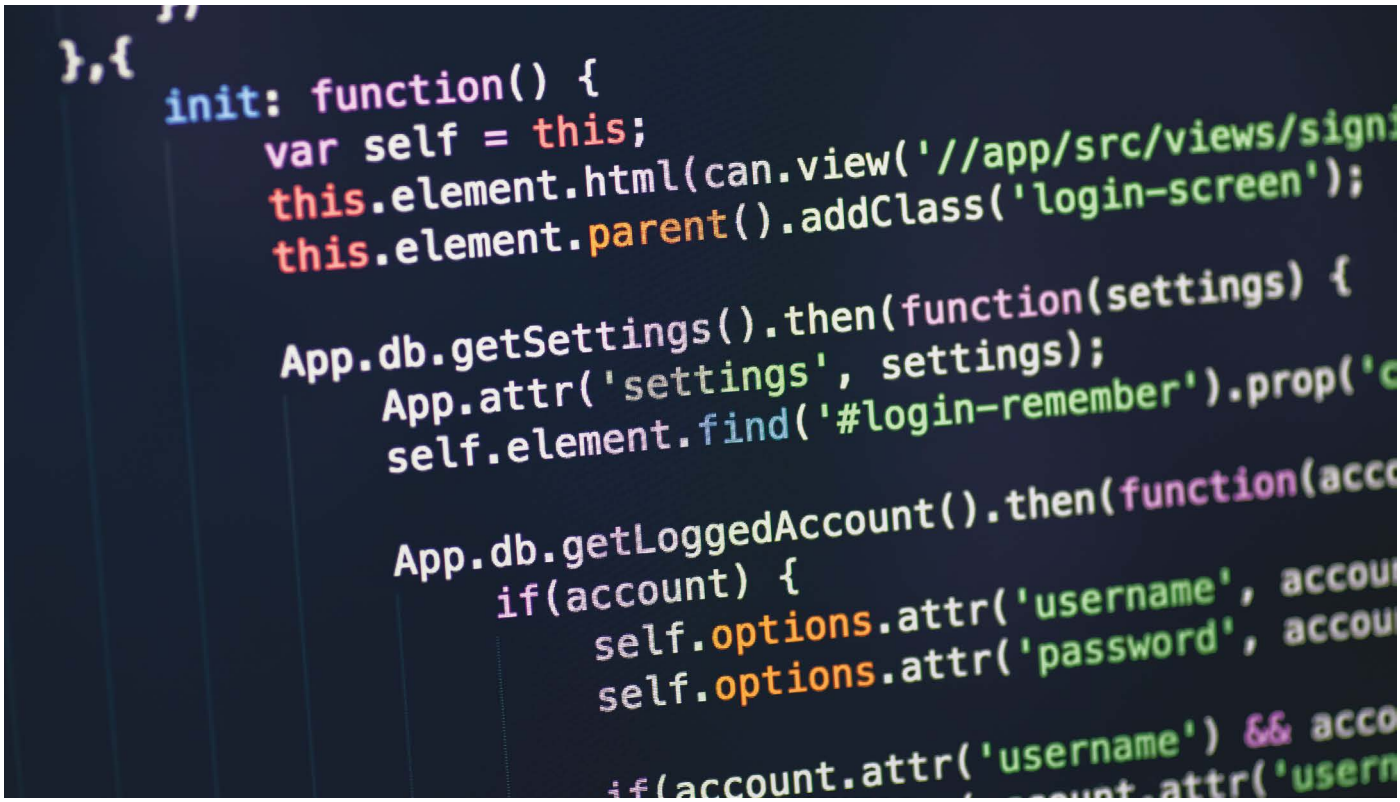
Work

Your story

Send your careers story to: naturecareerseditor@nature.com



SHUTTERSTOCK

TOOLS THAT EASE DATA COLLECTION FROM THE WEB

Custom web scrapers are driving research — and collaborations.

By Nicholas J. DeVito, Georgia C. Richards and Peter Inglesby

In research, time and resources are precious. Automating common tasks, such as data collection, can make a project efficient and repeatable, leading in turn to increased productivity and output. You will end up with a shareable and reproducible method for data collection that can be verified, used and expanded on by others — in other words, a computationally reproducible data-collection workflow.

In a current project, we are analysing coroners' reports to help to prevent future deaths. It has required downloading more than 3,000 PDFs to search for opioid-related deaths, a huge data-collection task. In discussion with the larger team, we decided that this task was a good candidate for automation. With a few days of work, we were able to write a computer program that could quickly, efficiently and reproducibly collect all the PDFs and create a spreadsheet that documented each case.

Such a tool is called a 'web scraper', and our group employs them regularly. We use them to collect information from clinical-trial registries, and to enrich our OpenPrescribing.net data set, which tracks primary-care prescribing in England — tasks that would range from annoying to impossible without the help of some relatively simple code.

In the case of our coroner-reports project, we could manually screen and save about 25 case reports every hour. Now, our program can save more than 1,000 cases per hour while we work on other things, a 40-fold time saving. It also opens up opportunities for collaboration, because we can share the resulting database. And we can keep that database up to date by re-running our program as new PDFs are posted.

How does scraping work?

Web scrapers are computer programs that extract information from — that is, 'scrape' — web sites. The structure and content of a web page are encoded in Hypertext Markup Language (HTML), which you can see using your browser's 'view source' or 'inspect element' function. A scraper understands HTML, and is able to parse and extract information from it. For example, you can program your scraper to extract specific fields of information from an online table or download documents linked on the page.

A common scraping task involves iterating over every possible URL from www.example.com/data/1 to www.example.com/data/100

(sometimes called 'crawling') and storing what you need from each page without the risk of human error during extraction. Once your program is written, you can recapture these data whenever you need to, assuming the structure of the website stays mostly the same.

How do I get started?

Not all scraping tasks require programming. When you visit a web page in your browser, off-the-shelf browser extensions such as webscraper.io let you click on the elements of the page that contain the data that you're interested in. They can then automatically parse the relevant parts of the HTML and export the data as a spreadsheet.

The alternative is to build your own scraper — a more difficult process, but one that offers greater control. We use Python, but any modern programming language should work. (For specific packages, Requests and BeautifulSoup work well together in Python; for R, try rvest.) It's worth checking whether anyone else has already written a scraper for your data source. If not, there's no shortage of resources and free tutorials to help you to get started no matter your preferred language.

As with most programming projects, there will be some trial and error, and different websites might use different data structures or variations in how their HTML is implemented that will require tweaks to your approach. Yet this problem-solving aspect of development can be quite rewarding. As you get more comfortable with the process, overcoming these barriers will start to seem like second nature.

But be advised: depending on the number of pages, your Internet connection and the website's server, a scraping job could still take days. If you have access and know-how, running your code on a private server can help. On a personal computer, make sure to prevent your computer from sleeping, which will disrupt the Internet connection. Also, think carefully about how your scraper can fail. Ideally, you should have a way to log failures so that you know what worked, what didn't and where to investigate further.

Things to consider

Can you get the data an easier way? Scraping all 300,000+ records off of ClinicalTrials.gov every day would be a massive job for our FDAAA TrialsTracker project. Luckily, ClinicalTrials.gov makes their full dataset available for download; our software simply grabs that file once per day. We weren't so lucky with the data for our EU TrialsTracker, so we scrape the EU registry monthly.

If there's no bulk download available, check to see whether the website has an application programming interface (API). An API lets software interact with a website's data directly, rather than requesting the HTML. This can be much less burdensome than scraping individual web pages, but there might be a fee associated with API access (see, for example, Google's Map API). In our work, the PubMed API is often useful. Alternatively, check whether the website operators can provide the data to you directly.

Can this website be scraped? Some websites don't make their data available directly in the HTML and might require some more advanced techniques (check resources such as StackOverflow for help with specific questions). Other websites include protections like captchas and anti-denial-of-service (DoS) measures that can make scraping difficult. A few websites simply don't want to be scraped and are built to discourage it. It's also common to allow scraping but only if you follow certain rules, usually codified in a robots.txt file.

Are you being a courteous scraper? Every time your program requests data from a website, the underlying information needs to be 'served' to you. You can only move so quickly in a browser, but a scraper could potentially send hundreds to thousands of requests per minute. Hammering a web server like that can slow, or entirely bring down, the website (essentially performing an unintentional DoS attack). This could get you temporarily, or even permanently, blocked from the website — and you should take care to minimize the chances of harm. For instance, you can pause your program for a few seconds between each request. (Check the site's robots.txt file to see whether it specifies a desired pause length.)

Are the data restricted? Be sure to check for licensing or copyright restrictions on the extracted data. You might be able to use what you scrape, but it's worth checking that you can also legally share it. Ideally, the website content licence will be readily available. Whether or not you can share the data, you should share your code using services such as GitHub — this is good open-science practice and ensures that others can discover, repeat and build on what you've done.

We strongly feel that more researchers should be developing code to help conduct their research, and then sharing it with the community. If manual data collection has been an issue for your project, a web scraper could be the perfect solution and a great beginner coding project. Scrapers are complex enough to teach important lessons about software development, but common and well-documented enough that beginners can feel confident experimenting. Writing some relatively simple code on your computer and having it interact with the outside world can feel like a research superpower. What are you waiting for?

Nicholas J. DeVito and **Georgia C. Richards** are doctoral candidates and researchers and **Peter Inglesby** is a software engineer, at the EBM DataLab at the University of Oxford, UK.