. But good scraping is more than tools. A scraper that ignores robots.txt, hammers a server, collects personal data carelessly, or silently breaks when a page changes is a liability, not an asset.

This book teaches both halves: the techniques and the responsibility. Every example runs against **Cobblestone Market**, a realistic demo shop that comes with the book and runs on your own computer, complete with pagination, reviews, a login area, a JSON API, a JavaScript-rendered page, robots.txt rules, and a real rate limit. You can experiment freely, make mistakes, and see real responses without bothering anyone.

## Who This Book Is For

- **Analysts and data professionals** who know some Python and need reliable, repeatable data collection instead of copy and paste.
- **Developers** who must build or maintain scrapers, crawlers, or data-collection jobs and want them to be polite, robust, and testable.
- **Researchers and students** who collect public web data for projects and want to do it ethically and reproducibly.
- **Technical leads** who need to understand what a responsible scraping pipeline looks like before approving one.

You should be comfortable with basic Python: variables, functions, lists and dictionaries, loops, and installing packages with pip. You do not need prior experience with HTTP, HTML parsing, or asynchronous programming.

## What You Will Learn

- When scraping is the right tool, when an API or download is better, and how to collect data responsibly: robots.txt, terms of use, rate limits, and personal data.
- HTTP from a scraper's point of view: methods, status codes, headers, cookies, redirects, timeouts, and caching.

- Fetching pages with requests and httpx: sessions, headers, retries with backoff, Retry-After, and conditional requests with ETags.
- Extracting data with Beautiful Soup, CSS selectors, XPath and lxml, plus JSON-LD, embedded JSON, and HTML tables.
- Crawling: pagination, sitemaps, link following, deduplication, and avoiding crawl traps.
- Forms and logins on accounts you own, JSON APIs as a better alternative, and how to approach JavaScript-rendered pages.
- Async scraping with httpx and aiohttp, with concurrency limits and rate limiting that keep you a good guest.
- Cleaning, validating, and storing data in CSV, JSON Lines, and SQLite, and exploring it with pandas and DuckDB.
- Production skills: testing scrapers with fixtures, logging, scheduling, monitoring, change detection, and a complete pipeline.

## How This Book Is Organized

**Part I — Foundations** (Chapters 1–5) explains what scraping is, how to do it responsibly, sets up your tools and the demo site, and teaches the HTTP and HTML you need. **Part II — Fetching Pages** (Chapters 6–10) builds your first scraper and makes fetching robust with sessions, httpx, retries, and caching. **Part III — Parsing and Extracting** (Chapters 11–15) covers Beautiful Soup, CSS selectors, XPath, crawling with pagination and sitemaps, and structured data inside pages. **Part IV — Beyond Static Pages** (Chapters 16–20) handles logins, JSON APIs, JavaScript-rendered pages, and polite asynchronous scraping. **Part V — From Raw Data to Clean Datasets** (Chapters 21–24) cleans, validates, stores, and explores what you collect. **Part VI — Production Pipelines** (Chapters 25–28) adds testing, scheduling, monitoring, and change detection, and finishes with the complete pipeline. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

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

## The Running Example: Cobblestone Market

Throughout the book you work for **Hollowbrook Analytics**, a fictional market-research firm. Hollowbrook has written permission from **Cobblestone Market**, a fictional online shop of handmade goods, to collect its public catalog for a weekly price-and-stock report called **Market Pulse**. Cobblestone Market sells thirty products from eight makers across five categories, from a $8.50 soap bar to a $1,240.00 walnut coffee table, with sale prices, stock levels, reviews, and a wholesale price list behind a login.

The shop is a small Python program included with the book, `cobblestone_site.py`, which runs on your own computer at `http://127.0.0.1:8800`. It behaves like a real site: it paginates, redirects, sets cookies, publishes robots.txt and a sitemap, answers with 429 when you go too fast, fails now and then, and can jump "one week ahead" so you can practice change detection. Chapter by chapter we build **cobblescrape**, a polite, tested scraper package, and finish with a scheduled pipeline that crawls the shop, cleans and validates the data, stores it in SQLite with price history, and reports exactly what changed. Because the data is small and fixed, every output in this book can be reproduced and checked.

## Versions and Conventions

Examples use **Python 3.12** or newer and recent versions of requests, httpx, aiohttp, Beautiful Soup, lxml, pandas, DuckDB, Typer, and pytest. Everything was run on the demo site and the output shown is real; timings and timestamps will differ on your machine. Browser automation with Playwright is described but not required. Commands are shown as `python ...`; on Windows you can also use `py`. Chapter 3 shows how to make Windows consoles print accented characters correctly.

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

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

> **WARNING**
> A common mistake, or something that can cause errors, data loss, or harm to the sites you visit.

Code listings are captioned with their file name, such as `cobblescrape/parse.py`. Program output appears in a separate block captioned **Output**. Long lines are wrapped to fit the page. **Practice only on the demo site or on sites you have permission to collect from.**

# Contents

Preface

**PART I — FOUNDATIONS**

1 Web Scraping Today: What It Is, When to Use It, and the Cobblestone Project

2 Scraping Responsibly: robots.txt, Terms, Rate Limits, and Personal Data

3 Setting Up: Python, Packages, and the Cobblestone Market Demo Site

4 HTTP for Scrapers: Requests, Responses, Status Codes, and Headers

5 HTML and the DOM: Reading Pages the Way a Scraper Does

**PART II — FETCHING PAGES**

6 Your First Scraper with requests

7 Sessions, Headers, Cookies, and Redirects

8 httpx: A Modern HTTP Client, Timeouts, and Connection Reuse

9 Errors, Retries, and Backoff: Handling 429s, 5xx, and Flaky Networks

10 Caching and Conditional Requests: ETags, Last-Modified, and Local Caches

**PART III — PARSING AND EXTRACTING**

11 Beautiful Soup Fundamentals: Navigating and Searching the Tree

12 CSS Selectors in Depth: Extracting Product Pages Reliably

13 XPath and lxml: Precise Extraction at Speed

14 Pagination and Crawling: Following Links, Sitemaps, and Avoiding Traps

15 Structured Data in Pages: JSON-LD, Embedded JSON, and HTML Tables

**PART IV — BEYOND STATIC PAGES**

16 Forms and Logins: CSRF Tokens, Sessions, and Accounts You Own

17 JSON APIs vs Scraping: Finding and Using the API First

18 JavaScript-Rendered Pages: Detecting Them, Finding the Data, and Browser Automation

19 Async Scraping with httpx and asyncio: Concurrency with Politeness

20 aiohttp and Rate-Limiting Patterns: Semaphores, Token Buckets, and Budgets

**PART V — FROM RAW DATA TO CLEAN DATASETS**

21 Cleaning Scraped Data: Text, Prices, Dates, Units, and Encodings

22 Modeling and Validating Records: Dataclasses, Types, and Quality Checks

23 Storing Results: CSV, JSON Lines, and SQLite

24 Exploring Scraped Data with pandas and DuckDB

**PART VI — PRODUCTION PIPELINES**

25 Testing Scrapers: Fixtures, Offline Tests, and Catching Layout Changes

26 Scheduling, Logging, and Monitoring Scrapers

27 Change Detection and Price History

28 Project: The Complete Cobblestone Data Pipeline

**APPENDICES**

A Web Scraping Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

# Foundations

What web scraping is and when to use it, how to collect data responsibly with robots.txt, terms, rate limits, and care for personal data, setting up Python and the Cobblestone Market demo site, and the HTTP and HTML every scraper depends on.

1 · Web Scraping Today

2 · Scraping Responsibly

3 · Setting Up

4 · HTTP for Scrapers

5 · HTML and the DOM

CHAPTER 1 · PART I — FOUNDATIONS

# Web Scraping Today: What It Is, When to Use It, and the Cobblestone Project

*Before we write a single request, we need a clear picture of what web scraping is, when it is the right tool, and when it is not. This chapter gives you that picture, introduces the fictional shop and research firm we work with for the rest of the book, and shows you where the journey ends: a polite, tested data pipeline that tells you exactly what changed in a catalog from one week to the next.*

---

## What We Mean by Web Scraping

Almost every website you visit is built for human eyes. A browser downloads some HTML, a stylesheet, perhaps some JavaScript and images, and paints a page that a person can read. The information on that page, though, is often useful to more than one reader. A shop's prices, a library's opening hours, a city's list of planning applications, a research group's publication list: all of it is data, but it arrives wrapped in layout, navigation, and decoration that were designed for people, not programs.

**Web scraping** is the practice of writing a program that fetches those pages and pulls the data back out of them in a structured form, such as rows in a CSV file or records in a database. The program does what you would do by hand with copy and paste, but it does it consistently, quickly, and as often as you need.

Several related terms float around, and it helps to keep them apart from the start:

- **Fetching** (or downloading) is the network part: sending an HTTP request and receiving a response. Libraries such as `requests` and `httpx` do this for us.
- **Parsing** (or extraction) is turning the response body, usually HTML or JSON, into Python values. Beautiful Soup and lxml do the heavy lifting for HTML; the standard library's `json` module handles JSON.
- **Crawling** is the act of discovering pages by following links: from a catalog page to the next catalog page, from a listing to each product, from a sitemap to every URL it names. A crawler decides *what* to fetch next; a scraper decides *what to keep* from each page. Most real projects need a little of both.
- **Data collection** is the wider umbrella. It covers scraping, but also calling official APIs, downloading published data files, reading feeds, and anything else that gets external data into your hands in a repeatable way.

This book is called *Web Scraping and Data Collection with Python* for a reason. Scraping HTML is one technique among several, and frequently it is not the best one. A good data collector knows all the options and picks the one that is cheapest, most reliable, and most respectful of the site that provides the data.

## Why People Scrape: Common Use Cases

Scraping shows up in more places than you might expect. A few patterns account for most of the projects we see in practice:

- **Price and stock monitoring.** Retailers, suppliers, and analysts track how prices and availability move over time. The interesting signal is usually the *change*: a price that went up, a sale that started, a product that sold out or came back.
- **Research.** Social scientists, journalists, and public-interest groups collect public records, announcements, or listings to study trends that nobody has published as a ready-made dataset.
- **Catalog aggregation.** A comparison site or an internal tool brings together product details from several sources, with permission, into one consistent format.
- **Change alerts.** Someone needs to know when a page changes: a new policy document, a new job posting, a status page that flips from green to red. A small scheduled scraper that compares today's result with yesterday's is often all it takes.
- **Internal glue.** Sometimes the "website" is an old internal system that has a web interface and no export button. Scraping your own organization's tools, with the owner's blessing, can be the quickest bridge to a modern workflow.

Notice what these use cases have in common. They need data that is *repeatable* (collected the same way every time), *traceable* (you can say where each value came from and when), and *clean* (prices are numbers, dates are dates). A one-off copy and paste is fine for a single afternoon; a weekly report that people rely on needs engineering. That engineering is what this book is about.

## The Decision Ladder: Scrape Last, Not First

When you need data from a website, it is tempting to open the browser's developer tools, find the right HTML elements, and start writing a parser. Resist that temptation for a few minutes and climb what we call the **decision ladder** instead. Each rung is usually cheaper and more stable than the one below it.

```
1. Official API        documented, typed, versioned, built for programs
      |
2. Data export         a CSV/JSON download, a bulk file, a "download data" button
      |
3. Feed or sitemap     RSS/Atom feeds, sitemap.xml: structured lists of what exists
      |
```

```
4. Scraping HTML        parse pages built for humans -- the last resort
      |
(and at every rung)     read the terms, check robots.txt, ask if unsure
```

**An official API** is the best case. It returns data in a format designed for programs, usually JSON, with stable field names and documented limits. When a site changes its visual design, the API normally stays the same, so your code keeps working. APIs also tend to be lighter for the site to serve, because they skip templates, images, and scripts.

**A data export** is the next best thing. Many organizations publish downloadable files: open-data portals, statistics offices, product feeds for partners. One download replaces hundreds of page requests.

**Feeds and sitemaps** are structured lists that sites publish so that other programs can discover their content. A sitemap tells you every URL the site wants indexed, often with a "last modified" date. That alone can save you from crawling navigation pages you do not need.

**Scraping HTML** sits at the bottom of the ladder, not because it is shameful but because it is fragile. HTML is a presentation format. A designer can rename a class, move a price into a different element, or wrap a name in a new tag, and your parser silently returns nothing. Scraping also costs the site more: each page includes markup, and often a lot of it, that you will throw away.

In this book we climb the ladder properly. The demo shop we use has an API, a sitemap, and HTML pages, and we use all three. We scrape the HTML because that is the skill most projects eventually need, and because it teaches you how the web actually works. But in Chapter 17 we will show that the API gives the same answers with fewer requests and less code, and we will discuss when that should change your design.

> **TIP**
>
> Before writing any parser, spend ten minutes looking for an easier route. Search the site for "API", "developers", "data", or "download". Look at `/robots.txt` for a `Sitemap:` line. Open your browser's Network panel while the page loads and see whether the data arrives as JSON from a separate URL. Those ten minutes regularly save days.

## How a Scraper Works

Every scraper in this book, from the ten-line script in the next section to the final pipeline, follows the same basic loop. It is worth seeing the whole loop now, so that each later chapter has a place to fit.

```
        +-----------+     +-----------+     +-----------+     +-----------+
URL --> |   FETCH   | --> |   PARSE   | --> |   CLEAN   | --> |   STORE   |
        | HTTP GET  |     | HTML/JSON |     | text,     |     | CSV,      |
        | politely  |     | to fields |     | prices,   |     | SQLite    |
        +-----------+     +-----------+     | dates     |     +-----------+
              ^                 |           +-----------+           |
```

```
                |   next page /   |                                  v
                +--- next link ---+                       compare with last run,
                                                          report, repeat next week
```

1. **Fetch.** Send an HTTP request for a URL and receive a response with a status code, headers, and a body. Fetching politely means identifying yourself, honoring the site's rules, waiting between requests, setting timeouts, and backing off when the server says it is busy. Part II of the book covers this stage.
2. **Parse.** Turn the response body into fields. For HTML, that means navigating the document tree and selecting elements with CSS selectors or XPath. For JSON, it means reading keys. Part III covers parsing.
3. **Discover more URLs.** Many pages lead to other pages: a "Next" link, a list of products, a sitemap. The crawler adds those to a queue, avoiding duplicates and staying within the site's allowed areas.
4. **Clean.** Raw text is messy. A price arrives as `"$1,240.00"`, a name is surrounded by newlines and an invisible non-breaking space, a date reads "December 22, 2025". Cleaning turns these into a `Decimal`, a tidy string, and a `date`. Part V covers cleaning, validation, and storage.
5. **Store.** Save the results in a durable, queryable form: CSV for sharing, SQLite for history.
6. **Repeat and compare.** The real value often comes from running the same scraper on a schedule and comparing results over time. Part VI covers scheduling, monitoring, testing, and change detection.

Each stage can fail in its own way, and a production scraper has to cope with all of them: a timeout during fetching, a missing element during parsing, an unexpected currency symbol during cleaning, a locked file during storage. Much of the craft lies in handling those failures calmly and visibly rather than crashing, or worse, quietly writing wrong data.

## A First Taste

Let us make the loop concrete with a tiny example. Do not worry about setting it up yet; Chapter 3 walks you through installing Python, the libraries, and the demo site, and Chapter 6 builds a proper first scraper step by step. For now, just read the code and the output and notice how little is needed to get started.

scripts/first_look.py

```
"""A first taste: fetch one catalog page politely and list what is on it."""
import requests
from bs4 import BeautifulSoup

BASE_URL = "http://127.0.0.1:8800"
USER_AGENT = "cobblescrape/1.0 (+mailto:data@hollowbrook.example)"

resp = requests.get(f"{BASE_URL}/catalog/", headers={"User-Agent": USER_AGENT},
                    timeout=10)
```

```
resp.raise_for_status()

soup = BeautifulSoup(resp.text, "lxml")
print(soup.select_one("p.result-count").get_text(strip=True))
for card in soup.select("li.product-card")[:5]:
    sku = card["data-sku"]
    name = " ".join(card.select_one("h2.product-name").get_text().split())
    price = card.select_one("span.amount").get_text(strip=True)
    print(f"{sku}  {name:<26} {price:>8}")
```

Output

```
Showing 1–12 of 30 products
CM-POT-001  Speckled Stoneware Mug       $24.00
CM-POT-002  Glazed Serving Bowl          $48.00
CM-POT-003  Salt & Pepper Cellar Set     $32.00
CM-POT-004  Celadon Teapot               $54.40
CM-POT-005  Stoneware Dinner Plate       $29.50
```

Even this small script already contains the habits we will keep throughout the book. It sends a **User-Agent** header that says who we are and how to contact us. It passes a **timeout**, because the `requests` library will otherwise wait forever for a server that never answers. It calls `raise_for_status()`, so that an error page is not mistaken for data. And it selects elements by stable, meaningful hooks, the `data-sku` attribute and classes such as `product-name` and `amount`, rather than by their position on the page.

Look at the price of the Celadon Teapot: $54.40. That is the sale price; the regular price of $68.00 sits next to it in a separate element that this script ignores. A real scraper must decide which price it wants and record both. That kind of detail, invisible in a quick demo, is exactly what turns into wrong numbers in a report if nobody thinks about it. We return to it in Chapter 12.

Now look at the same five products fetched from the demo shop's JSON API instead of its HTML:

scripts/first_look_api.py

```
"""The same five products from the site's JSON API: no HTML parsing needed."""
import requests

BASE_URL = "http://127.0.0.1:8800"
USER_AGENT = "cobblescrape/1.0 (+mailto:data@hollowbrook.example)"

resp = requests.get(f"{BASE_URL}/api/v1/products", params={"per_page": 5},
                    headers={"User-Agent": USER_AGENT}, timeout=10)
resp.raise_for_status()
payload = resp.json()
print(payload["meta"])
for item in payload["data"]:
    price = item["sale_price"] or item["price"]
    print(f'{item["sku"]}  {item["name"]:<26} {"$" + price:>8}  stock {item["stock"]}')
```

Output

```
{'page': 1, 'per_page': 5, 'total': 30, 'total_pages': 6}
CM-POT-001  Speckled Stoneware Mug       $24.00  stock 35
CM-POT-002  Glazed Serving Bowl          $48.00  stock 12
CM-POT-003  Salt & Pepper Cellar Set     $32.00  stock 3
CM-POT-004  Celadon Teapot               $54.40  stock 8
CM-POT-005  Stoneware Dinner Plate       $29.50  stock 0
```

The API version needs no HTML parser at all. It also gives us more: an exact stock count (the HTML card only says "In stock" or "Only 3 left"), and a `meta` block that tells us how many products and pages exist, so we never have to guess. This is the decision ladder in action. If the site you need offers an API, and its terms let you use it for your purpose, it is usually the better choice.

> **NOTE**
>
> The demo site lives at `http://127.0.0.1:8800`, which is your own computer. Every example in this book talks only to that address. You will never be asked to point code at a real website while learning, and you should not: practicing on someone else's production server costs them money and can break their terms of use.

## Static Pages and JavaScript Pages

The catalog page in our example is a **static** page in the sense that matters to scrapers: the HTML the server sends already contains the product names and prices. Fetch it with any HTTP client, and the data is there in the response body.

Many modern sites work differently. The server sends a nearly empty HTML "shell" plus a JavaScript program. The browser runs that program, which requests the real data, typically JSON from an API URL, and then builds the visible page. If you fetch such a page with `requests`, you get the shell and nothing else, because `requests` does not run JavaScript.

The demo shop has one such page, `/deals/`, on purpose. A browser shows a list of five current deals; a plain HTTP client sees only the text "Loading deals..." where the list should be. In Chapter 18 we investigate it properly. The good news, which you will see there, is that a JavaScript page almost always gets its data from somewhere you can reach directly: JSON embedded in the page itself, or the API URL the script calls. Driving a full browser with a tool such as Playwright is the last option, not the first, because it is slower and heavier for both you and the site.

So the decision ladder applies inside a single page, too. Before reaching for a browser, look for the data in its most structured form.

# Meet the Project: Hollowbrook, Market Pulse, and Cobblestone Market

Learning scraping with disconnected toy snippets has a problem: nothing ever adds up to a working system, and the hard parts (retries, pagination, cleaning, change detection) never appear. So this book follows one realistic project from the first request to a scheduled pipeline. All of the people and businesses in it are fictional.

## Cobblestone Market

**Cobblestone Market** is an online shop for handmade goods: pottery, candles and soap, textiles, woodcraft, and paper goods from a handful of small makers such as Hearth & Kiln, Wick & Whistle, and Loom & Larch. On day one of our story it lists 30 products across five categories. Its pages look like the pages of many small shops: a paginated catalog, category pages, product pages with a specification table and customer reviews, a login area for wholesale partners, a JSON API, a sitemap, and a robots.txt file.

The shop also contains, deliberately, many of the small annoyances that make real scraping hard. Product names are wrapped in stray whitespace, and four of them hide a non-breaking space that looks exactly like a normal space. One product's specification table is missing its weight row. One price has a thousands separator. One product name has accented letters that crash a Windows console set to the wrong code page. The server enforces a rate limit and answers too-eager clients with status 429. One API endpoint fails twice before it succeeds. The deals page needs JavaScript. A week after our first visit, prices change, one product disappears, and a new one arrives. Every one of these details is there to teach a specific lesson, and each lesson appears in the chapter where it belongs.

## Hollowbrook Analytics and Market Pulse

We play the role of **Hollowbrook Analytics**, a small market-research firm. Hollowbrook has Cobblestone Market's **written permission** to collect its public catalog for a weekly price-and-stock report called **Market Pulse**. The shop's terms of use, which you can read on the demo site at `/terms/`, allow automated access for learning purposes if you identify your client, respect robots.txt, and keep to one request per second.

That framing is not decoration. Real scraping projects begin with the question "are we allowed to do this, and under what conditions?", and the answer shapes the code. Because the terms require us to identify ourselves, every request we send carries this User-Agent:

```
cobblescrape/1.0 (+mailto:data@hollowbrook.example)
```

It names our tool and version and gives an email address where the shop can reach us if our scraper misbehaves. Because the terms ask for one request per second, our scraper waits one second between requests, even though the demo site would allow short bursts. Because the site publishes a robots.txt file, we read it and obey it. Chapter 2 explains each of these rules and why they matter.

### Why a local demo site?

The demo shop is a small Python program, `cobblestone_site.py`, that ships with this book and uses only the standard library. You start it on your own computer and it listens on `127.0.0.1`, an address that only your machine can reach. There are good reasons to learn this way:

- **It is safe and fair.** You cannot overload or annoy anybody's real server, no matter what mistakes you make while learning. Mistakes are part of learning, and the place to make them is your own machine.
- **It is stable.** Real websites change their layout without notice. A book that scrapes a live site is out of date the day it is printed. The demo site produces exactly the same pages every time, so the outputs you see in this book are the outputs you will get.
- **It is complete.** The demo includes situations that are hard to find safely in the wild: rate limiting, flaky endpoints, logins with CSRF tokens, conditional requests, and changes between two snapshots. You can switch them on and off with command-line options.
- **It works offline.** Once you have Python and the libraries installed, you need no internet connection at all.

Everything you learn transfers directly to real sites. The HTTP is real HTTP, the HTML is ordinary HTML, and the problems are the problems you will meet elsewhere. The only difference is that here, permission is guaranteed and the data never changes behind your back unless you ask it to.

> **WARNING**
>
> The fact that a technique works against the demo site does not mean you may use it against any site you like. Every real target needs its own check of terms, robots.txt, and the kind of data involved, as Chapter 2 explains. When in doubt, ask the site owner, and for anything with legal weight, ask a lawyer.

## What We Build: The cobblescrape Package

Over the course of the book we build a Python package called **`cobblescrape`**. It starts in Chapter 6 as a single script that prints product names, and grows chapter by chapter into a small, well-organized project. By the end it looks like this:

```
cobblestone-scraper/
  cobblescrape/
    config.py      settings from environment variables, our User-Agent
    robots.py      read and obey robots.txt
    fetch.py       PoliteSession: delays, retries, backoff, caching
    parse.py       turn HTML into ProductSummary, Product, Review objects
    clean.py       prices, stock text, weights, dates, whitespace
    crawl.py       follow pagination, visit every product
    api.py         the JSON API route to the same data
    account.py     log in to the wholesale area we are allowed to see
    aio.py         concurrent fetching with strict rate limits
```

```
    models.py      typed records (dataclasses)
    validate.py    quality checks before anything is stored
    store.py       CSV export and SQLite history
    changes.py     what changed since the last run?
    pipeline.py    crawl -> parse -> clean -> validate -> store -> compare
    cli.py         the command-line interface
  tests/           offline tests with saved HTML fixtures
  data/            cobblestone.db, products.csv, cache/
```

The final pipeline is run with one command and prints a short, honest summary. Here is what Market Pulse looks like on the second weekly run, after the shop's catalog has changed. This is the exact output you will reproduce yourself in Chapter 27 and Chapter 28:

Output

```
run 2: ok - 30 products, 33 requests, 0 retries, 0 errors
7 changes:
  removed      CM-WOD-005  Carved Cherry Spoon              listed -> gone
  sale_started CM-CND-003  Cedar Smoke Candle               $18.00 -> $14.40
  added        CM-PAP-007  Cobblestone Street Calendar           - -> $24.00
  price        CM-POT-001  Speckled Stoneware Mug           $24.00 -> $26.00
  sale_ended   CM-POT-004  Celadon Teapot                   $54.40 -> $68.00
  stock        CM-POT-005  Stoneware Dinner Plate                0 -> 24
  stock        CM-TXT-002  Hand-Woven Wool Throw                  4 -> 0
```

Read that output as the client would. In one week, a product was withdrawn, a new product appeared, one regular price went up by two dollars, one sale started and another ended, one sold-out item was restocked, and a nearly sold-out item ran out. That is the kind of insight Market Pulse exists to deliver, and every number in it is traceable back to a specific page fetched at a specific time.

Look at the first line, too. Thirty products took 33 requests: three catalog pages plus thirty product pages. At one request per second, the whole crawl takes a little over half a minute. There were no retries and no errors. A scraper that reports its own behavior like this is one you can trust, schedule, and defend if the site owner ever asks what you are doing.

## A Tour of the Book

The book has six parts that follow the scraper loop from the ground up. You can read it straight through, which is how the project grows, or jump to a part when you need it. Each chapter starts the demo site, runs real code against it, and shows the real output.

| Part | Chapters | What you learn |
|---|---|---|
| I. Foundations | 1–5 | Responsible scraping, setup, HTTP, HTML and the DOM |
| II. Fetching Pages | 6–10 | requests, sessions, httpx, retries and backoff, caching |
| III. Parsing and Extracting | 11–15 | Beautiful Soup, CSS selectors, XPath, crawling, structured data |
| IV. Beyond Static Pages | 16–20 | Logins, JSON APIs, JavaScript pages, async with rate limits |

| | | |
|---|---|---|
| V. Raw Data to Clean Datasets | 21–24 | Cleaning, validation, CSV and SQLite, a first look with pandas |
| VI. Production Pipelines | 25–28 | Testing, scheduling and monitoring, change detection, the final project |

Part I, which you are reading now, lays the foundation: responsible scraping in Chapter 2, setup in Chapter 3, and HTTP and HTML from a scraper's point of view in Chapters 4 and 5. Part II gets pages reliably, with sessions, retries, and caching. Part III turns pages into data with three kinds of selectors and teaches safe crawling. Part IV handles logins, APIs, JavaScript pages, and polite concurrency. Part V cleans, validates, and stores the data, and Part VI makes the whole thing tested, scheduled, monitored, and able to report changes.

The appendices collect a cheat sheet, templates, a production checklist, and a glossary for quick reference once you have finished the book.

## Who This Book Is For

We assume you already write basic Python: functions, lists and dictionaries, a few classes, and installing packages with `pip`. You might be an analyst who needs data that nobody has packaged for you, a developer asked to "just grab the prices from that site", or a researcher who wants a repeatable collection process instead of a folder of copied spreadsheets.

We do not re-teach Python basics, and we do not go deep into data analysis or data warehousing; other books cover those well. When we use pandas or DuckDB, we keep it to a first look at the data we collected. Our focus is the collection itself: getting the right data, from the right source, politely, reliably, and repeatably.

## What This Book Is Not

It is just as important to say what we will *not* teach.

- **No evasion.** We do not teach how to get around blocks, CAPTCHAs, bot detection, or rate limits. We do not rotate proxies to hide, disguise our client as a regular browser, or spoof browser fingerprints. If a site does not want automated access, or slows you down, the right responses are to slow down further, to ask, or to use an official channel. Chapter 2 explains why, and every later chapter follows that rule.
- **No scraping of real sites.** Every example targets the local demo site. We may mention well-known resources in passing, but never as targets.
- **No legal advice.** We discuss the legal and ethical landscape in general terms so that you know which questions to ask. For a real project, you check the site's terms, prefer an official API, ask for permission where needed, and consult a lawyer when the stakes justify it.

- **No mass harvesting of personal data.** The demo shop's reviews include fictional reviewer names, and we treat them the way you should treat real personal data: collect only what the analysis needs, do not republish it, and pseudonymize it if you keep it.

These limits are not there to make the book less useful. In our experience, the scrapers that last are the ones that a site owner would be comfortable seeing in their access logs. Politeness and reliability turn out to be the same engineering: a scraper that waits, retries carefully, caches, and identifies itself is also one that rarely breaks, rarely gets blocked, and is easy to explain to a client, a manager, or the site itself.

## How to Read the Code in This Book

A few conventions will make the examples easier to follow:

- Commands are shown for a generic shell, for example `python -m pip install ...`. On Windows, `py -3.12` or `py` works in place of `python`; Chapter 3 explains this along with virtual environments. Where PowerShell syntax differs, such as setting environment variables, we show both forms.
- Files are labeled with their path in the project, such as `scripts/first_look.py` or `cobblescrape/fetch.py`. Program output always appears in a separate block labeled **Output**, and it is real output from running the code against the demo site.
- The demo site always appears at `http://127.0.0.1:8800`. If you run it on a different port, change the URL in the code to match.
- Early chapters sometimes show a simpler version of a function that a later chapter extends; we say so when that happens.

The book's download includes the demo site and a reference copy of the finished `cobblescrape` package with its tests. Use it when you get stuck, but try to type the code yourself as you go.

## Try It Yourself

You do not need any code for these exercises; they prepare your thinking for the chapters ahead.

1. Pick three kinds of data you might want to collect for your own work. For each one, walk down the decision ladder: is there an official API, a data export, a feed, or a sitemap? Write down which rung you would most likely end up on, and why.
2. Using the first output in this chapter, list every field you can see for each product. Then list the fields you would want for a price-and-stock report. Which ones are missing from the listing page and would need a visit to each product page or a call to the API?
3. Read the Market Pulse change list again. For each of the seven changes, write one sentence that a non-technical client would want to read in the weekly report.