I'm not a bot

Interested in learning how to build a web scraping tool but not sure what coding languages you need to master, or how to implement one? If so, you're not alone. There are thousands of new and intermediate developers who are looking to learn how to code useful, efficient web scrapers using coding languages they already know. Why is there so much interest in this skill? Because knowing how to code a web scraper, or data extraction tool, is highly prized in the tech job market. With so much data readily available online, companies in a variety of industries need a reliable, cost-effective – and again, efficient – way to get to the structured and semi-structured data that they need for research, competitive analysis, debugging, etc. From ecommerce to real estate to stock trading, web scraping tools can provide updated data in near real-time for businesses as well as their developers. If you know how to code a scraping tool, you'll likely have no shortage of job opportunities. Free web scraping guide. Download our beginner-friendly web scraping guide [PDF] and learn how to collect and use publicly available data. Luckily, there are plenty of online courses to learn how to code a web scraping script or app. From beginners to professional developers, here are the best courses to learn the mechanics behind web scraping fast (and without leaving your house!) That said, each of the courses below does require at least basic knowledge of more advanced coding languages like Python or Javascript, so if you're still learning the basic front-end languages, it's best to take intro courses in more advanced languages before diving into any of the options below. Pre-requisites: Basic Python Knowledge Cost: $-$$* This easy-to-follow course for beginners teaches the basics of web scraping with Python. In addition to getting acquainted with Beautiful Soup and how to work with static files, you'll also learn a couple of practical projects. If you're looking for a user-friendly course on how to build out simple email automation scripts, or extract data from an existing site, you'll want to check out this highly-rated Udemy course. *Udemy offers several limited-time-only discount periods per year, so you can get 50-80% discounts on course prices if you time it right. This course is rated 9.2/10 on Coursemarks Pre-requisites: Basic Python knowledge, Python 3, Anaconda distribution Cost: $-$$ Slightly more advanced, this best-selling course also covers the fundamentals of web scraping with Python – but with greater attention paid to some of the more challenging situations you might run into in deploying a web scraper. Topics like scraping 'gated' or 'locked' data, implementing APIs, scraping Javascript and limiting the number of requests are all hurdles that you'll likely run into in real-world scraping scenarios. As with the Beginners course above, the focus here is not only on teaching the fundamentals, but also encourages students to use their newfound skills with practical, in-course scraping projects. Pre-requisites: part of the Python 3 Programming Specialization, so Python Basics, and Python Functions, Files and Dictionaries courses are recommended Cost: Free – $$* This course from the University of Michigan will teach you how to create and use data extraction across a host of different use cases. U of M is recognized for high-quality course offerings for developers, providing both absolute beginners and professionals with practical coding education. You'll learn about complex data structures, how to process json data, and caching requested data – in addition to a range of other topics that will help you master data collection with Python. *Most Coursera courses – this one included – can be taken for free after filling out a short Financial Aid application. If you don't need a Certificate of Completion, this is a cost-effective way to learn new skills that will help your career. Pre-requisites: Intermediate Python Cost: $-$$* In 17 video modules that span 4 hours, this course is a deep dive into the tools and technologies that underpin data scraping. With a greater emphasis on understanding the structure of html and XPath notation, DataCamp's Web Scraping in Python course aims to teach users how to build scraping tools that scale. This course leans heavily on creating web scrapers with the Scrapy Python library, but learners can easily adapt their newfound knowledge to extract data with BeautifulSoup or Selenium. This is a great option for those on a data science career track, covering creating automated data scraping tools before you dive into how to import and clean extracted data in the follow-up course. *DataCamp offers several reasonable pricing options, and offers students a free month to try their platform before committing to a monthly rate. Pre-requisites: JavaScript Cost: $-$$$* If you're looking to jump into practical uses of data scraping as quickly as possible, this robust course from Packt is likely your best bet. The course modules cover a variety of real-world scenarios, including scraping from search engines, hotel and ecommerce websites – each with a different coding library, including CasparJS and Cheerio. The final module covers saving extracted data to AWS using a NodeJS server. They also offer the code bundle for the course on GitHub, so if you're already a GitHub user – and who isn't? – you have quick access to dig into the code immediately and start scraping. Packt offers a robust library of courses in a variety of formats, so if you learn better with an ebook than with video instruction, this is a good option for you. *Offers competitive pricing, and 10-day free trial is available for new users Whether you're a coding 'hobbyist' looking to expand your skills, or a professional developer adding to your resume, knowing how to create a web scraper is a useful, practical data analysis skillset in this era of Big Data. With so much information readily available online, modern companies are looking to access that data, and to use it to improve their own products and services. In learning web scraping basics, you'll help them unlock new opportunities to grow. Last Updated on June 16, 2025Let me take you back to the first time I tried to scrape a website for business data. I was sitting at my kitchen table, a cup of coffee in one hand and a half-baked Python script in the other, trying to wrangle product prices from a competitor's site. I thought, "How hard could this be?" Spoiler: I ended up with a CSV file full of empty cells and a newfound respect for anyone who claims to "just automate it with Python." Fast forward to 2025, and web scraping has become the backbone of data-driven business—fueling sales, ecommerce, marketing, and operations teams with real-time insights that would be impossible to gather manually. But here's the kicker: while Python web scraping is more powerful than ever, the landscape is shifting. The market for web scraping is booming—valued at . Nearly to drive smarter decisions. Yet, the real challenge isn't just about writing code—it's about choosing the right tool for the job, scaling up, and not losing your mind maintaining a zoo of scripts. In this ultimate guide, I'll walk you through every major Python web scraping library (with code examples), real business use cases, and why, despite my love for Python, I think no-code solutions like are the best bet for most business users in 2025. What is Python Web Scraping? A Non-Technical Introduction Let's break it down: web scraping is just a fancy way of saying "automated copy-paste." Instead of hiring an army of interns to collect product prices, contact lists, or reviews, you use software to visit web pages, extract the data you need, and spit it out into a spreadsheet or database. Python web scraping means you're using Python scripts to do this—fetching web pages, parsing the HTML, and pulling out the nuggets of information you care about. Think of it as sending a digital assistant to browse websites for you, 24/7, never needing a coffee break. The most common data types scraped by businesses? Pricing info, product details, contacts, reviews, images, news articles, and even real estate listings. And while some sites offer APIs for this, most don't—or they limit what you can access. That's where web scraping comes in: it lets you tap into publicly available data at scale, even when there's no official "download" button in sight. Why Python Web Scraping Matters for Business Teams Let's get real: in 2025, if your business isn't leveraging web scraping, you're probably leaving money on the table. Here's why: Automate Manual Data Collection: No more copy-pasting rows from competitor sites or online directories. Real-Time Insights: Get up-to-date pricing, inventory, or market trends as they happen. Scale: Scrape thousands of pages in the time it takes to microwave your lunch. ROI: Companies using data-driven strategies report . Here's a quick table of high-impact use cases: The bottom line: web scraping is the secret sauce behind smarter, faster, and more competitive business decisions. Overview: All Major Python Web Scraping Libraries (With Code Snippets) I promised you a complete tour, so buckle up. Python's ecosystem for web scraping is massive—there's a library for every flavor of scraping, from simple page downloads to full-blown browser automation. Here's the lay of the land, with code snippets for each: urllib and urllib3: The Basics of HTTP Requests These are Python's built-in tools for making HTTP requests. They're low-level, a bit clunky, but reliable for basic tasks. import urllib3, urllib3.util http = urllib3.PoolManager() headers = urllib3.util.make_headers(user_agent="MyBot/1.0") response = http.request('GET', "< ", headers=headers) print(response.status) # HTTP status code print(response.data[:100]) # first 100 bytes of content Use these if you want zero dependencies or need fine-grained control. But for most jobs, you'll want something friendlier—like requests. requests: The Most Popular Python Web Scraping Library If Python scraping had a mascot, it would be the requests library. It's simple, powerful, and handles all the HTTP heavy lifting. import requests r = requests.get("< ", headers={"User-Agent": "MyBot/1.0"}) print(r.status_code) # 200 print(r.json()) # parsed JSON content (if response was JSON) Why is it so popular? It manages cookies, sessions, redirects, and more—so you can focus on getting data, not wrestling with HTTP minutiae. Just remember: requests only fetches the HTML. To extract data, you'll need a parser like BeautifulSoup. BeautifulSoup: Easy HTML Parsing and Data Extraction BeautifulSoup is the go-to for parsing HTML in Python. It's forgiving, beginner-friendly, and works hand-in-hand with requests. from bs4 import BeautifulSoup html = "

Widget

$19.99" soup = BeautifulSoup(html, 'html.parser') title = soup.find('h2').text # "Widget" price = soup.find('span', class_='price').text # "$19.99" It's perfect for small-to-medium projects or when you're just getting started. For huge datasets or complex queries, you might want to level up to lxml. lxml and XPath: Fast, Powerful HTML/XML Parsing If you need speed or want to use XPath (a query language for XML/HTML), lxml is your friend. from lxml import html doc = html.fromstring(page_content) prices = doc.xpath("//span[@class='price']/text()") XPath lets you grab data with surgical precision. lxml is fast and efficient, but the learning curve is a bit steeper than BeautifulSoup. Scrapy: The Framework for Large-Scale Web Crawling Scrapy is the heavyweight champion for big scraping jobs. It's a full framework—think of it as Django for web scraping. import scrapy class QuotesSpider(scrapy.Spider): name = "quotes" start_urls = ["< "] def parse(self, response): for quote in response.css("div.quote"): yield { "text": quote.css("span.text::text").get(), "author": quote.css("small.author::text").get(), } Scrapy handles asynchronous requests, follows links, manages pipelines, and exports data in multiple formats. It's a bit much for tiny scripts, but unbeatable for crawling thousands of pages. Selenium, Playwright, and Pyppeteer: Scraping Dynamic Websites When you hit a site that loads data with JavaScript, you need browser automation. Selenium and Playwright are the big names here. Selenium Example: from selenium import webdriver from selenium.webdriver.common.by import By driver = webdriver.Chrome() driver.get("< ") driver.find_element(By.NAME, "username").send_keys("user123") driver.find_element(By.NAME, "password").send_keys("secret") driver.find_element(By.ID, "submit-btn").click() titles = [el.text for el in driver.find_elements(By.CLASS_NAME, "product-title")] Playwright Example: from playwright.sync_api import sync_playwright with sync_playwright() as p: browser = p.chromium.launch(headless=True) page = browser.new_page() page.goto("< >") page.wait_for_selector(".item") data = page.eval_on_selector(".item", "el => el.textContent") These tools can handle any site a human can, but they're slower and heavier than pure HTTP scraping. Use them when you have to, not just because you can. MechanicalSoup: Automates form submissions and navigation, built on top of Requests and BeautifulSoup. import mechanicalsoup browser = mechanicalsoup.StatefulBrowser() browser.open("< ") browser.select_form('form#loginForm') browser["username"] = "user123" browser["password"] = "secret" browser.submit_selected() page = browser.get_current_page() print(page.title.text) RoboBrowser: Similar to MechanicalSoup, but less maintained. PyQuery: jQuery-style HTML parsing. from pyquery import PyQuery as pq doc = pq("HelloWorld") print(doc("p.title").text()) # "Hello" print(doc("p").eq(1).text()) # "World" Requests-HTML: Combines HTTP requests, parsing, and even JavaScript rendering. from requests_html import HTMLSession session = HTMLSession() r = session.get("< >") r.html.render(timeout=20) links = [a.text for a in r.html.find("a.story-link")] Use these when you want a shortcut for forms, CSS selectors, or light JS rendering. Asyncio and Aiohttp: Speeding Up Python Web Scraping For scraping hundreds or thousands of pages, synchronous requests are just too slow. Enter aiohttp and asyncio for concurrent scraping. import aiohttp, asyncio async def fetch_page(session, url): async with session.get(url) as resp: return await resp.text() async def fetch_all(urls): async with aiohttp.ClientSession() as session: tasks = [fetch_page(session, url) for url in urls] return await asyncio.gather(*tasks) urls = ["< ", "< "] html_pages = asyncio.run(fetch_all(urls)) This approach can fetch dozens of pages at once, dramatically speeding up your scrape. Specialized Libraries: PRAW (Reddit), PyPDF2, and More PRAW: For scraping Reddit via its API. import praw reddit = praw.Reddit(client_id='XXX', client_secret='YYY', user_agent='myapp') for submission in reddit.subreddit("learnpython").hot(limit=5): print(submission.title, submission.score) PyPDF2: For extracting text from PDFs. from PyPDF2 import PdfReader reader = PdfReader("sample.pdf") num_pages = len(reader.pages) text = reader.pages[0].extract_text() Others: There are libraries for Instagram, Twitter, OCR (Tesseract), and more. If you have a weird data source, chances are someone has built a Python library for it. Comparison Table: Python Scraping Libraries Scrape data from any website using AI Let's walk through a real-world example: scraping product listings from a (hypothetical) ecommerce site, handling pagination, and exporting to CSV. import requests from bs4 import BeautifulSoup import csv base_url = "< " page_num = 1 all_products = [] while True: url = base_url if page_num == 1 else f"{base_url}/page/{page_num}" print(f"Scraping page: {url}") response = requests.get(url, timeout=10) if response.status_code != 200: print(f"Page {page_num} returned status {response.status_code}, stopping.") break soup = BeautifulSoup(response.text, 'html.parser') products = soup.find_all('div', class_='product-item') if not products: print("No more products found, stopping.") break for prod in products: name_tag = prod.find('h2', class_='product-title') price_tag = prod.find('span', class_='price') name = name_tag.get_text(strip=True) if name_tag else "N/A" price = price_tag.get_text(strip=True) if price_tag else "N/A" all_products.append((name, price)) page_num += 1 print(f"Collected {len(all_products)} products. Saving to CSV...") with open('products_data.csv', 'w', newline='', encoding='utf-8') as f: writer = csv.writer(f) writer.writerow(["Product Name", "Price"]) writer.writerows(all_products) print("Data saved to products_data.csv") What's happening here? Loop through pages, fetch HTML, parse products, collect name and price, and stop when no more products are found. Export the results to CSV for easy analysis. Want to export to Excel instead? Use pandas: import pandas as pd df = pd.DataFrame(all_products, columns=["Product Name", "Price"]) df.to_excel("products_data.xlsx", index=False) Handling Forms, Logins, and Sessions in Python Web Scraping Many sites require login or form submission. Here's how you can handle that: Using requests with a session: session = requests.Session() login_data = {"username": "user123", "password": "secret"} session.post("< ", data=login_data) resp = session.get("< ") Using MechanicalSoup: import mechanicalsoup browser = mechanicalsoup.StatefulBrowser() browser.open("< ") browser.select_form('form#login') browser["user"] = "user123" browser["pass"] = "secret" browser.submit_selected() Sessions help you persist cookies and stay logged in as you scrape multiple pages. Scraping Dynamic Content and JavaScript-Rendered Pages If the data isn't in the HTML (view source shows empty divs), you'll need browser automation. Selenium Example: from selenium.webdriver.support.ui import WebDriverWait from selenium.webdriver.support import expected_conditions as EC driver.get("< ") WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, 'stats-table'))) html = driver.page_source Or, if you can find the API endpoint that the JavaScript calls, just use requests to fetch the JSON directly—it's way faster. Exporting Scraped Data: CSV, Excel, Databases, and More CSV: Use Python's csv module (see above). Excel: Use pandas or openpyxl. Google Sheets: Use the gspread library. import gspread gc = gspread.service_account(filename="credentials.json") sh = gc.open("My Data Sheet") worksheet = sh.sheet1 worksheet.clear() worksheet.append_row(["Name", "Price"]) for name, price in all_products: worksheet.append_row([name, price]) Databases: Use sqlite3, pymysql, psycopg2, or SQLAlchemy for SQL databases. For NoSQL, use pymongo for MongoDB. Comparing Python Web Scraping to Modern No-Code Solutions: Why Thunderbit is the Top Choice in 2025 Now, let's talk about the elephant in the room: maintenance. Coding your own scrapers is great—until you need to scrape 100 different sites, each with its own quirks, and they all break the night before your big report is due. Been there, done that, got the gray hairs. That's why I'm such a fan of . Here's why it's my top pick for business users in 2025: No Coding Required: Thunderbit gives you a visual interface. Click "AI Suggest Fields," adjust the columns, hit "Scrape," and you're done. No Python, no debugging, no Stack Overflow marathons. Scales to Thousands of Pages: Need to scrape 10,000 product listings? Thunderbit's cloud engine can handle it, and you don't have to babysit a script. Zero Maintenance: If you're tracking 100 competitor sites for ecommerce analysis, maintaining 100 Python scripts is a nightmare. With Thunderbit, you just select or tweak a template, and their AI adapts to layout changes automatically. Subpage and Pagination Support: Thunderbit can follow links to subpages, handle pagination, and even enrich your data by visiting each product's detail page. Instant Templates: For popular sites (Amazon, Zillow, LinkedIn, etc.), Thunderbit has pre-built templates. One click, and you have your data. Free Data Export: Export to Excel, Google Sheets, Airtable, or Notion—no extra charge. Let's put it this way: if you're a business user who just wants the data, Thunderbit is like having a personal data butler. If you're a developer who loves tinkering, Python is still your playground—but even then, sometimes you just want to get the job done. Try Thunderbit AI Web Scraper for Free Best Practices for Ethical and Legal Python Web Scraping Web scraping is powerful, but it comes with responsibility. Here's how to stay on the right side of the law (and karma): Check robots.txt: Respect the site's wishes on what can be scraped. Read the Terms of Service: Some sites explicitly forbid scraping. Violating ToS can get you blocked or even sued. Rate Limit: Don't hammer servers—add delays between requests. Avoid Personal Data: Be careful with scraping emails, phone numbers, or anything that could be considered personal under GDPR or CCPA. Don't Circumvent Anti-Bot Measures: If a site uses CAPTCHAs or aggressive blocking, think twice. Attribute Sources: If you publish analysis, give credit to where the data came from. For more on the legal landscape, check out this and . Resources to Learn More Python Web Scraping (Courses, Docs, Communities) Want to go deeper? Here's my curated list of the best resources: Official Docs: Books: "Web Scraping with Python" by Ryan Mitchell "Automate the Boring Stuff with Python" by Al Sweigart Online Guides: Video Tutorials: Corey Schafer's YouTube channel Communities: And of course, if you want to see how no-code scraping works, check out the or the . Conclusion & Key Takeaways: Choosing the Right Web Scraping Solution in 2025 Python web scraping is incredibly powerful and flexible. If you love code, want full control, and don't mind a little maintenance, it's a great choice. There's a Python library for every scraping need—static pages, dynamic content, forms, APIs, PDFs, you name it. But for most business users, maintaining dozens of scripts is a pain. If your goal is to get data fast, at scale, and without a computer science degree, is the way to go. Thunderbit's AI-powered, no-code interface lets you scrape any website in a couple of clicks, handle subpages and pagination, and export data wherever you need it—no Python required. Ethics and legality matter: Always check site policies, respect privacy, and scrape responsibly. So, whether you're a Python pro or just want the data without the drama, the tools are better than ever in 2025. My advice? Try both approaches, see what fits your workflow, and don't be afraid to let the robots do the boring stuff—just make sure they're polite about it. And if you're tired of chasing broken scripts, give a spin. Your future self (and your coffee supply) will thank you. Want more? Check out or for hands-on guides and the latest scraping strategies. Want to give your brand videos a cinematic edge? Join our visual experts and special guests for an info-packed hour of insights to elevate your next video project. Tune in on June 24 at 11am ET.Register NowHow can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The CollectionWant to give your brand videos a cinematic edge? Join our visual experts and special guests for an info-packed hour of insights to elevate your next video project. Tune in on June 24 at 11am ET.Register NowHow can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The CollectionWant to give your brand videos a cinematic edge? Join our visual experts and special guests for an info-packed hour of insights to elevate your next video project. Tune in on June 24 at 11am ET.Register NowHow can financial brands set themselves apart through visual storytelling? Our experts explain how.Learn MoreThe Motorsport Images Collections captures events from 1895 to today's most recent coverage.Discover The Collection Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: Web Scraping With Beautiful Soup and Python Beautiful Soup is a Python library designed for parsing HTML and XML documents. It creates parse trees that make it straightforward to extract data from HTML documents you've scraped from the internet. Beautiful Soup is a useful tool in your web scraping toolkit, allowing you to conveniently extract specific information from HTML, even from complex static websites. In this tutorial, you'll learn how to build a web scraper using Beautiful Soup along with the Requests library to scrape and parse job listings from a static website. Static websites provide consistent HTML content, while dynamic sites may require handling JavaScript. For dynamic websites, you'll need to incorporate additional tools that can execute JavaScript, such as Scrapy or Selenium. By the end of this tutorial, you'll understand that: You can use Beautiful Soup for parsing HTML and XML documents to extract data from web pages. Beautiful Soup is named after a song in Alice's Adventures in Wonderland by Lewis Carroll, based on its ability to tackle poorly structured HTML known as tag soup. You'll often use Beautiful Soup in your web scraping pipeline when scraping static content, while you'll need additional tools such as Selenium to handle dynamic, JavaScript-rendered pages. Using Beautiful Soup is legal because you only use it for parsing documents. Web scraping in general is also legal if you respect a website's terms of service and copyright laws. Working through this project will give you the knowledge and tools that you need to scrape any static website out there on the World Wide Web. If you like learning with hands-on examples and have a basic understanding of Python and HTML, then this tutorial is for you! You can download the project source code by clicking on the link below: Take the Quiz: Test your knowledge with our interactive "Beautiful Soup: Build a Web Scraper With Python" quiz. You'll receive a score upon completion to help you track your learning progress: Interactive Quiz Beautiful Soup: Build a Web Scraper With Python In this quiz, you'll test your understanding of web scraping using Python. By working through this quiz, you'll revisit how to inspect the HTML structure of a target site, decipher data encoded in URLs, and use Requests and Beautiful Soup for scraping and parsing data. Web scraping is the process of gathering information from the internet. Even copying and pasting the lyrics of your favorite song can be considered a form of web scraping! However, the term "web scraping" usually refers to a process that involves automation. While some websites don't like it when automatic scrapers gather their data, which can lead to legal issues, others don't mind it. If you're scraping a page respectfully for educational purposes, then you're unlikely to have any problems. Still, it's a good idea to do some research on your own to make sure you're not violating any Terms of Service before you start a large-scale web scraping project. Say that you like to surf—both in the ocean and online—and you're looking for employment. It's clear that you're not interested in just any job. With a surfer's mindset, you're waiting for the perfect opportunity to roll your way! You know about a job site that offers precisely the kinds of jobs you want. Unfortunately, a new position only pops up once in a blue moon, and the site doesn't provide an email notification service. You consider checking up on it every day, but that doesn't sound like the most fun and productive way to spend your time. You'd rather be outside surfing real-life waves! Thankfully, Python offers a way to apply your surfer's mindset. Instead of having to check the job site every day, you can use Python to help automate the repetitive parts of your job search. With automated web scraping, you can write the code once, and it'll get the information that you need many times and from many pages. Note: In contrast, when you try to get information manually, you might spend a lot of time clicking, scrolling, and searching, especially if you need large amounts of data from websites that are regularly updated with new content. Manual web scraping can take a lot of time and be highly repetitive and error-prone. There's so much information on the internet, with new information constantly being added. You'll probably be interested in some of that data, and much of it is out there for the taking. Whether you're actually on the job hunt or just want to automatically download all the lyrics of your favorite artist, automated web scraping can help you accomplish your goals. The internet has grown organically out of many sources. It combines many different technologies, styles, and personalities, and it continues to grow every day. In other words, the internet is a hot mess! Because of this, you'll run into some challenges when scraping the web: Variety: Every website is different. While you'll encounter general structures that repeat themselves, each website is unique and will need personal treatment if you want to extract the relevant information. Durability: Websites constantly change. Say you've built a shiny new web scraper that automatically cherry-picks what you want from your resource of interest. The first time you run your script, it works flawlessly. But when you run the same script a while later, you run into a discouraging and lengthy stack of tracebacks! Unstable scripts are a realistic scenario because many websites are in active development. If a site's structure changes, then your scraper might not be able to navigate the sitemap correctly or find the relevant information. The good news is that changes to websites are often small and incremental, so you'll likely be able to update your scraper with minimal adjustments. Still, keep in mind that the internet is dynamic and keeps on changing. Therefore, the scrapers you build will probably require maintenance. You can set up continuous integration to run scraping tests periodically to ensure that your main script doesn't break without your knowledge. Some website providers offer application programming interfaces (APIs) that allow you to access their data in a predefined manner. With APIs, you can avoid parsing HTML. Instead, you can access the data directly using formats like JSON and XML. HTML is primarily a way to visually present content to users. When you use an API, the data collection process is generally more stable than it is through web scraping. That's because developers create APIs to be consumed by programs rather than by human eyes. The front-end presentation of a site might change often, but a change in the website's design doesn't affect its API structure. The structure of an API is usually more permanent, which means it's a more reliable source of the site's data. However, APIs can change as well. The challenges of both variety and durability apply to APIs just as they do to websites. Additionally, it's much harder to inspect the structure of an API by yourself if the provided documentation lacks quality. The approach and tools you need to gather information using APIs is outside the scope of this tutorial. To learn more about it, check out API Integration in Python. In this tutorial, you'll build a web scraper that fetches Python software developer job listings from a fake Python job site. It's an example site with fake job postings that you can freely scrape to train your skills. Your web scraper will parse the HTML on the site to pick out the relevant information and filter that content for specific words. You can scrape any site on the internet that you can look at, but the difficulty of doing so depends on the site. This tutorial offers you an introduction to web scraping to help you understand the overall process. Then, you can apply this same process for every website that you want to scrape. Note: Real-life job boards may quickly change in structure and availability. To offer you a smooth learning experience, this tutorial focuses on a self-hosted static site that's guaranteed to stay the same. This gives you a reliable playground to practice the skills that you need for web scraping. Throughout the tutorial, you'll also encounter a few exercise blocks. You can click to expand them and challenge yourself by completing the tasks described within. Before you write any Python code, you need to get to know the website that you want to scrape. Getting to know the website should be your first step for any web scraping project that you want to tackle. You'll need to understand the site structure to extract the information relevant for you. Start by opening the site that you want to scrape with your favorite browser. Click through the site and interact with it just like any typical job searcher would. For example, you can scroll through the main page of the website: On that page, you can see many job postings in a card format. Each of them has two buttons. If you click on Learn, then you'll visit Real Python's home page. If you click on Apply, then you'll see a new page that contains more detailed descriptions of the job on that card. You might also notice that the URL in your browser's address bar changes when you navigate to one of those pages. You can encode a lot of information in a URL. Becoming familiar with how URLs work and what they're made of will help you on your web scraping journey. For example, you might find yourself on a details page that has the following URL: You can deconstruct the above URL into two main parts: The base URL points to the main location of the web resource. In the example above, the base URL is . The path to a specific resource location points to a unique job description. In the example above, the path is fake-jobs/jobs/senior-python-developer-0.html. Any job posted on this website will share the same base URL. However, the location of the unique resources will be different depending on the job posting that you view. Usually, similar resources on a website will share a similar location, such as the folder structure fake-jobs/jobs/. However, the final part of the path points to a specific resource and will be different for each job posting. In this case, it's a static HTML file named senior-python-developer-0.html. URLs can hold more information than just the location of a file. Some websites use query parameters to encode values that you submit when performing a search. You can think of them as query strings that you send to the database to retrieve specific records. You'll find query parameters at the end of a URL. For example, if you go to Indeed and search for "software developer" in "Australia" through the site's search bar, you'll see that the URL changes to include these values as query parameters: The query parameters in this URL are ?q=software+developer&l=Australia. Query parameters consist of three parts: Start: You can identify the beginning of the query parameters by looking for the question mark (?). Information: You'll find the pieces of information that constitute one query parameter encoded in key-value pairs, where related keys and values are joined together by an equal sign (key=value). Separator: You'll see an ampersand symbol (&) separating multiple query parameters if there are more than one. Equipped with this information, you can separate the URL's query parameters into two key-value pairs: q=software+developer selects the type of job. l=Australia selects the location of the job. Try to change the search parameters and observe how that affects your URL. Go ahead and enter new values in the search bar of the Indeed job board: Change these values to observe the changes in the URL. Next, try to change the values directly in your URL. See what happens when you paste the following URL into your browser's address bar: If you change and submit the values in the website's search box, then it'll be directly reflected in the URL's query parameters and vice versa. If you change either of them, then you'll see different results on the website. As you can see, exploring the URLs of a site can give you insight into how to retrieve data from the website's server. Head back to Fake Python jobs and continue to explore it. This site is a static website containing hardcoded information. It doesn't operate on top of a database, which is why you won't have to work with query parameters in this scraping tutorial. Next, you'll want to learn more about how the data is structured for display. You'll need to understand the page structure to pick what you want from the HTML response that you'll collect in one of the upcoming steps. Developer tools can help you understand the structure of a website. All modern browsers come with developer tools installed. In this section, you'll learn how to work with the developer tools in Chrome. The process will be very similar on other modern browsers. In Chrome on macOS, you can open up the developer tools through the menu by selecting View → Developer → Developer Tools. On Windows and Linux, you can access them by clicking the top-right menu button (⋮) and selecting More Tools → Developer Tools. You can also access your developer tools by right-clicking on the page and selecting the Inspect option or using a keyboard shortcut: Mac: Cmd+Alt+I Windows/Linux: Ctrl+Shift+I Developer tools allow you to interactively explore the site's document object model (DOM) to better understand your source. To dig into your page's DOM, select the Elements tab in developer tools. You'll see a structure with clickable HTML elements. You can expand, collapse, and even edit elements right in your browser: The HTML on the right represents the structure of the page you can see on the left. You can think of the text displayed in your browser as the HTML structure of the page. If you're interested, then you can read more about the difference between the DOM and HTML. When you right-click elements on the page, you can select Inspect to zoom to their location in the DOM. You can also hover over the HTML text on your right and see the corresponding elements light up on the page. Click to expand the exercise block for a specific task to practice using your developer tools: Find a single job posting. What HTML element is it wrapped in, and what other HTML elements does it contain? Play around and explore! The more you get to know the page you're working with, the easier it'll be to scrape. But don't get too overwhelmed with all that HTML text. You'll use the power of programming to step through this maze and cherry-pick the information that's relevant to you. Now that you have an idea of what you're working with, it's time to start using Python. First, you'll want to get the site's HTML code into your Python script so that you can interact with it. For this task, you'll use Python's Requests library. Before you install any external package, you'll need to create a virtual environment for your project. Activate your new virtual environment, then type the following command in your terminal to install the Requests library: Then open up a new file in your favorite text editor and call it scraper.py. You only need a few lines of code to retrieve the HTML: Copied! When you run this code, it issues an HTTP GET request to the given URL. It retrieves the HTML data that the server sends back and stores that data in a Python object you called page. If you print the .text attribute of page, then you'll notice that it looks just like the HTML you inspected earlier with your browser's developer tools. You've successfully fetched the static site content from the internet! You now have access to the site's HTML from within your Python script. The website that you're scraping in this tutorial serves static HTML content. In this scenario, the server that hosts the site sends back HTML documents that already contain all the data a user gets to see. When you inspected the page with developer tools earlier on, you discovered that a single job posting consists of the following long and messy-looking HTML: It can be challenging to wrap your head around a long block of HTML code. To make it easier to read, you can use an HTML formatter to clean up the HTML automatically. Good readability can help you better understand the structure of any block of code. While improved HTML formatting may or may not help, it's always worth a try. Note: Keep in mind that every website looks different. That's why it's necessary to inspect and understand the structure of the site you're working with before moving forward. The HTML you'll encounter will sometimes be confusing. Luckily, the HTML of this job board has descriptive class names on the elements that you're interested in: class="title is-5" contains the title of the job posting. class="subtitle is-6 company" contains the name of the company that offers the position. class="location" contains the location where you'd be working. If you ever get lost in a large pile of HTML, remember that you can always go back to your browser and use the developer tools to further explore the HTML structure interactively. By now, you've successfully harnessed the power and user-friendly design of Python's Requests library. With only a few lines of code, you managed to scrape static HTML content from the web and make it available for further processing. While this was a breeze, you may encounter more challenging situations when working on your own web scraping projects. Before you learn how to select the relevant information from the HTML that you just scraped, you'll take a quick look at two more challenging situations. Some pages contain information that's hidden behind a login. This means you'll need an account to be able to scrape anything from the page. Just like you need to log in on your browser when you want to access content on such a page, you'll also need to log in from your Python script. The Requests library comes with the built-in capacity to handle authentication. With these techniques, you can log in to websites when making the HTTP request from your Python script and then scrape information that's hidden behind a login. You won't need to log in to access the job board information, so this tutorial won't cover authentication. Many modern websites don't send back static HTML content like this practice site does. If you're dealing with a dynamic website, then you could receive JavaScript code as a response. This code will look completely different from what you see when you inspect the same page with your browser's developer tools. Note: In this tutorial, the term dynamic website refers to a website that doesn't return the same HTML that you see when viewing the page in your browser. Dynamic websites are designed to provide their functionality in collaboration with the clients' browsers. Instead of sending HTML pages, these apps send JavaScript code that instructs your browser to create the desired HTML. Web apps deliver dynamic content this way to offload work from the server to the clients' machines, as well as to avoid page reloads and improve the overall user experience. Your browser will diligently execute the JavaScript code it receives from a server and create the DOM and HTML for you locally. However, if you request a dynamic website in your Python script, then you won't get the HTML page content. When you use Requests, you receive only what the server sends back. In the case of a dynamic website, you'll end up with JavaScript code without the relevant data. The only way to go from that code to the content that you're interested in is to execute the code, just like your browser does. The Requests library can't do that for you, but there are other solutions that can: Requests-HTML is a project created by the author of the Requests library that allows you to render JavaScript using syntax that's similar to the syntax in Requests. It also includes capabilities for parsing the data by using Beautiful Soup under the hood. Selenium is another popular choice for scraping dynamic content. Selenium automates a full browser and can execute JavaScript, allowing you to interact with and retrieve the fully rendered HTML response for your script. You won't go deeper into scraping dynamically-generated content in this tutorial. If you need to scrape a dynamic website, then you can look into one of the options mentioned above. You've successfully scraped some HTML from the internet, but when you look at it, it looks like a mess. There are tons of HTML elements here and there, thousands of attributes scattered around—and maybe there's some JavaScript mixed in as well? It's time to parse this lengthy code response with the help of Python to make it more accessible so you can pick out the data that you want. Beautiful Soup is a Python library for parsing structured data. It allows you to interact with HTML in a similar way to how you interact with a web page using developer tools. The library exposes intuitive methods that you can use to explore the HTML you received. Note: The name Beautiful Soup originates from the Lewis Carroll song Beautiful Soup in Alice's Adventures in Wonderland, where a character sings about beautiful soup. This name reflects the library's ability to parse poorly formed HTML that's also known as tag soup. To get started, use your terminal to install Beautiful Soup into your virtual environment: Then, import the library in your Python script and create a BeautifulSoup object: Copied! When you add the two highlighted lines of code, then you create a BeautifulSoup object that takes page.content as input, which is the HTML content that you scraped earlier. Note: You'll want to pass .content instead of .text to avoid problems with character encoding. The .content attribute holds raw bytes, which Python's built-in HTML parser can decode better than the text representation you printed earlier using the .text attribute. The second argument that you pass to the class constructor, "html.parser", makes sure that you use an appropriate parser for HTML content. At this point, you're set up with a BeautifulSoup object that you named soup. You can now run your script using Python's interactive mode: When you use the command-option -i to run a script, then Python executes the code and drops you into a REPL environment. This can be a good way to continue exploring the scraped HTML through the user-friendly lens of Beautiful Soup. In an HTML web page, every element can have an id attribute assigned. As the name already suggests, that id attribute makes the element uniquely identifiable on the page. You can begin to parse your page by selecting a specific element by its ID. Switch back to developer tools and identify the HTML object that contains all the job postings. Explore by hovering over parts of the page and using right-click to Inspect. Note: It helps to periodically switch back to your browser and explore the page interactively using developer tools. You'll get a better idea of where and how to find the exact elements that you're looking for. In this case, the element that you're looking for is a with an id attribute that has the value "ResultsContainer". It has some other attributes as well, but below is the gist of what you're looking for: Beautiful Soup allows you to find that specific HTML element by its ID: For easier viewing, you can prettify any BeautifulSoup object when you print it out. If you call .prettify() on the results variable that you assigned above, then you'll see all the HTML contained within the neatly structured: When you find an element by its ID, you can pick out one specific element from among the rest of the HTML, no matter how large the source code of the website is. Now you can focus on working with only this part of the page's HTML. It looks like your soup just got a little thinner! Nevertheless, it's still quite dense. You've seen that every job posting is wrapped in a element with the class card-content. Now you can work with your new object called results and select only the job postings in it. These are, after all, the parts of the HTML that you're interested in! You can pick out all job cards in a single line of code: Here, you call .find_all() on results, which is a BeautifulSoup object. It returns an iterable containing all the HTML for all the job listings displayed on that page. Take a look at all of them: That's pretty neat already, but there's still a lot of HTML! You saw earlier that your page has descriptive class names on some elements. You can pick out those child elements from each job posting with .find(): Each job_card is another BeautifulSoup() object. Therefore, you can use the same methods on it as you did on its parent element, results. With this code snippet, you're getting closer and closer to the data that you're actually interested in. Still, there's a lot going on with all those HTML tags and attributes floating around: Next, you'll learn how to narrow down this output to access only the text content that you're interested in. You only want to see the title, company, and location of each job posting. And behold! Beautiful Soup has got you covered. You can add .text to a BeautifulSoup object to return only the text content of the HTML elements that the object contains: Run the above code snippet, and you'll see the text of each element displayed. However, you'll also get some extra whitespace. But no worries, because you're working with Python strings so you can .strip() the superfluous whitespace. You can also apply any other familiar Python string methods to further clean up your text: The results finally look much better! You've now got a readable list of jobs, associated company names, and each job's location. However, you're specifically looking for a position as a software developer, and these results contain job postings in many other fields as well. Not all of the job listings are developer jobs. Instead of printing out all the jobs listed on the website, you'll first filter them using keywords. You know that job titles in the page are kept within

elements. To filter for only specific jobs, you can use the string argument: This code finds all

elements where the contained string matches "Python" exactly. Note that you're directly calling the method on your first results variable. If you go ahead and print() the output of the above code snippet to your console, then you might be disappointed because it'll be empty: There was a Python job in the search results, so why isn't it showing up? When you use string as you did above, your program looks for that string exactly. Any variations in the spelling, capitalization, or whitespace will prevent the element from matching. In the next section, you'll find a way to make your search string more general. In addition to strings, you can sometimes pass functions as arguments to Beautiful Soup methods. You can change the previous line of code to use a function instead: Now you're passing an anonymous function to the string argument. The lambda function looks at the text of each

element, converts it to lowercase, and checks whether the substring "python" is found anywhere. You can check whether you managed to identify all the Python jobs with this approach: Your program has found ten matching job posts that include the word "python" in their job title! Finding elements based on their text content is a powerful way to filter your HTML response for specific information. Beautiful Soup allows you to use exact strings or functions as arguments for filtering text in BeautifulSoup objects. However, when you try to print the information of the filtered Python jobs like you've done before, you run into an error: This traceback message is a common error that you'll run into a lot when you're scraping information from the internet. Inspect the HTML of an element in your python_jobs list. What does it look like? Where do you think the error is coming from? When you look at a single element in python_jobs, you'll see that it consists of only the

element that contains the job title: When you revisit the code you used to select the items, you'll notice that's what you targeted. You filtered for only the

title elements of the job postings that contain the word "python". As you can see, these elements don't include the rest of the information about the job. The error message you received earlier was related to this: You tried to find the job title, the company name, and the job's location in each element in python_jobs, but each element contains only the job title text. Your diligent parsing library still looks for the other ones, too, and returns None because it can't find them. Then, print() fails with the shown error message when you try to extract the .text attribute from one of these None objects. The text you're looking for is nested in sibling elements of the

elements that your filter returns. Beautiful Soup can help you select sibling, child, and parent elements of each BeautifulSoup object. One way to get access to all the information for a job is to step up in the hierarchy of the DOM starting from the

elements that you identified. Take another look at the HTML of a single job posting, for example, using your developer tools. Then, find the

element that contains the job title and its closest parent element that contains the information you're interested in: The element with the card-content class contains all the information you want. It's a third-level parent of the

title element that you found using your filter. With this information in mind, you can now use the elements in python_jobs and fetch their great-grandparent elements to get access to all the information you want: You added a list comprehension that operates on each of the

title elements in python_jobs that you got by filtering with the lambda expression. You're selecting the parent element of the parent element of the parent element of each

title element. That’s three generations up! When you were looking at the HTML of a single job posting, you identified that this specific parent element with the class name card-content contains all the information you need. Now you can adapt the code in your for loop to iterate over the parent elements instead: When you run your script another time, you’ll see that your code once again has access to all the relevant information. That’s because you’re now looping over the elements instead of just the

title elements. Using the .parent attribute that each BeautifulSoup object comes with gives you an intuitive way to step through your DOM structure and address the elements you need. You can also access child elements and sibling elements in a similar manner. Read up on navigating the tree for more information. At this point, you’ve already written code that scrapes the site and filters its HTML for relevant job postings. Well done! However, what’s still missing is fetching the link to apply for a job. While inspecting the page, you found two links at the bottom of each card. If you use .text on the link elements in the same way you did for the other elements, then you won’t get the URLs that you’re interested in: If you execute the code shown above, then you’ll get the link text for Learn and Apply instead of the associated URLs. That’s because the .text attribute leaves only the visible content of an HTML element. It strips away all HTML tags, including the HTML attributes containing the URL, and leaves you with just the link text. To get the URL instead, you need to extract the value of one of the HTML attributes instead of discarding it. The URL of a link element is associated with the href HTML attribute. The specific URL that you’re looking for is the value of the href attribute of the second tag at the bottom of the HTML for a single job posting: Start by fetching all the elements in a job card. Then, extract the value of their href attributes using square-bracket notation: In this code snippet, you first fetch all the links from each of the filtered job postings. Then, you extract the href attribute, which contains the URL, using ["href"] and print it to your console. Each job card has two links associated with it. However, you’re only looking for the second link, so you’ll apply a small edit to the code: In the updated code snippet, you use indexing to pick the second link element from the results of .find_all() using its index ([1]). Then, you directly extract the URL using the square-bracket notation with the "href" key, thereby fetching the value of the href attribute. You can use the same square-bracket notation to extract other HTML attributes as well. You’re now happy with the results and are ready to put it all together into your scraper.py script. When you assemble the useful lines of code that you wrote during your exploration, you’ll end up with a Python web scraping script that extracts the job title, company, location, and application link from the scraped website: Copied! You could continue to work on your script and refactor it, but at this point, it does the job you wanted and presents you with the information you need when you want to apply for a Python developer job: All you need to do now to check for new Python jobs on the job board is run your Python script. This leaves you with plenty of time to get out there and catch some waves! If you’ve written the code alongside this tutorial, then you can run your script as is to see the fake job information pop up in your terminal. Your next step is to tackle a real-life job board! To keep practicing your new skills, you can revisit the web scraping process described in this tutorial by using any or all of the following sites: Python.org Job Board PythonJobs Remote The linked websites return their search results as static HTML responses, similar to the Fake Python job board. Therefore, you can scrape them using only Requests and Beautiful Soup. Start going through this tutorial again from the beginning using one of these other sites. You’ll see that each website’s structure is different and that you’ll need to rebuild the code in a slightly different way to fetch the data you want. Tackling this challenge is a great way to practice the concepts that you just learned. While it might make you sweat every so often, your coding skills will be stronger in the end! During your second attempt, you can also explore additional features of Beautiful Soup. Use the documentation as your guidebook and inspiration. Extra practice will help you become more proficient at web scraping with Python, Requests, and Beautiful Soup. To wrap up your journey, you could then give your code a final makeover and create a command-line interface (CLI) app that scrapes one of the job boards and filters the results by a keyword that you can input on each execution. Your CLI tool could allow you to search for specific types of jobs, or jobs in particular locations. If you’re interested in learning how to adapt your script as a command-line interface, then check out the Build Command-Line Interfaces With Python’s argparse tutorial. The Requests library provides a user-friendly way to scrape static HTML from the internet with Python. You can then parse the HTML with another package called Beautiful Soup. You’ll find that Beautiful Soup will cater to most of your parsing needs, including navigation and advanced searching. Both packages will be trusted and helpful companions on your web scraping adventures. In this tutorial, you’ve learned how to: Step through a web scraping pipeline from start to finish Inspect the HTML structure of your target site with your browser’s developer tools Decipher the data encoded in URLs Download the page’s HTML content using Python’s Requests library Parse the downloaded HTML with Beautiful Soup to extract relevant information Build a script that fetches job offers from the web and displays relevant information in your console With this broad pipeline in mind and two powerful libraries in your toolkit, you can go out and see what other websites you can scrape. Have fun, and always remember to be respectful and use your programming skills responsibly. Happy scraping! Now that you have some experience with Beautiful Soup and web scraping in Python, you can use the questions and answers below to check your understanding and recap what you’ve learned. These FAQs are related to the most important concepts you’ve covered in this tutorial. Click the Show/Hide toggle beside each question to reveal the answer: Web scraping is the automated process of extracting data from websites. It’s useful because it allows you to gather large amounts of data efficiently and systematically, which can be beneficial for research, data analysis, or keeping track of updates on specific sites, such as job postings. You can use your browser’s developer tools to inspect the HTML structure of a website. To do this, right-click on any element of the page and select Inspect. This will allow you to view the underlying HTML code, helping you understand how the data you want is structured. The Requests library is used to send HTTP requests to a website and retrieve the HTML content of the web page. You’ll need to get the raw HTML before you can parse and process it with Beautiful Soup. Beautiful Soup is a Python library used for parsing HTML and XML documents. It provides Pythonic idioms for iterating, searching, and modifying the parse tree, making it easier to extract the necessary data from the HTML content you scraped from the internet. Some challenges include handling dynamic content generated by JavaScript, accessing login-protected pages, dealing with changes in website structure that could break your scraper, and navigating legal issues related to the terms of service of the websites you’re scraping. It’s important to approach this work responsibly and ethically. Take the Quiz: Test your knowledge with our interactive “Beautiful Soup: Build a Web Scraper With Python” quiz. You’ll receive a score upon completion to help you track your learning progress: Interactive Quiz Beautiful Soup: Build a Web Scraper With Python In this quiz, you'll test your understanding of web scraping using Python. By working through this quiz, you'll revisit how to inspect the HTML structure of a target site, decipher data encoded in URLs, and use Requests and Beautiful Soup for scraping and parsing data. Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: Web Scraping With Beautiful Soup and Python