1 Introduction

Have you ever wanted a robot to perform the repetitive, time-consuming tasks you hate doing? Tasks that stop you from pursuing key questions in your field, such as where the money goes, or whose name keeps cropping up in a regulator's reports? Perhaps you'd like a robot to save you combining hundreds of spreadsheets into one easy-to-interrogate one? Or to comb through thousands of documents looking for any mention of a particular issue, person or company? Perhaps your robot would just grab a table or two from the web and put it in a spreadsheet to save you a few clicks in your day-to-day work, or do things more quickly than your competition so you get to a story first.

Scraping - getting a computer to capture information from online sources - is that robot. You tell the scraper what to do, and how to do it, then sit back and get on with the things that cannot be automated, like speaking to sources or reading up on how a particular set of data has been gathered, or reports prepared.

It's not exactly *Wall-E*, but it's rarely used compared to the other three ways to get hold of data (from a source, via access to information laws, or through advanced search techniques). And yet it is probably the most powerful way for journalists to get hold of information. Scraping is *faster than FOI*, provides *more granular results* than most advanced searches - and allows you to grab data that organisations would *rather you didn't have.*

Oh yes, I like that last one too.

It also allows you to collate and analyse information that no one may have collected before: notices, mentions, documents, relationships, decisions - in fact, anything that's been digitised. That information might be stored in all sorts of ways: tables buried in PDFs and webpages, information hidden behind search forms, or scattered across hundreds of pages or spreadsheets with no single link to them all. What's notable about scraping is that it is not just about grabbing data from online sources - it is also about putting it into some sort of structure. Because what we really want to do is ask it questions.

And if you're curious, and deadlines and topicality are important to you, then scraping is a wonderful skill to have.

A book about not reading books

I was moved to write this book when I noticed many journalists were trying to learn scraping and programming but struggling to get a foothold - or losing momentum once they did.

As an educator it struck me that many were losing motivation either because they weren't getting results quickly enough, or because there was too much to learn all at once.

As people who typically have a humanities-based educational background - and I include myself here - journalists approach learning in a particular way. If you want to learn a subject such as history or English, you read a book. But programming cannot be learned from books alone.

People learning programming read books, yes, but that's not all: they experiment and solve problems. In fact, the books are often there as a *resource* for when they are solving problems, rather than the focus of learning.

The best advice for anyone seeking to learn scraping or data journalism, then, is this: find a problem to solve first.

This book is designed to help you tackle the obstacles you will find in typical scraping problems. It is structured as a series of tasks, each of which makes up a chapter, and each of which produces tangible results.

Starting from very simple scraping techniques which are no more complicated than a spreadsheet formula, and taking in tools from Google Docs to Scraperwiki along the way, the book introduces you to different programming principles as and when they are needed. You'll be scraping within 5 minutes of reading this - but it's what you learn from that, and how you build on it, which is the really important thing.

Unlike general books about programming languages, everything you learn here will have a direct application for journalism, and each principle of programming will be related to their application in scraping for newsgathering.

And unlike standalone guides and blog posts that cover particular tools or techniques, this book aims to give you skills that you can apply in new situations and with new tools.

Because programming is not about simply knowing a language - it is about a way of looking at problems, diagnosing them, and solving them. If you were used to getting things right first time in school and college, get unused to it: half of the skill with scraping - and half the fun - is working out why things went wrong. And things always go wrong.

There is a saying I particularly like on this subject: *Pulvis et umbra sumus*:

> "*To know what to ask is already to know half.*"

This book aims to teach you not just the principles of programming but the practices; the questions to ask when tackling scraping problems; and the places to ask them.

I'm not a programmer

Although this book covers some principles of programming, I'm not a programmer: I'm a journalist and educator who uses programming to get hold of information.

That means I sometimes do or explain things in a way that some programmers might find ungainly.

So, two things you need to know: if you're a programmer and something in this book bothers you - let me know. I have done my best to check that my explanations make sense to programmers as well as journalists - but the book is not aimed at programmers.

That means that, although it is important to get the terminology correct, it is more important for a journalist that something works and makes sense. And so I have aimed to simplify things wherever possible rather than bog down things with details or debates which add nothing for the beginner.

For example, scraping itself is known by various different terms, which are often a source of conflict - is it “screen-scraping”? Or “data mining”? Whatever you call it, for the purposes of this book scraping is used generically to refer to the process of grabbing information from a file (a webpage or document) or a series of files.

Because the nature of those files can vary so widely, scraping itself varies enormously as well. The technical challenges of grabbing numbers buried in hundreds of PDFs that are only accessible through a search interface are very different from grabbing data from a series of tables all linked from the same page.

We’ll tackle a number of different challenges as we go through the book - starting with something very simple: scraping a table from a webpage.

As we do that we’ll be introduced to the two pieces of jargon I want you to understand first: functions, and parameters. You will be using both from the start, and understand how they form the basis of most scraping.

PS: This isn’t a book

Oh, before that? One more thing: this book is a work in progress. You can download new chapters as they are published, but more importantly, you can influence what gets written and how. If you find any mistakes, or things you want added or explained further, let me know through any of the following ways:

- On the Facebook page for this book at Facebook.com/ScrapingForJour
- On the support blog for this book at ScrapingForJournalists.posterous.com/[2]
- On Twitter @paulbradshaw[3]

Now, let's begin.

[1]http://www.facebook.com/ScrapingForJournalists
[2]http://scrapingforjournalists.posterous.com/
[3]http://twitter.com/paulbradshaw

2 Scraper #1: Start scraping in 5 minutes

List of prisons

From Wikipedia, the free encyclopedia

This page provides a **list of prisons** by country.

You can write a very basic scraper by using Google Drive, selecting **Create>Spreadsheet**, and adapting this formula - it doesn't matter where you type it:

```
=ImportHTML("ENTER THE URL HERE", "table", 1)
```

This formula will go to the **URL** you specify, look for a **table**, and pull the **first one** into your spreadsheet.

If you're using a Portuguese, Spanish or German version of Google Docs - or have any problems with the formula - use semi colons instead of commas. We're using commas here because this convention will continue when we get into programming in later chapters.

Let's imagine it's the day after a big horse race where two horses died, and you want some context. Or let's say there's a topical story relating to prisons and you want to get a global overview of the field: you could use this formula by typing it into the first cell of an empty Google Docs spreadsheet and replacing ENTER THE URL HERE with http://www.horsedeathwatch.com or http://en.wikipedia.org/wiki/List_of_prisons. Try it and see what happens. It should look like this:

```
=ImportHTML("http://en.wikipedia.org/wiki/List_-
of_prisons", "table", 1)
```

Don't copy and paste this - it's always better to type directly to avoid problems with hyphenation and curly quotation marks, etc.

After a moment, the spreadsheet should start to pull in data from the first table on that webpage.

So, you've written a scraper. It's a very basic one, but by understanding how it works and building on it you can start to make more and more ambitious scrapers with different languages and tools.

How it works: functions and parameters

```
=ImportHTML("http://en.wikipedia.org/wiki/List_of_-
prisons", "table", 1)
```

The scraping formula above has two core ingredients: a function, and parameters:

- **importHTML** is the **function**. Functions (as you might expect) *do* things. According to Google Docs' Help pages[1] this one "imports the data in a particular table or list from an HTML page"
- Everything within the parentheses (brackets) are the **parameters**. Parameters are the *ingredients* that the function needs in order to work. In this case, there are three: a URL, the word "table", and a number 1.

You can use different functions in scraping to tackle different problems, or achieve different results. Google Docs, for example, also has functions called importXML, importFeed and importData - some of which we'll cover later. And if you're writing scrapers with languages like Python, Ruby or PHP you can create your own functions that extract particular pieces of data from a page or PDF.

[1]http://support.google.com/docs/bin/answer.py?hl=en&answer=155182

What are the parameters? Strings and indexes

Back to the formula:

```
    =ImportHTML("http://en.wikipedia.org/wiki/List_-
of_prisons", "table", 1)
```

In addition to the function and parameters, it's important to explain some other things you should notice:

- Firstly, the = sign at the start. This tells Google Docs that this is a **formula**, rather than a simple number or text entry
- Secondly, notice that two of the three parameters use straight quotation marks: the URL, and "table". This is because they are **strings**: strings are basically words, phrases or any other collection (i.e. *string*) of characters. The computer treats these differently to other types of information, such as numbers, dates, or cell references - we'll come across these again later.
- The third parameter does not use quotation marks, because it is a number. In fact, in this case it's a number with a particular meaning: an **index** - the position of the table we're looking for (first, second, third, etc)

Knowing these things helps both in avoiding mistakes (for example, if you omit a quotation mark or use curly quotation marks it won't work) and in adapting a scraper…

For example, perhaps the table you got wasn't the one you wanted. Try replacing the number 1 in your formula with a number 2. This should now scrape the second table (in Google Docs an index starts from 1).

Knowing to search for information (often called '**documentation**') on a function is important too. The page on Google Docs Help[2], for example, explains that we can use "list" instead of "table" if you wanted to grab a list from the webpage.

So try that, and see what happens (make sure the webpage has a list).

```
=ImportHTML("http://en.wikipedia.org/wiki/List_-
of_prisons", "list", 1)
```

You can also try replacing either string with a cell reference. For example:

```
=ImportHTML(A2, "list", 1)
```

And then in cell A2 type or paste:

```
http://en.wikipedia.org/wiki/List_of_prisons
```

Notice that you don't need quotation marks around the URL if it's in another cell.

Using cell references like this makes it easier to change your formula: instead of having to edit the whole formula you only have to change the value of the cell that it's drawing from.

For examples of scrapers that do all of the above, see this example[3].

[2]http://support.google.com/docs/bin/answer.py?hl=en&answer=155182

[3]https://docs.google.com/spreadsheet/ccc?key=0ApTo6f5Yj1iJdDBSb0FPQm9jUjYzdjcyNWlUTjVYMFE

Tables and lists?

There's one final element in this scraper that deserves some further exploration: what it means by "table" or "list".

When we say "table" or "list" we are specifically asking it to look for a **HTML tag** in the code of the webpage. You can - and should - do this yourself…

Look at the raw HTML of your webpage by right-clicking on the webpage and selecting **View Page Source**, or using the shortcuts CTRL+U (Windows) and CMD+U (Mac) in Firefox, or a plugin like Firebug. You can also view it by selecting **Tools** > **Web Developer** > **Page Source** in Firefox or **View** > **Developer** > **View Source** in Chrome. *Note: for viewing source HTML, Firefox and Chrome are generally better set up.*

You'll now see the HTML. Use **Edit>Find** on your browser (or CTRL+F) to search for **<table**

When =importHTML looks for a table, this is what it looks for - and it will grab everything between <table> and </table> (which marks the end of the table)

With "list", =importHTML is looking for the tags **<ul>** (unordered list - normally displayed as bullet lists) or **<ol>** (ordered list - normally displayed as numbered lists). The end of each list is indicated by either </ul> or </ol>.

Both tables and lists will include other tags, such as <li> (list item), <tr> (table row) and <td> (table data) which add further structure - and that's what Google Docs uses to decide how to organise that data across rows and columns - but you don't need to worry about them.

How do you know what index number to use? Well, there are two ways: you can look at the raw HTML and count how many tables there are - and which one you need. Or you can just use trial and error, beginning with 1, and going up until it grabs the table you want. That's normally quicker.

Trial and error, by the way, is a common way of learning in scraping - it's quite typical not to get things right first time, and you shouldn't be disheartened if things go wrong at first.

Don't expect yourself to know everything there is to know about programming: half the fun is solving the inevitable problems that arise, and half the skill is in the techniques that you use to solve them (some of which I'll cover here), and learning along the way.

Scraping tip #1: Finding out about functions

We've already mentioned one of those problem-solving techniques, which is to look for the Help pages relating to the function you're using - what's often called the '**documentation**'.

When you come across a function (pretty much any word that comes after the = sign) it's always a good idea to Google it. Google Docs has extensive help pages - documentation - that explain what the function does, as well as discussion around particular questions.

Likewise, as you explore more powerful scrapers such as those hosted on Scraperwiki or Github, search for ‘documentation’ and the name of the function to find out more about how it works.

Recap

Before we move on, here’s a summary of what we’ve covered:

- **Functions** *do things…*
- they need ingredients to do this, supplied in **parameters**
- There are different kinds of parameters: **strings**, for example, are collections of characters, indicated by quotation marks
- and an **index** is a position indicated by a number, such as first (1), second (2) and so on.
- The strings “table” and “list” in this formula refer to particular **HTML tags** in the code underlying a page

Although this is described as a 'scraper' the results only exist as long as the page does. The advantage of this is that your spreadsheet will update every time the page does (you can set the spreadsheet to notify you by email whenever it updates by going to ***Tools>Notification rules*** *in the Google spreadsheet and selecting how often you want to be updated of changes).*

The disadvantage is that if the webpage disappears, so will your data. So it's a good idea to keep a static copy of that data in case the webpage is taken down or changed. You can do this by selecting all the cells and clicking on ***Edit>Copy*** *then going to a new spreadsheet and clicking on* ***Edit>Paste values only***

We'll come back to these concepts again and again, beginning with HTML. But before you do that - try this…

Tests

To reinforce what you've just learned - or to test you've learned it at all - here are some tasks to get you solving problems creatively:

- Let's say you need a list of towns in Hungary (this was an actual task I needed to undertake for a story). What formula would you write to scrape the first table on this page: http://en.wikipedia.org/wiki/List_-of_cities_and_towns_in_Hungary

- To make things easier for yourself, how can you change the formula so it uses cell references for each of the three parameters? (Make sure each cell has the relevant parameter in it)
- How can you change one of those cells so that the formula scrapes the second table?
- How can you change it so it scrapes a list instead?
- Look at the source code for the page you're scraping - try using the Find command (CTRL+F) to count the tables and work out which one you need to scrape the table of smaller cities - adapt your formula so it scrapes that
- Try to explain what a parameter is (tip: choose someone who isn't going to run away screaming)
- Try to explain what an index is
- Try to explain what a string is
- Look for the documentation on related functions like importData and importFeed - can you get those working?

Once you're happy that you've nailed these core concepts, it's time to move on to Scraper #2…