SCRAPING FOR JOURNALISTS

How to grab data from hundreds of sources, put it in a form you can interrogate — and still hit deadlines

PAUL BRADSHAW

Scraping for Journalists

How to grab information from hundreds of sources, put it in data you can interrogate - and still hit deadlines

Paul Bradshaw

This book is for sale at
http://leanpub.com/scrapingforjournalists

This version was published on 2014-04-02

Tweet This Book!

Please help Paul Bradshaw by spreading the word about this book on Twitter!

The suggested hashtag for this book is #scrapingforjournos.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#scrapingforjournos

Also By Paul Bradshaw

8000 Holes: How the 2012 Olympic Torch Relay Lost its Way

Model for the 21st Century Newsroom - Redux

Stories and Streams

Data Journalism Heist

For Joseph, who loves robots, Max, who likes asking questions, and Claire, who has all the answers.

Contents

CONTENTS

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/ScrapingForJournalists[1]
- 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

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 **
** (unordered list - normally displayed as bullet lists) or **
** (ordered list - normally displayed as numbered lists). The end of each list is indicated by either or .

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 ta-

ble 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...

3 Scraper #2: What happens when the data isn't in a table?

WEST MIDLANDS, ENGLAND
Wychavon District Council
Civic Centre, Queen Elizabeth Drive, Penshore, Wc

WEST MIDLANDS, ENGLAND
UPDATED **Wyre Forest District Council**
Civic Centre, Stourport-on-Severn, Worcestershire

API Get this info as **xml** or **json**

More often than not, the data that you want won't be presented in a handy table or list on a single webpage, so you'll need a more powerful scraper. In this exercise we're going to explore the concept of **structure**: why it's central in scraping, and how to find it.

And we'll do it with another Google Docs function: **importXML**

ImportXML - as you'd imagine - is similar to importHTML. It is designed for grabbing information from a webpage based on the parameters you give it.

But it is able to look for much more than a "table" or a "list", which means we can look for other types of structure.

Strong structure: XML

At its most basic, importXML allows us to scrape XML pages. XML is a heavily structured format - much more structured than HTML.

It is often used to describe products, people or objects in a database. For example, XML data for books might look like this:

```
<books>

  <book>

      <title>Online Journalism Handbook</title>

      <author>Paul Bradshaw</author>

      <author>Liisa Rohumaa</author>

  </book>

  <book>

      <title>Magazine  Editing  (3rd
      Edition)</title>

      <author>John Morrish</author>

      <author>Paul Bradshaw</author>

  </book>
```

```
</books>
```

The category 'books' has a 'book' in it, and that book has an 'author' and a 'title', and so on. It also has a second book, with another title and author, and so on.

This 'tree' of information, with branches of different types of information, may go down to deeper levels: we could have 'authors' within book, which in turn has multiple 'author' entries, and so on.

Many browsers - such as Internet Explorer - struggle to display an XML page, so if you try to view one it sometimes tries to download it instead. For best results try Chrome (which adds colour coding and other design elements which make it easier to understand) or, failing that, Firefox.

For a case study of how XML is provided by one organisation - the IOC - and used within the newsroom, read The New York Times's Jacqui Maher's article *London Calling: winning the data Olympics*[1]

Scraping XML

Let's say you need a list of all the councils in England, and it's available as an XML file. Here's an example of using importXML in a Google Docs spreadsheet to scrape that XML file:

---

[1]http://source.mozillaopennews.org/en-US/learning/london-calling-winning-data-olympics/

```
=importXML("http://openlylocal.com/councils.xml",
"councils/council")
```

Type this into any cell (save the spreadsheet first), press Enter, and after a few moments you should see the sheet fill with details of councils.

This function has similar parameters to importHTML (see the previous chapter) - but only two of them: a URL, and a query (`"councils/council"`).

To see what it's looking for, open the same webpage in a browser that can handle XML well. In other words, stay the hell away from Internet Explorer (as I say, Chrome is particularly good with XML or, failing that, Firefox).

That URL, again, is http://openlylocal.com/councils.xml[2] (how do you find this when wandering around the web? Look for a link to 'XML' - in this case, at the bottom of http://openlylocal.com/councils/[3])

You will see that the page has a very clear structure: starting with ‹councils› (ignore the `type="array"` bit), which branches into a series of tags called ‹council›, each of which in turn contains a series of tags: ‹address›, ‹authority-type›, and so on.

You can tell that a tag is contained by another tag, because it is indented after it. And you can collapse the contents of a tag by clicking on the triangular arrow next to it - so if you click on the triangular arrow next to the first ‹council› you will see that there's another one that follows it.

---

[2]http://openlylocal.com/councils.xml
[3]http://openlylocal.com/councils/

An XML page as it looks in a browser such as Firefox or Chrome

So the query in our formula...

```
=importXML("http://openlylocal.com/councils.xml",
"councils/council")
```

...looks for each ‹council› tag within the ‹councils› tag, and brings back the contents - each ‹council› in its own row.

And because ‹council› has a number of tags within it, each one of those is given its own *column.*

To show how you might customise this, try changing it to be more specific as follows:

```
=importXML("http://openlylocal.com/councils.xml",
"councils/council/address")
```

Now it's looking for the contents of ‹councils›‹council›‹address› - so you'll have a single column of just the addresses.

You could also be less fussy and adapt it as follows:

```
=importXML("http://openlylocal.com/councils.xml",
"councils")
```

...again, because ‹councils› contains a number of ‹council› tags, each is put in its own column.

Finally, try adding an **index** to the end of your formula. You'll remember that an index indicates the position of something. So in our first scraper we used the index 1 to grab the first table. In the importXML formula the index is added in square brackets, like so:

```
=importXML("http://openlylocal.com/councils.xml",
"councils/council[1]")
```

Try the formula above - then try using different numbers to see which ‹council› tag it grabs. We'll cover indexes more later.

When ImportXML doesn't work

If you use ImportXML with an XML file and it doesn't work, it may be because the XML is not properly formatted. You can test this with the W3School's XML validator[4].

In that case, a quicker option may be to use Open Refine[5] (formerly Google Refine). One this is running and you click 'Create Project', you can select the Web addresses (URLs) option (instead of uploading from your machine). In that window, paste the URL of the XML file and click Next - it should pick up that it's XML and ask you to pick the first 'node' - think of this as the first row in a spreadsheet. In the example above, this would be the first 'council'.

If that fails, you can try one of the numerous XML to CSV converters online (search for "XML to CSV converter"). Sometimes these will generate more than one CSV which you'll need to re-combine using a field both have in common (the spreadsheet formula VLOOKUP is ideal for this).

Once in Open Refine you can then export the file as CSV, HTML, XLS etc.

---

[4]http://www.w3schools.com/xml/xml_validator.asp
[5]http://openrefine.org/download.html

Recap

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

- **XML** is a structured language
- We can use structure in a language to '**drill down**' to particular elements, such as the contents of one tag within another tag.
- We can also add an **index** to specify which individual element we want, such as the first instance, second, and so on.

But I've not shown you the importXML function because you're likely to come across XML a lot (although it's perfect if you do). This function is even *more* useful when you're dealing with HTML pages, as we'll see in the next scraper...

Tests

Before that, however, it's worth testing the knowledge you've gained from this chapter and ways you might apply it. Here are some tests to try:

- Adapt your =importXML formula so that it uses cell references instead of strings - as you did in the last chapter.
- Change the formula so that you just grab all the *names* of each council (you'll have to work out what tag is used for those)

- Change it so you grab the 34th council in the list. I know there's no particular reason to do so. Do it *just because you can.*
- Find some other XML sources and try to scrape those. Here's a tip: include `filetype:xml` in your search (note that there's no space after the colon) if you're using Google - e.g. `'NHS filetype:xml'`
- Test yourself on drilling down through a tree of XML tags, and on using indexes
- Sometimes XML is used to provide information to Flash elements in webpages. In these cases try looking at the source code for the webpage hosting the Flash movie, and searching for 'xml' - if the movie is pulling data from an xml file this is where you might see what that file is. The link will probably be *relative* (i.e. it begins with / rather than `http://`. In this case add it to the site's *base* URL - for example if the flash page is http://bobswebsite.com/flashmovie and the xml file address is /flashdata.xml then the full URL of that file is probably http://bobswebsite.com/flashdata.xml (we'll cover relative and base URLs later in the book). Look at that file using a browser which structures the XML (Chrome is best). Secondly, use the Find facility (CTRL+F) to find a piece of information you want, and then see what tag it's in. For more on this search for information on 'using XML in Flash' - there are some useful videos that walk you through the process from the producer's side.

- Read up on the language being used to describe your query - it's called Xpath. There is a [tutorial on w3schools.com](http://www.w3schools.com/xpath/)[6] but lots of other tutorials exist if that doesn't work for you. Get used to searching for the one that suits you, rather than settling on the first you find.

---

[6]http://www.w3schools.com/xpath/

4 Scraper #3: Looking for structure in HTML
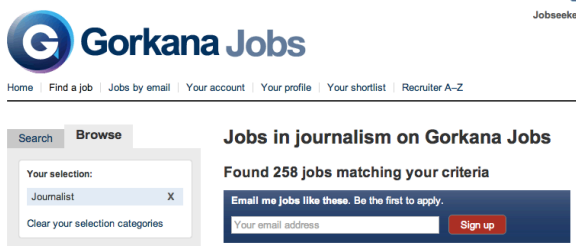
Now our story concerns the prevalence of unpaid internships in journalism. One useful source might be job ads. To grab that data and put it in a spreadsheet, here's an example of one formula that uses importXML to scrape vacancy details from the Gorkana jobs listing site:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='jobWrap']")
```

⚠️ *If you're copying this formula, make sure to check the quotation marks and inverted commas are straight, not curly. Better still, type it out yourself.*

As you can see, this has some very complicated-looking code in the second parameter, known as the query (`//div[@class='jobWrap'`

And to understand this, we'll need to take a detour into the structure of HTML.

Detour: Introduction to HTML and the LIFO rule

HTML (HyperText Markup Language) describes content on webpages. The 'markup' bit means that it marks up content in the same way as a sub-editor might mark up a reporter's copy.

It tells a browser whether a particular piece of content is a header, a list, a link, a table, emphasised, and so on. It tells you if something is an image, and what that image represents. It can even say whether a section of content is a piece of navigation, a header or footer, an article, product, advert and so on.

HTML is written in tags contained within triangular brackets, also known as chevrons, like this: <>. Here are some examples:

- The `<p>` tag indicates a new paragraph.
- `<h1>` indicates a 'first level' header - or the most important header of all. `<h2>` indicates a header which is slightly less important, and a `<h3>` header is less important, and so on, down to `<h6>`.
- `<em>` indicates that a word is emphasised. `<strong>` indicates a strong emphasis (bold).
- `<img` tells the browser to display an image, which can be found at a location indicated by `src="IMAGE URL`

```
HERE">
```

And so on. If you see a tag that you don't recognise, **Google it**.

Think of a tag as pressing a button: when you see a tag `<strong>` it is like pressing the 'bold' button on a Word document[1]. The tag `</strong>` (note the backslash at the start) turns the formatting 'off'.

The first tag is called an 'opening' tag, and the second a 'closing' tag. This is quite important for scraping because quite often it involves grabbing everything between an opening and closing tag.

Because tags can contain other tags, HTML is supposed to follow the **LIFO rule**: Last In First Out. In other words, if you have more than one tag turned 'on' (open), you should turn the last one off (close) first, like so:

```
<html>

   <head>

   </head>

   <body>

      <p>Words in

         <strong>bold

         </strong>
```

---

[1] http://html5doctor.com/i-b-em-strong-element/

```
        </p>

    </body>

</html>
```
In the above example, the `<strong>` tag is contained ('nested') within the `<p>` tag, which is nested in the `<body>` tag, which is nested in the `<html>` tag. Each has to be closed in reverse order: `<strong>` was the last one in, so it should be the first out, then `<p>`, `<body>` and finally `<html>`.

Oh, and they're not always conveniently indented as above, by the way.

Attributes and values

As well as the tag itself, we can find more information about a particular piece of content by a tag's **attributes and values**, like so:

```
<a href="http://onlinejournalismblog.com">
```
In this case:

- **<a>** is the tag
- **href** is the attribute
- "**http://onlinejournalismblog.com**" is the value. Values are normally contained within quotation marks.

You'll notice that only the tag is turned off - with - which is an easy way to identify it.

Here's similar code for an image:

```
<img src="http://onlinejournalismblog.com/logo.png"
/>
```

Again, the `<img>` tag tells us that this is an image, but where does it come from? The **src** attribute directs us to the source, and the 'value' of that source is "http://onlinejournalismblog.com/log

> *i* `<img` is one of the few tags which are opened and closed within the same tag: the backslash at the end of the image code above closes it, so you don't need a second, `</img>` tag. Other examples include the line break tag, `<br />` and the horizontal rule tag `<hr />`)

A single tag can have multiple attributes and values. An image, for example, is likely to not only have a source, but also a title, alternative description, and other values, like so:

```
<img src="http://onlinejournalismblog.com/logo.png"
alt="OJB logo" title="Site logo" />
```

Classifying sections of content: div, span, classes and ids

The use of attributes and values is particularly important when it comes to the use of the `<div>` tag to divide content into different sections and the `'id='` and `'class='` attributes to classify them. These sections are often what we want to scrape.

The Gorkana jobs webpage at http://www.gorkanajobs.co.uk/jobs/journalist/[2], for example, uses the following HTML tags to separate different types of content (to see these right-click on the page and View Source, then search for "<div" or another tag or attribute)

```
<div id="header" class="uk-site">
<div class="wrapper">
<div id="recruiters">
<div id="content">
<div class="content-wrapper">
<div class="content-inner">
<div id="primary">
<div class="fieldWrapper">
```

...in fact, there are around 150 different <div> tags on that single page, which makes the class and id attributes particularly useful, as they **help us identify the specific piece of content we want to scrape**.

And class and id attributes are not just used for <div> tags - the same page includes the following:

```
<li class="first">
<ul class="recruiterDetails">
<span class="buttonAlt">
<form class="contrastBg block box-innerSmall"
<label class="hideme"
<li id="job10598" class="regular">
<p class="apply">
<strong class="active">
<a class="page"
```

If the data that we want to scrape is contained in one of these tags, it again makes it much easier to specify, as we explore in the next section.

Back to Scraper #3: Scraping a <div> in a HTML webpage

Now that you know all this, you might be able to recognise some elements in the query of our importXML scraper:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='jobWrap']")
```

You can see that it contains the words 'div' and 'class'. And, spotting that, you might also search the HTML of that page for 'jobWrap' (try it). If you did, you would find this:

```
<div class="jobWrap">
```

This is the tag containing the content that the formula scrapes (until you get to the next </div> tag). And once you know that, you can customise the scraper to scrape the contents of any div class without needing to understand the slashes, brackets and @ signs that surround it - although we will come on to those.

This is often how coding operates: you find a piece of code that already works, and adapt it to your own needs. As you become more ambitious, or hit problems, you try to find out solutions - but it's a process of trial and error rather than necessarily trying to learn everything you might need to know, all at once.

So, how can we adapt this code? Here it is again - look for the key words **div**, **class** and **jobWrap**:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='jobWrap']")
```

Now try to guess how you would change that code to scrape the contents of this tag:

```
<div class="adBody">
```

The answer is that we just need to change the 'jobWrap' bit of the importXML scraper to reflect the different div class, like so:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='adBody']")
```

> *You may be wondering whether the upper and lower case letters matter. I could tell you, but invariably the best answer is: **try it, and you'll find out**. Waiting for someone else to tell you, or to find the answer somewhere else will just take you longer. Trial and error is quicker and makes you a better programmer - it's a useful habit to acquire.*

You can further adapt the scraper by using 'id' instead of 'class' if that's what your HTML uses, and replacing div with whatever tag contains the information you want to scrape. See if you can use that technique to adapt your scraper to grab the contents of each of the following tags listed previously:

```
<li class="first">
<ul class="recruiterDetails">
<span class="buttonAlt">
<form class="contrastBg block box-innerSmall"
```

```
<label class="hideme"
<li id="job10598" class="regular">
<p class="apply">
<strong class="active">
<a class="page"
```

Recap

We've covered quite a bit with this scraper, so here's a summary of the key principles:

- You can find structure in HTML if you **look for combinations of the tags, attributes and values** containing your data
- You can **adapt an existing scraper** to look for different sections of different pages by changing elements that you recognise
- **Trial and error** is an important technique in seeing what works and what doesn't - don't be afraid of making mistakes: you're not going to break anything (and if you're worried about losing data, just create a copy)

But trial and error is only the first step. When you hit a barrier, it's time to look at the documentation.

Tests

Once again, this book is not designed to show you how to write just one scraper, but how to understand the processes

behind writing scrapers generally. So, try some of the following challenges to see how you can adapt to different situations:

- Adapt your =importXML formula so that it uses cell references instead of strings - as you did in the last two chapters.
- Change the formula for some of those other tags and attributes that we listed. Why might some not work?
- Find a webpage that you check regularly and see if you can use importXML instead to gather that information.
- Set up an alert whenever the spreadsheet is updated - go to Tools > Notification rules
- Look for more tutorials on using importXML to gather data. Here is one example from distilled[3].

---

[3] http://www.distilled.net/blog/distilled/guide-to-google-docs-importxml/

5 Scraper #4: Finding more structure in webpages: Xpath

In scraping tip #1 I mentioned the importance of searching for **documentation** on the code that you're using. If we want to understand more about the importXML function we need to do just that.

If you search for 'importXML Google Docs', then, you'll find the Google Docs Help page for the function[1]. It says this about the 'query' part of the function's two parameters (which contains the funny code):

> "Query is the XPath query to run on the data given at the URL. Each result from the XPath query is placed in its own row of the spreadsheet. For more information about XPath, please visit http://www.w3schools.com/xpath/[2]."

Follow that link and you'll find much fuller explanation of XPath and what exactly those slashes, brackets and @ signs mean.

But it also gives you a clue for some other searches you can perform to **see if other people have written**

---

[1] http://support.google.com/docs/bin/answer.py?hl=en&answer=155184
[2] http://www.w3schools.com/xpath/

guides specific to the importXML function. Try searching for 'importxml xpath', for example, and you'll find:

- A tutorial from SEO Gadget[3]
- And The ImportXML Guide for Google Docs[4]

The latter includes a section explaining XPath, as well as a video and various other helpful tools. I won't duplicate them here - the point is that if you hit a problem you can often find other people online who have already tackled it - if you can find the right words to use in a search.

So, back to our formula. Now we've googled the documentation for the function, and followed that up with another search, we can start to use the information at those links to understand the code behind our scraper:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='jobWrap']")
```

The double slash:

```
//
```

...basically means a tag, so

```
//div
```

means

```
<div
```

The @ sign means an **attribute** of that tag - remember these are the words that appear within the tag but not at the start, e.g. `width`, `height`, `href`, `title`, `class`, `id`, and so on.

---

[3]https://seogadget.co.uk/playing-around-with-importxml-in-google-spreadsheets/

[4]http://www.distilled.net/blog/distilled/guide-to-google-docs-importxml/

That really confirms what we'd already worked out through trial and error - but it helps to have other examples and to have the whole thing explained. If you have any other questions, just google them. For example, if you missed what those square brackets mean, search for "Square brackets xpath" and you'll find various explanations, including one on the W3C pages you've already come across[5]:

> "**Predicates** are used to find a specific node or a node that contains a specific value. Predicates are always embedded in **square brackets**."

So these are predicates, and they're used to find 'nodes', which is just a fancy way of saying 'specific bits of content' - we already know what it does.

Square brackets - as you encountered earlier - can be used not just to specify that you want to grab contents with a particular attribute (such as a width) or an attribute value (such as a div class), but also a particular index (such as the first or second), position (such as last) and so on. Here are some of the examples they give:

```
/bookstore/book[1]
```

Selects the first book element that is the child of the bookstore element.

```
/bookstore/book[last()]
```

Selects the last book element that is the child of the bookstore element

---

[5]http://www.w3schools.com/xpath/xpath_syntax.asp

```
/bookstore/book[last()-1]
```

Selects the last but one book element that is the child of the bookstore element

```
//title[@lang]
```

Selects all the title elements that have an attribute named lang

```
//title[@lang='eng']
```

Selects all the title elements that have an attribute named lang with a value of 'eng'

So we've added some knowledge to what we already had. To reinforce that, try adapting your importXML scraper's code to scrape different elements of the page (or a different one) using each predicate.

You might also wonder what happens if you don't have square brackets. Again, those pages will answer your question: instead of grabbing the text between tags, you'll be grabbing the value of the tag itself, which is most useful if you want to **grab the value of a link** rather than the *link text*:

```
=importXML("http://www.gorkanajobs.co.uk/jobs/journalist/",
"//div[@class='jobWrap']//@href")
```

This is a key distinction to make so I want to pause a moment to emphasise it: **until now we have scraped text, but the formula above scrapes HTML**: specifically, the value of a tag's attribute (the tag being a, the attribute being href, and the value being found after href=").

Again, now that you have this knowledge, practise it by playing with your scraping formula.

Scraping a link also opens up a new possibility: we

could write a series of formulae that looked at each resulting URL and scraped *that* (assuming that they shared a common structure) - see scraper #5 below.

There's more to learn about XPath if you want to develop this further. For example, you can also use **string functions** (such as starts-with or contains) to be less specific about the attributes you're looking for. This page[6] has more on them, while below are some examples of importXML for scraping different bits of HTML which you can adapt appropriately.

HTML: `<div class="jobWrap">`

XPath: `//div[starts-with(@class, 'jobWrap')]`

HTML: `<div class="linktext"><h3>`

XPath: `//div[starts-with(@class, 'linktext')]//h3`

HTML: `<div class="linktext"><h3><a href="`

XPath: `//div[starts-with(@class, 'linktext')]//h3//@href`

HTML: `<span id="follower_count">`

XPath: `//span[@id='follower_count']`

HTML: *The first `<td>` in each `<tr>`*

XPath: `//td[1]`

HTML: *The first `<td><a href="`... *in each* `<tr>`*

XPath: `//td[1]//@href`

HTML: *Any linked text within* `<font face="Verdana">`
- see source for http://www.winvisible.org

XPath: `//font[@face="Verdana"]//a`

HTML: *The links themselves*

XPath: `//font[@face="Verdana"]//@href`

HTML: `<div class="title">`

---

[6]http://www.w3.org/TR/xpath/

XPath: `//div[@class='title']`
HTML: *First link within* `<div class="title">`
XPath: `//div[@class='title']//@href[1]`

For a series of examples of =importXML in action, see this spreadsheet[7], where the URL and the XPath code are in separate cells so you can see them more easily:

Also see the tutorials on the Online Journalism Blog at http://onlinejournalismblog.com/tag/importxml/[8]

Recap

We're almost done with Google Docs and importXML now. They've been very useful in teaching some basic concepts in scraping and coding that will come in very handy when we tackle more ambitious scrapers - while allowing us to see instant results without a huge amount of code.

The key things to take away from this journey into XPath documentation:

- **Google for key pieces of jargon** to see who else has tackled the same code and tools as you're exploring. For example we looked for 'importXML XPath' and 'XPath square brackets'
- We've been introduced to the idea of **predicates**, which we can use to specify the position (either an **index** or relationship to 'last' position) of the item

---

[7]https://docs.google.com/spreadsheet/ccc?key=
0ApTo6f5Yj1iJdGFuMVlsZzZySHNqZGhkdjNPMENtRHc
[8]http://onlinejournalismblog.com/tag/importxml/

we want, or particular qualities they have (such as attributes and values). This concept will reappear later - although it won't necessarily have the same name.

- We've touched on the difference between scraping text and *scraping the property* of HTML.
- We've also briefly encountered **string functions**, which do a similar job in identifying the qualities of a string of characters, for example, what characters a word begins with, or contains.

These are going to be very useful when we start scraping with more and more powerful tools and languages.

Tests

- If you didn't try them during the chapter, try adapting your importXML scraper's code to scrape different elements of the page (or a different one) using each predicate.
- Find a page with both links and link text. Alter your scraper so that it scrapes one, and then the other.
- Search within the documentation (use CTRL+F) at http://www.w3.org/TR/xpath/[9] for 'starts-with' and then try some of the functions listed alongside it, such as 'contains'. Not all the functions will work with the importXML function - or find anything on the page you're scraping - but make a note of some

---

[9]http://www.w3.org/TR/xpath/

of the results and keep a copy of the spreadsheet so you can refer to it and adapt it later.

- Try to scrape different pages with different problems - use the same techniques of trial and error, documentation, and adapting existing code to solve them.
- Have a play with the Firefox XPath Checker add-on[10] and see how that might help speed up your work (*Recommendation by reader Richard van Noorden*). Also try out Chrome DevTools[11] and the Firebug addon for Firefox[12], which allow you to select an element on a webpage and see the XPath you need (*both tips from Mark Schaver on the NICAR-L mailing list*).

---

[10]https://addons.mozilla.org/en-US/firefox/addon/xpath-checker/
[11]https://developers.google.com/chrome-developer-tools/
[12]http://getfirebug.com/

6 Scraper #5: Scraping multiple pages with Google Docs
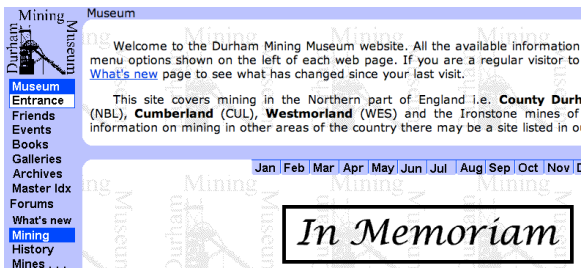
So far we have used functions in Google Docs to scrape single pages. Those pages have had progressively more complex structures:

- data in a table or a list;
- data in an XML file online;
- data held within a HTML tag (such as `<div class="jobWrap"`); and
- the properties of a HTML tag (such as the value of a link).

But what if you want to scrape data from more than one page? Well, if it's only a couple, then you might write

the same formula more than once on different sheets. You could also **use cell references to copy a formula down a series of cells**, each of which refers to a different URL - for example:

http://www.gorkanajobs.co.uk/jobs/journalist/	http://www.gorkanajobs.co.uk/jobs/pr/	http://www.gorkanajobs.co.uk/jobs/online/
=ImportXML(A1,"//h4")	=ImportXML(B1,"//h4")	=ImportXML(C1,"//h4")

Six cells in a Google Docs spreadsheet formula scraping different URLs

> ⚠️ *This can work either across a series of rows or down a series of columns, depending on what data you are bringing back: if the data takes up more than one column or row then bear in mind that it will overwrite over columns/rows*

This can be useful for a series of URLs, but if you have too many then you should start looking at tools other than Google Docs (in particular you can only use 50 importXML or importHTML formulae on a single spreadsheet. You can always copy and paste the results as 'values only' to get around that limitation, but it's still a lot of work if you have to do it more than once)

We're going to have to leave Google Docs and look at another Google tool to do this properly.

Recap

There are quite a few workarounds, or 'hacks', in scraping multiple pages with Google Docs, and these are worth considering occasionally if you have a project which suits it. Here are the key things to take from this short section:

- You can **copy a formula down** a column or across a row of cells and the formula will look at a different cell, with a different URL, each time - **if you're using a cell reference** (e.g. A1, where the URL is typed in cell A1) instead of a URL in your formula.
- If you're doing this, **try to write a formula which only grabs one piece of data** (or at least: one row, or one column) per URL.
- The Google Docs functions importHTML and importXML have **a limit of 50** per spreadsheet.
- You can get around this limit by copying your results and pasting them over the formulae using *Edit>Paste values only* - note that this means the results won't be updated when the pages change.

Tests

There's only one test for this chapter - but it's one that may take some time to figure out. Try to make sure you do take that time, because the process of working this out is going to come in useful when you come to later scrapers:

- Find a webpage with a list of links on it, and use importXML to scrape those URLs.
- Then, use another importXML formula and refer to one of the cells containing the results of that scraper, to scrape something from it
- Copy the formula down so it grab the same bit of data from a different URL (in each cell) each time

In other words, you should have one formula grabbing a list of URLs, then another formula copied down which grabs a bit of data from each URL in the list, in turn.

If you need a webpage to try this on, try the data linked from Durham Mining Museum[1] - this data is inconsistent, too, so you'll have to try to work out ways of dealing with that.

Here's an example of one way of dealing with it, if you need it[2].

---

[1]http://www.dmm.org.uk/mindex.htm

[2]https://docs.google.com/spreadsheet/ccc?key=
0ApTo6f5Yj1iJdDZ5RTFXcThPeExLcWt6dVJLZERhLWc
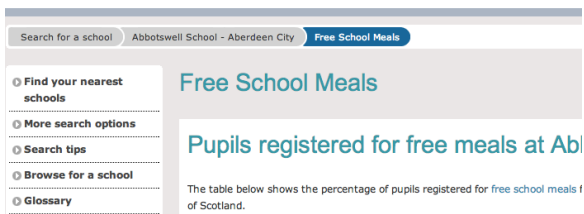
7 Scraper #6: Structure in URLs - using Google Refine as a scraper

Just as you would look for structure in a HTML page when scraping it, if you're scraping a series of URLs it is useful if you can find a structure common to them all.

If the webpages are generated by a database, it's quite likely that they will indeed share a common structure. Here, for example, are a series of pages containing Scottish schools data that I was asked to scrape for a *Scotsman* data blog post on poverty:

- http://www.ltscotland.org.uk/scottishschoolsonline/ school-

s/freemealentitlement.asp?iSchoolID=5237521[1]

- http://www.ltscotland.org.uk/scottishschoolsonline/ school-s/freemealentitlement.asp?iSchoolID=5237629[2]
- http://www.ltscotland.org.uk/scottishschoolsonline/ school-s/freemealentitlement.asp?iSchoolID=5237823[3]

In this instance, you can see that the URL is identical apart from a 7 digit code at the end: the ID of the school the data refers to.

Assembling the ingredients

With the **basic URL structure** identified, we already have half of our ingredients. What we need next is a list of the ID codes that we're going to use to complete each URL.

An advanced search for "list seed number scottish schools filetype:xls"[4] brings up a link to this spreadsheet (XLS)[5] which gives us just that.

---

[1]http://www.ltscotland.org.uk/scottishschoolsonline/schools/freemealentitlement.asp?iSchoolID=5237521

[2]http://www.ltscotland.org.uk/scottishschoolsonline/schools/freemealentitlement.asp?iSchoolID=5237629

[3]http://www.ltscotland.org.uk/scottishschoolsonline/schools/freemealentitlement.asp?iSchoolID=5237823

[4]https://www.google.co.uk/webhp?rlz=1C1GPCK_enGB454GB455&sourceid=chrome-instant&ix=heb&ie=UTF-8&ion=1#sclient=psy-ab&hl=en&rlz=1C1GPCK_enGB454GB455&site=webhp&source=hp&q=list+seed+number+scottish+schools+filetype:xls&pbx=1&oq=list+seed+number+scottish+schools+filetype:xls&aq=f&aqi=&aql=&gs_sm=e&gs_upl=74020l77151l0l77535l13l12l0l0l0l0l137l1079l7.5l12l0&fp=1&biw=1280&bih=856&ion=1&bav=on.2,or.r_gc.r_pw.r_cp.r_qf.&cad=b

[5]www.scotland.gov.uk/Resource/Doc/933/0031698.xls

The spreadsheet will need editing: **remove any rows you don't need**. This will reduce the time that the scraper will take in going through them. For example, if you're only interested in one local authority, or one type of school, sort your spreadsheet so that you can delete those above or below them.

Now to combine the ID codes with the base URL.

Bringing your data into Google Refine

Google Refine is available as a free download[6] - just search for it and download the version for your computer.

Once installed, it runs in your browser. If you have Microsoft's Internet Explorer set as your default browser, either change your default browser (Google Refine doesn't run well in Internet Explorer), or - once Google Refine is running - open another browser and type *127.0.0.1:3333* into the address bar.

Once you have Google Refine installed and running, open it in a browser and create a new project. Browse your computer to select the edited spreadsheet containing the school IDs (when importing, uncheck the option to *Parse as numbers*).

At the top of the school ID column click on the drop-down menu and select **Edit column > Add column based on this column**...

---

[6]http://code.google.com/p/google-refine/wiki/Downloads

In the New column name box at the top call this 'URL'.

Now we're going to use a language called **GREL** (Google Refine Expression Language). In the *Expression* box type the following piece of GREL:

```
"http://www.ltscotland.org.uk/scottishschoolsonline/
schools/freemealentitlement.asp?iSchoolID="+value
```

> *Type in the code yourself - the line above includes a space for formatting purposes which you should not include in the working code. Also, if you're copying quotation marks you may have problems*

The 'value' bit means the value of each cell in the column you just selected. The plus sign adds it to the end of the URL in quotes.

In the *Preview* window you should see the results - you can even copy one of the resulting URLs and paste it into a browser to check it works.

> *On one occasion Google Refine added .0 to the end of the ID number, ruining the URL. You can solve this by changing 'value' to* `value.substring(0,7)` *- this extracts the first 7 characters of the ID number, omitting the '.0'*

Click **OK** if you're happy, and you should have a new column with a URL for each school ID.

Grabbing the HTML for each page

Now click on the top of this new URL column and select **Edit column > Add column by fetching URLs**...

In the *New column name* box at the top call this 'HTML'.

All you need in the *Expression* window is 'value', so leave that as it is.

Click **OK**.

Google Refine will now go to each of those URLs and fetch the HTML contents. As we have a couple thousand rows here, this will take a long time - hours, depending on the speed of your computer and internet connection (it may not work at all if either isn't very fast). So leave it running and come back to it later.

Alternatively, if you're not too patient, you can cancel it and try again with only a couple of rows, just to see it working.

Extracting data from the raw HTML with parseHTML

When it's finished you'll have another column where each cell is a bunch of HTML. You'll need to create a new column to extract what you need from that, and you'll also need some GREL expressions explained in the documentation[7] (once again, googling for documentation on the language is a key method in learning how to code).

---

[7]http://code.google.com/p/google-refine/wiki/StrippingHTML

First you need to identify what data you want, and where it is in the HTML. To find it, right-click on one of the webpages containing the data, and search for a key phrase or figure that you want to extract. Around that data you want to find a HTML tag like `<table class="destinations">` or `<div id="statistics">`. Keep that open in another window while you tweak the expression we come onto below...

Back in Google Refine, at the top of the HTML column click on the drop-down menu and select **Edit column > Add column based on this column**...

In the *New column name* box at the top give it a name describing the data you're going to pull out.

In the *Expression* box type the following piece of GREL (Google Refine Expression Language):

```
value.parseHtml().select("table.destinations")[0]
.select("tr").toString()
```

(Again, type the code yourself rather than copying from here or you may have problems)

I'll break down what this is doing:

```
value.parseHtml()
```

parse the HTML in each cell (value)

```
.select("table.destinations")
```

find a table with a class (.) of "destinations" (in the source HTML this reads `<table class="destinations">`. If it was `<div id="statistics">` then you would write `.select("div#statistics")` - the hash sign representing an 'id' and the full stop representing a 'class'.

```
[0]
```

This zero in square brackets tells Refine to only grab the first table - a number 1 would indicate the second, and so on. This is because numbering ("indexing") begins with zero in Google Refine (as it does in most programming - Google Docs being an exception).

```
.select("tr")
```

Now, within that table, find anything within the tag `<tr>`

```
.toString()
```

And convert the results into a string of text.

Class names that use spaces

If you have a class name with a space in it, such as `<table class="stats section">` then you need to use full stops instead of the spaces. For example, `.select("table.stats section")` would not work, because `parseHtml` treats a space as a way of separating one element from another. Instead, then, you would write `.select("table.stats.section")`. Seems a little odd, but it works.

The results of that expression in the Preview window should look something like this:

```
<tr> <th></th> <th>Abbotswell School</th> <th>Aberdeen
City</th> <th>Scotland</th> </tr> <tr> <th>Percentage
of pupils</th> <td>25.5%</td> <td>16.3%</td> <td>22.6%</td>
</tr>
```

This is still HTML, but a much smaller and manageable chunk. You could, if you chose, now export it as a spreadsheet file and use various techniques to get rid of the tags (Find and Replace, for example) and split the data into separate columns (the =SPLIT formula[8], for example).

Or you could further tweak your GREL code in Refine to drill further into your data, like so *(remembering to type the code yourself rather than copying)*:

```
value.parseHtml().select("table.destinations")[0]
.select("td")[0].toString()
```

Which would give you this:

```
<td>25.5%</td>
```

Or you can use the **htmlText** function (detailed in the GREL documentation here[9]) to grab the text content rather than the tags, like so:

```
value.parseHtml().select("table.destinations")[0]
.select("td")[0].htmlText()
```

Or - if your data was consistently positioned - you can add the .**substring** function to strip out the HTML by focusing on the characters from one position to another (5 and 10 in this case) like so:

```
value.parseHtml().select("table.destinations")[0]
.select("td")[0].toString().substring(5,10)
```

---

[8]http://excelnotes.posterous.com/splitting-a-vote-or-other-piece-of-data-into

[9]http://code.google.com/p/google-refine/wiki/GRELOtherFunctions

The .substring function needs two parameters (in parentheses): the start point and end point that it is extracting from and to. In this case, starting at character 6 (indicated by 5 - remember that indexes start counting from zero in Refine) and ending at character 11 (which it does not grab). For more on this and other functions, check the documentation.

When you're happy, click **OK** and you should have a new column for that data. You can repeat this for every piece of data you want to extract into a new column.

Then click **Export** in the upper right corner and save as a CSV or Excel file.

Recap

Google Refine isn't designed for scraping - it's a data cleaning tool, and its 'fetching URLs' command is really designed to fetch simple feeds of structured data (such as XML or JSON) rather than whole HTML pages.

So by using it to grab whole pages rather than small parts of them, this means it takes a lot of time. But the above process does teach a valuable lesson about scraping. Here are the key takeaways:

- **Look for structure in URLs** as well as in the HTML of each page.

- **Find a source of data that completes that struc-ture**. In this example we used school codes, but others might be the names of regions or countries, postcodes, names, years or other dates. A good way to do this is to search for specific jargon that would only be found in a list of data, and add the search limit filetype:xls.
- **Combine the two to get a list of unique URLs that share a common structure**, which can then be scraped. In this example we used GREL to do that, but you could also use the =CONCATENATE function in Google Docs or Excel (search for the documentation)

Later we'll use these same techniques in a more ad-vanced scraper which isn't so time- and bandwidth-heavy.

Tests

- Try the same process with postcodes and JSON by following the steps on this tutorial on the Online Journalism Blog[10]
- Try the other possibilities listed at the end of that post with charities or local authority data.
- Try a search on the European Investment Bank web-page[11] for financed projects. Look at the URL that's

---

[10]http://onlinejournalismblog.com/2010/12/16/adding-geographical-information-to-a-spreadsheet-based-on-postcodes-google-refine-and-apis/

[11]http://www.eib.org/projects/loans/list/index.htm

generated by your search - what structure is there, and how might you generate your own URLs? (Clue: look at the source code on the search page)

- Try other functions in GREL that might help you add data to a spreadsheet. You can find recipes at http://code.google.com/p/google-refine/wiki/Recipes[12] - I also maintain bookmarks on GREL at Delicious.com/paulb/grel[13]

- Try to achieve the same results in Google Docs by adding the IDs to the URLs (you can use =CON-CATENATE or the & sign) and then using the importXML function to grab from each webpage. Note: you'll only be able to use the function 50 times.

---

[12]http://code.google.com/p/google-refine/wiki/Recipes
[13]http://delicious.com/paulb/grel

🗨 Reader notes

Soren Jones *notes of the problem with Google Refine adding .0 to the ID number:*

"The issue is Excel storing the numbers as numbers not text. To change the numbers to text in Excel, see *Three ways to convert numbers to text*[14]. I suggest the third way. In my experience with Excel for Mac 2011, the first way doesn't work (the cell is formatted as text aligned left, but the number is still stored as a number, if that makes sense).

"Another option, if you want to import all numbers in a spreadsheet as text, is to save the Excel file as a CSV file. And then uncheck "Parse cell text into numbers, dates, ..." at the *Configure Parsing Options* step. All numbers in the preview window will change from green (numeric) to black."

Soren Jones *also adds of the substring function:*

"As the data length can vary, other readers might also find this useful:

```
with(value.parseHtml().select("table.destinations")
[0].select("td")[0], e, e.htmlText())
```

"Which would give you this:

```
25.5%
```

"The result of `value.parseHtml().select("table.destinations")[0].select("td")[0]` (`<td>25.5%</td>`) is assigned to a variable (e.g., e), and then `htmlText()` returns the text from within an element (including all child elements). In the example, `ownText()` would produce the same result as `htmlText()`.

"More at http://code.google.com/p/google-refine/wiki/GRELControls[15]   and

[15]http://code.google.com/p/google-refine/wiki/GRELControls
[16]http://code.google.com/p/google-refine/wiki/GRELOtherFunctions
[17]http://code.google.com/p/google-refine/wiki/StrippingHTML

8 Scraper #7: Scraping multiple pages with 'next' links using Outwit Hub

Another thing to look for if you are scraping multiple pages is '**next**' **links** that take you from one page of data (often results from a search) to the next. If the data you want is presented in this way then the scraping job may be relatively straightforward.

OutWit Hub is a tool designed to tackle exactly this situation. There's a free Light version which will scrape up to 100 rows of data, and a very affordable Pro version (around 50 euros at the time of writing) which does a lot more. The software is updated, too, and has had some particularly useful features added recently which we'll come on to. For this task, the Light version should do - then you can decide whether you want to pay for the Pro version. (Although we'll be covering how to do the same thing in Scraperwiki later, OutWit Hub is still useful for performing a scraping job if you haven't got the time to code but have got the time to leave OutWit Hub running on a job).

Download OutWit Hub from the Outwit website. You can either use it as a Firefox add-on or a standalone applica-

tion. As the tool started as an add-on, most tutorials you'll find will be written with that use in mind (the application was only introduced with a more recent version), so I'll write this assuming that you're using OutWit as an add-on.

If you are using the OutWit Hub add-on, a special OutWit button should appear on Firefox's navigation to the left of the address box (If not, go to **Tools > Add-ons** and look for it in the **Extensions** menu. Click **Enable** if it's available, or look for other instructions).

If you click on that button while on a page that you want to scrape, a new window will open titled OutWit Hub where you can start to customise your scraper.

A new alternative to OutWit Hub is the Chrome extension Scraper[1]. Michelle Minkoff provides a useful guide to using that tool in *Web Scraping without Programming â€" NICAR 2012 â€" Hands-On Tutorial*[2].

---

[1]https://chrome.google.com/webstore/detail/scraper/mbigbapnjcgaffohmbkdlecaccepngjd?hl=en

[2]http://michelleminkoff.com/web-scraping-without-programming-nicar-2012-hands-on-tutorial/

Creating a basic scraper in OutWit Hub

OutWit Hub comes with its own tutorials (once you start using it the tutorial starts up - it's well worth clicking through), so I won't repeat that material here. The key thing is to go to the first page that you want to scrape, and then click on the OutWit button next to the address bar to open it up.

Once OutWit Hub is open, you have a number of options. If the data is in an obvious structure, such as **tables** or **lists**, then click on the relevant option on the left, under **data**. There's also the '**guess**' option, which can be quite useful - try all three on the Gorkana jobs listing page we used earlier: you'll notice that the 'guess' option is actually pretty spot-on.

If any of those options work in showing you the data you need, then great. You can simply click **Export** (bottom right) to save the data as CSV, Excel, or other formats.

But best of all: if you want to **scrape subsequent pages by following 'Next' links**, you can tell it to do that too, by clicking the button next to the address bar that looks like a 'fast forward' button (if you hover over it, a box should appear saying "**Auto browse through series of pages**").

Before you do that, however, make sure to change two boxes at the bottom of OutWit in the *On page load* section: firstly, *untick* **Empty** - otherwise it will empty its 'cache' of previous results when it loads a new one. Secondly, *tick* **Catch selection** - this will do exactly that.

Note that to the left you can also tick **Deduplicate** if you don't want any duplicate entries. My personal preference, however, is to do that cleaning afterwards - you want the raw data to be as raw as possible, and sometimes a lot of duplicate entries may be newsworthy or noteworthy itself.

Try this with the Gorkana jobs page³) on the 'guess' view or, let's say you wanted to analyse the makeup of torchbearers nominated through one of the major Olympic torch relay sponsors, try to scrape the tables generated by an empty search on the Bank of Scotland's list of Olympic torchbearers⁴ (perform the empty search first before launching OutWit on the first page of results).

How long it takes to finish depends on the number of pages and the size of each page: the Gorkana jobs pages, for example, have a lot of code and so take longer to load. With many of these jobs you may have to leave it running overnight. Once you've finished, though, click **Export** as before to grab the results in spreadsheet form.

Customised scrapers in OutWit

If none of the default options (tables, lists, guess) work then you'll have to try to make your own. To do this, click on the '**scrapers**' view on the lower left menu (under **automators**), click on **New**, and give it a name.

At this point you should be in pretty familiar territory.

---

³http://www.gorkanajobs.co.uk/jobs/journalist/

⁴http://www.bankofscotlandlondon2012.co.uk/In-your-community/ Olympic-Torch-Relay/search/?pg=0

You have a table to fill with the following:

- 'Description' - this is the column heading for this particular piece of data
- 'Marker before' - this is the code or text that always appears before your data: normally a piece of HTML like `<div class="jobWrap">`. Actually, `"jobWrap">` would be more than enough
- 'Marker after' - this is the code or text that appears after it, e.g. `</div>` - again, just `<` is often enough to make this work, if there are no other chevrons before it.

The rest is optional, but does the following:

- use the **Separator** option if there's more than one piece of data separated by a comma, for example - and use the List of labels option to say what each one should be called
- use **Replace** to add extra text to your data, such as adding the root URL (e.g. http://site.com/) to a relative URL (e.g. /jobs/journalism)
- use **Format** to further specify the data you want.

By now you'll know that if you want to explore these optional extras further, you should search for documentation mentioning these key terms and 'OutWit Hub' - you'll probably end up on OutWit Hub's own Help pages such as this one[5]

---

[5]http://www.outwit.com/support/help/hub/helpcenter/editingScrapers.xhtml

An example scraper - grabbing job ads from the organisation G4S[6] is shown below:

Example scraper in OutWit Hub

However you construct your scraper, once it's ready, click **Execute** towards the upper right corner of the scraper area. You'll then see the results of the scraper. If you want to collect more than one page, use the 'fast forward' button as before - but make sure you have unticked the '**empty**' box under 'on page load', and ticked 'catch' instead - this basically means that when each page loads, the data is caught, rather than emptied.

As before, once it's finished, click **Export** (under the results, to the right) to export as a CSV file.

The export option

---

[6]http://careers.g4s.com/search-job.php?showall=y&showDistance= N&location_range=&SortBy=3&pageNum=14

Recap

OutWit Hub is an excellent tool to get to know. This section has really just introduced you to the tool and its basic workings, which you should be familiar with from previous sections. But revisiting those in a different environment is a good experience of how these principles recur:

- OutWit Hub is **designed to look for particular structures** such as tables, lists, links, and so on.
- But best of all, it is **designed to find and follow 'next' links**
- You can create customised scrapers that **look for specific structure in** the HTML surrounding your data.

There's a lot more that you can do with OutWit Hub - for example grabbing documents from links on a page. As always, explore the documentation created by OutWit and by its users to find out more.

We're not going to explore all of that, but the next couple of chapters will explore some of the tool's specialist functionality in order to teach some useful general scraping problem-solving techniques, including playing with some useful code.

Tests

- Look for documentation that explains how to scrape the contents of a list of links - this is the 'down'

button on the menu

- Find other pages to scrape and create customised scrapers for those
- Find out how to identify images on a page and scrape them
- Likewise, how can you scrape email addresses - or documents?
- Look at the other options under **Automators** - see if you can find documentation on them (most of these are only available if you pay for a full licence)
- If you're really ambitious, explore how you can use **Regex** with Outwit hub to scrape webpages when a particular type of expression occurs

9 Scraper #8: Poorly formatted webpages - solving problems with OutWit

Ed Miliband: List of meetings and dinners with donors and trade union general secretaries

Tweet 8   Like 2 people like this.

30 March 2012

Ed Miliband MP today published details of all meetings and dinners with donors since he became party leader.

The list goes further than Mr Cameron's donor details as it lists all meetings and dinners with donors and trade union general secretaries in Mr Miliband's office or home since 25 September 2010.

It also includes any donor which has given more than £7,500 – far lower than Mr Cameron's decision to only publish donors over £50,000.

So far we've dealt with data that is structured enough to make it straightforward to scrape. But what if you encounter a page which is inconsistently formatted?

Two good examples of such poorly formatted data are the UK Labour Party's publication of meetings and dinners with donors and trade union general secretaries - at http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30[1] - and pilot groups of surg-

---

[1] http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30

eries in the UK who are controlling local health spending. They are called pathfinder consortia and listed at http://healthandcare.dh.gov.uk/context/consortia/[2]

The problem with the dinners list is that some dates cover more than one person. On March 3 2011, for example, the entry reads: "Len McCluskey (UNITE), Paul Kenny (GMB), Dave Prentis (Unison)".

The problem with the consortia list is that the regions are used as headers before a list of those consortia within that region.

In other words, in both cases we have one piece of data (the date, the region) which we want to associate with multiple pieces of other data (the names, the consortia). We also have - as you'll see - some pretty cruddy HTML hiding behind it too. Be patient with this chapter: I've intentionally chosen a frustrating webpage to take you through ways of dealing with them. The point here is not the data you get in the end, nor even the tool and its functionality, but how you think about the problems with the webpage containing your data.

Identifying what structure there is

It's unusual for a page to have no structure at all, so look at the source HTML (right-click somewhere on the page where there isn't an image or something else, and select **View source** or similar) and use CTRL+F to find the first piece of text you're after.

---

[2]http://healthandcare.dh.gov.uk/context/consortia/

In the consortia page, that text is 'East of England' - but
make sure you find the right one: the phrase is used four
times (in menus, for instance) before the first use we want
to grab: as a header for a series of consortia names.

That heading uses a header tag, like so:

```
<h3 id="eoe"><strong>East of England</strong></h3>
```

What's the structure that we can use to identify it in
OutWit Hub? We have a few options:

- We could use the `<strong>` tag as a marker before.
  Try it: you'll find there is other text using that which
  is not a header.
- `<h3` then is a better choice. And we can use `</strong>`
  as a marker after. In fact, we can just use `</`

Once again, **trial and error** has an important part to
play in this process.

Create a new custom scraper in OutWit Hub (under the
automators heading on the left).

In the **Description** column, call your first line *Region.*
In the **Marker before** column put:

```
<h3
```

���nd in **Marker after** put:

```
</
```

Click **Execute** to run it.

This will capture those regions, but also half a dozen
pieces of text from the end, such as 'Local sitemap' and
'Tag cloud'.

We need to be a bit more fussy. So amend our *Marker
before* to look for:

```
<h3 id=
```

This will get rid of all but one of those erroneous entries.

> Make sure that the scraper results screen doesn't have *Catch selection* ticked or you'll still see the results of your previous attempt - instead tick **Empty**. We can deal with that later.

Oh, and if your scraper is grabbing the HTML as well and you want to leave that out, tick the box marked **Clean text** underneath the results of the scraper where it says *Options* (I prefer to leave it off while testing so I can see when it's grabbing HTML but nothing else).

Now to grab the name of each consortium in each region. Looking at the HTML, this appears within the tags `<p><strong>`, like so:

```
<p><strong>Arterial Community Interest Group</strong><br
/>
```

So, set up a second row in your scraper with a *descripton* of 'Consortium' and the *Marker before* of:

```
<p><strong>
```

The *Marker after* once again is:

```
</
```

Click **Execute** and you'll see an immediate problem: the scraper has created a row for each region, and *multiple* columns for every consortium within that - making 43 columns in total.

Now there are ways to clean that up in a tool like Google Refine, but it would be much better - and easier - to sort it out here.

Repeating a heading or other piece of data for each part within it

In OutWit Hub, you can solve this with the **#repeat# directive**, which is used as follows:

In the **Description** column of your OutWit scraper, add #repeat# *directly before* the name you've given to that cell, e.g. `#repeat#Region` - this will:

- Give that column a Description of 'Region',
- Scrape the first match
- Repeat the result of that scrape as long as there is *new* data being gathered in other lines - until it finds *another* match
- It then repeats the same process again.

Change that first line of your scraper, then, so that the *Description* reads **#repeat#Region**, and click **Execute**.

You should now only have two columns: one for region, and one for consortium. If you're using the Pro version of OutWit Hub, the region will be repeated for each. If it's the free Light version of OutWit, your region won't be repeated, but you will have solved the columns problem (you will also only have 100 rows of results).

You can find out more about directives on the OutWit documentation[3]

With that problem solved, you can continue to add further data - testing after each one, because there are other problems lying in wait...

The next row would be geographical area, and the *Marker before* this would be:

```
Geographical area: </strong>
```

The *Marker after* would simply be

```
<
```

Clicking **Execute** reveals yet another multiple columns problem. Further examination reveals the cause: under the North West region there is a section which bundles together five consortia (CCGs) under "Ashton Wigan and Leigh with five CCGs".

We could use #repeat# again with Consortium to solve this problem - but this fails to grab the names of each consortium, each of which would still be simply "Ashton Wigan and Leigh with five CCGs".

So we'll have to tweak that Consortium line to grab the proper names rather than the subheading which has messed things up.

Those five CCGs have an extra piece of HTML styling like so:

```
<p style="padding-left: 30px;"><strong>1-ALPF
Health Commissioning Consortium</strong><br />
```

We now can't be so sure about the *marker before*

---

[3]http://www.outwit.com/support/help/hub/helpcenter/editingScrapers.xhtml

because the HTML isn't formatted consistently enough -
but the *marker after* might offer an alternative. Change the
Consortium line of your scraper so that the *marker before*
is simplified as:

```
<strong>
```

...and the *marker after* becomes more specific:

```
</strong><br
```

Click **Execute** and our Ashton, Wigan and Leigh con-
sortia are now found.

But we have a new problem!

There is an extra column once again - this time for
Geographical Area.

There are only two entries in this second column,
so *click on the top of the column twice to sort it - first
ascending and then descending* - which will bring the
culprits to the top.

Then, **look in the source HTML of the page you're
scraping for the section responsible**. There are only two
mentions of 'Basingstoke', on sequential lines, so a search
for this leads you there quicker.

Once you do find it, you'll have to look very closely at
the HTML. What is different?

```
<p><strong>Calleva (Previously Basingstoke</strong>)<br
/>
```

```
<strong>Geographical area: </strong> Basingstoke<br
/>
```

```
.strong.ropulation...... ./strong.......,...strong...s./.
</strong></p>
<p><strong>Calleva (Previously Basingstoke</strong>)<br />
<strong>Geographical area: </strong> Basingstoke<br />
<strong>Number of practices: </strong>21<br />
```

Why does this code cause your scraper a problem?

The answer is that there is a bracket between `</strong>`
and `<br />` - and in the Consortium line of our scraper it
is looking for `</strong><br` without anything in between.

There's a similar problem with the other culprit:

```
<p><strong>Wakefield Alliance<br />

Geographical area: </strong> Wakefield<br />
```

Here, the name of the consortium ends with `<br />` -
the `<strong>` tag is not closed at all.

Here are two possible solutions:

- Change the *marker after* to be *less* fussy and just
  look for `<br />` - try it: you'll find that this brings in
  too many results that don't fit.
- We could also shift the focus of the *marker after* to
  the right so that it looks for `
 Geographical
  area` - try it: it generates results in one row and 266
  columns. Nasty.

Instead, there are two other ways of solving the prob-
lem: **separators**, and **regex**.

Splitting a larger piece of data into bits: using separators

Maybe we've been looking at this data the wrong way.
Instead of seeing each part - name, location, population,

practices - as a separate piece of data to be grabbed, we
could instead **grab them all one bunch after another** -
consortium 1, consortium 2, and so on - which can then be
split up.

This, in fact, is how the HTML is coded:

```
<p><strong>Arterial Community Interest Group</strong><br
/>
```

```
<strong>Geographical area: </strong> Basildon
&amp; Billericay<br />
```

```
<strong>Number of practices: </strong> 12<br />
```

```
<strong>Population size: </strong> 62,657</p>
```

Notice that each consortium is wrapped in a `<p>` tag -
a paragraph. This actually gives us a very simple *marker
before*, while the closing tag `</p>` gives us an obvious
marker after.

By doing it this way, the clumsy and inconsistent
HTML inside those tags is no longer a problem. Although
we will have a new one.

If you want to get rid of previous lines in your OutWit
scraper, you don't have to delete them immediately: instead
you can untick them to make sure that they don't run.

So, **untick all the lines you've created so far**, *apart*
from *Region*. This means that they won't run when you
next click *Execute*.

Add a new line called *All details*. Now we can put as
our *marker before*:

```
<p>
```

...and our *marker after* is:

```
</p>
```

If you click **Execute** now, you will get two columns: one
with all the details for each consortium, bundled together;
and a second with the region data.

To separate those details into something more useful,
we need to use the **Separator** column. This is only available
in the Pro version, so if you don't want to pay for that,
you'll have to split it in a spreadsheet package or with the
free data cleaning tool *Google Refine*.

The *Separator* column specifies anything that you want
to split data on. There are two obvious candidates in our
data: within the `<p>` tag each line starts with `<strong>` and
ends with `<br />`

Try one, and then the other. Which do you prefer?

For me, the `<br />` option produces the cleaner results.
It means we now have one column for the consortium
name, one for the geographical area (*All details2*), and oth-
ers for practices (*All details3*), and population (*All details4*).

There are also two extra columns. Sorting these brings
one result to the top: Bracknell and Ascot. Looking at the
source HTML of our webpage we can see this is because
there is an extra section before that consortium which
generates extra tags. As it's only one row it will be simple
to clean this up in a spreadsheet after (make a note to).

We can also clean up the extra rows of data generated
which contain other information from this page. The sim-
plest way to do this is sorting too: when you open this in
a spreadsheet, sort by *All details4* (this is the last column
which *should* have data, and so the one that would work
best) so you can delete the rows with no entry.

Another piece of cleaning to do is removing the description text repeated in each row of your results before the numbers and names we actually need ('Number of practices:' etc.). A simple Find and Replace for each column in your spreadsheet package should solve this. Remember to format the columns as *numbers* afterwards, so calculations work properly.

Before all that, however, you might also want to change the column headings from *All details2*, *3*, etc. to something more meaningful. If you want to do this, use the column in your scraper next to *Separator* titled *List of Labels*.

In this case, on the row containing the *All details* scraper, double click in that column and type the following:

```
Name, Area, Practices, Population
```

This is a list of labels to apply to the columns generated by that line's scraper, in order. In other words, the pieces of data grabbed before the *first* separator will be labelled 'Name'; the data grabbed next will be labelled 'Area', and so on until all four labels have been used. Any extra columns will be labelled *All details5*, *All details6*, etc. by default.

Once you've done this, execute your scraper one final time, and use the **Export** button to save a CSV of the results.

Recap

This chapter focused on some of the specific functionality in OutWit, but it was really about problem-solving techniques:

- Ugly webpages can have structure, and professional-
  looking ones can lack structure in the HTML behind.
  Don't let either put you off attempting to scrape a
  page.
- Structure is in the mind of the beholder. **Look for
  repetition and patterns** both in the HTML used
  around the data, and the non-HTML text, such as
  prefixes, headings, and even numbers of characters.
  For example, dates may all be different, but they
  are likely to share the pattern 'two digits, space
  (or slash), two digits, space (or slash), two digits'.
  Postcodes, quantities of money and phone numbers
  are a similar 'structured' type of data.
- Use the #repeat# directive to add a section header to
  other pieces of data listed underneath.
- Use **trial and error** to try out your own ideas about
  possible structure (but make sure you *don't* have
  Catch selection ticked on the results page or you'll
  keep seeing the results of previous trials).
- **Sort the results** of your experiments to spot outliers
  generating extra columns where problems have oc-
  curred.
- **Take it step by step** to identify problems at each
  stage and solve them individually, rather than hav-
  ing to unpack several at once.
- Look closely at the source HTML around pieces of
  data that aren't being scraped properly - **how might
  the structure of the HTML there differ from the
  structure you're looking for?**

- When trying to solve a problem, **switch from being
  more specific to being less specific**: for example,
  you might start by only specifying one tag, then add
  another to be more specific, or take away all but the
  closing bracket of a tag to be less fussy.
- Take a step back from the line you're focusing on,
  and **look at the wider code**. For example, is there
  something on the preceding or following line of
  HTML you could use in the *marker before* or *marker
  after*? Is there a bigger code block that you could
  scrape in one line, then use **separators** to split up?

Tests

- *Find a similar 'list of' webpage* and see what prob-
  lems you get with trying to scrape it with OutWit.
  Do you need to scrape it in chunks and then split
  those with separators?
- Go back to the page we scraped in this chapter. *What
  other separators could you use?* (Tip: don't just look
  at HTML tags). Try them - what are the problems?
  Look at the HTML to work out why.
- If you have the Pro version of OutWit, *try some of the
  other directives* listed at http://www.outwit.com/support/help/hub/he
  - see if you can find one (or two) which would solve
  the problem of the extra rows of data we don't need.

---

[4]http://www.outwit.com/support/help/hub/helpcenter/editingScrapers.
xhtml

- Try to tackle the UK Labour Party's publication of
  meetings and dinners with donors and trade union
  general secretaries - at http://www.labour.org.uk/list-
  of-meetings-with-donors-and-general-secretaries,2012-
  03-30[5] - this is even uglier.

---

[5]http://www.labour.org.uk/list-of-meetings-with-donors-and-general-
secretaries,2012-03-30

10 Scraper #9: Scraping uglier HTML and 'regular expressions' in an OutWit scraper

In the last chapter I introduced you to poorly formatted data. We dealt with one particularly frustrating page on new bodies in the UK health service, but only briefly mentioned the UK Labour Party's publication of meetings and dinners with donors and trade union general secretaries - at http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30[1]

I'm sorry to have to tell you: this one is even worse.

The good news is, once you've tackled this page, you'll realise how lucky you are when you come across data that's been published with more structure.

Introducing Regex

One of the most powerful tools in a scraper's arsenal is its ability to look for a particular pattern of words, digits or characters that recurs. So far we've been very specific about that pattern - but what if you can't be that specific?

---

[1] http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30

If that's your problem then '**regex**' - short for 'regular expression' - may be one solution.

Regex looks like a jumbled bunch of nonsense, but is in fact just a code.

Some of it makes sense, like this:

`[a-z]`

Which means 'any lower case character from a to z'.

And this:

`[0-9]`

Which means 'any character from zero to nine'.

And some of it makes no sense at all when you first see it. But hopefully that should change by the end of this chapter.

To show how it works, we'll start with using it to specify a range of characters.

Using regex to specify a range of possible matches

Let's look at the UK Labour Party's publication of meetings and dinners with donors and trade union general secretaries - at http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30[2]

Let's say you're interested in **any entries from June or July 2011**.

In OutWit Hub, open up that URL, and go to the **scrapers** section. Click **New** to create one. Give it a name.

---

[2]http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30

In the first line of this new scraper, give it a *description*
of 'June July 2011' and in the *Marker before* section type
this regular expression:

```
/Ju[nl][ey]/
```

What does this mean? Firstly, the *forward slashes* at the
start and end tell OutWit that this is regex. These are what
is called an **identifier**, but don't worry about the jargon.

Our regex proper then starts with two letters, and a fur-
ther pair of letters inside some square brackets. Everything
within square brackets represents **character options**. In
other words, if there is a match for *either* of the lower case
characters 'n' or 'l' (in the first square brackets) and that
character is followed by *either* of the lower case characters
'e' or 'y' (in the second square brackets) then the scraper is
happy.

Before those square brackets, however, we have **non-
optional characters** which are *not* in square brackets: our
scraper is looking for two of these: a capital 'J' followed by
a lower case 'u'.

Put another way, our regex is saying this:

- look for 'Ju'
- followed by an n or l
- followed by an e or y

In other words, any instance of 'June' in the webpage
we're scraping would generate a match here, as would
'July' - but *not* 'june' or 'july' because they start with lower
case letters. To add these you could change your first letter

J to another set of square brackets to provide two options
like so: [Jj].

Because this regex has been entered in the **Marker
before** column, this scraper will look for a match against
any of those cases, and grab the data until whatever is
specified in the **Marker after** column.

So, now enter the ‹ character in *Marker after* (which
would be used at the first html tag after any text), and click
Execute to run the scraper

The results should look something like this:

- 2011 - Len McCluskey (UNITE)
- 2011 - John Hannett (USDAW)
- 2011 - George Guy (UCATT)

Now go back to your scraper and try some other
variations - e.g. so that it only grabs '2011' or 'George', or
anything *before* 'UNITE'.

Catching the regular expression too

The problem, of course, is that we're missing the actual
words 'June' or 'July', because we used those as the marker
before what we wanted to scrape.

To grab the expression that we're looking for, rather
than just using it as a *marker before*, use the **Format**
column. This specifies the format of what we're looking
for rather than what comes before or after it - and it *always*
uses regex.

Go back, then, to your scraper, and in that **Format** column type the same regex as before:

```
/Ju[nl][ey]/
```

...and delete the same regex from your *Marker before* column.

In the *marker after* column enter:

```
2011
```

Now click **Execute**.

This time you will find the opposite problem: you now have a series of rows which go: June, June, June, July, July, and so on.

This is what's happened: because you've asked for a match on anything beginning with Ju, followed by an n or l, and then followed by e or y, that's all you've grabbed: the words June or July (specifically, when used with your *marker after*: words June or July that occur directly before '2011' - other instances will not be grabbed).

We need more regex to specify what else we want.

I want *any* character: the wildcard and quantifiers

One of the most useful characters in regex is the full stop. This is a **wildcard**, meaning that it can stand for any character (just as a Joker is often used in cards) - except for line breaks and new lines.

So the following:

```
f.ll
```

...can mean 'fill', 'fall', 'full' or 'fell' (or indeed 'foll' or
any other word - note that this would grab the 'foll' bit
of 'follow' for instance: it doesn't care whether it grabs a
complete word or not - unless you tell it to care)

You can use wildcards as much as you like, so could
also try:

```
f..l
```

...which would grab 'feel', 'fowl', and many other
words, including the ones listed before.

If we wanted to be less fussy, then, we could rewrite
our regex for 'June or July' as follows:

```
Ju..
```

But remember that this would also grab parts of words,
such as the first four characters of 'Justice'. However, as
our scraper is also limited to matches that come before the
marker '2011', then that doesn't cause a problem here.

Matching zero, one or more characters - quantifiers

This wildcard is even more powerful when combined with
quantifiers. Quantifiers specify how many of a particular
character you want to match: one, one or more, zero or
more, or a particular number or range of numbers (such as
between 2 and 4).

The **plus sign**, **question mark and asterisk** are key
quantifiers if you want to be less fussy about the numbers
of a particular character, e.g.:

- `.?` - zero or one *of any character*
- `.*` - zero or **more** *of any character*
- `.+` - **one** or more *of any character*

Note the subtle differences: the first two will work even if *no character is there at all*. The plus sign only finds a match if the character is there.

This can also be combined with letters and ranges like so:

- `e?` - zero or one 'e'
- `e*` - zero or *more* 'e'
- `e+` - *one* or more 'e'
- `[0-9]+` - one or more numbers
- `[a-z]*` zero or more lower case characters
- `[A-Z]?` - zero or one upper case character
- `[a-z0-9A-Z]+` one or more lower case characters, numbers or upper case characters

Curly brackets after a character can be used to specify how many characters together you want to look for:

- `e{2}` - two 'e's, i.e. 'ee'
- `e{2,}` - *at least* two 'e's, which might be 'ee', 'eee', 'eeeeeeee' and so on.
- `e{2,4}` - between two and four 'e's: 'ee', 'eee' or 'eeee' but not 'eeeeee'

If you want your scraper to grab more than just the
month, then, you need to adapt the entry in that **Format**
column to grab all characters *after* the month too. Change
your regex in that column to this:

```
/Ju[nl][ey].*/
```

In the *marker after* column enter:

```
<
```

This should now match 'June' or 'July', *followed by zero
or more of any character* until it hits any ‹. Click **Execute**
to see if it works.

3 questions: What characters, how many, where?

The basic structure now merits an overview: regex consists
of the following:

- an identifier (the forward slashes that *identify* it as
  regex)
- a pattern (the characters we're looking for)
- and a modifier (how many, for example - our quan-
  tifiers)

Amit Arora puts it[3] this way: "To read (understand) or
write a regular expression pattern, you should ask the three
simple questions:

- "What characters to match ?

---

[3]http://www.digitalamit.com/article/regular_expression.phtml

- "How many characters to match ?
- "And where to match ?"

Using regex on an ugly page

Looking at the source HTML of our page on the Labour leader's meetings with donors and trade union general secretaries, we can see it's a pretty horrible piece of HTML which doesn't use list tags or even paragraphs. Instead, each new line is created with the `<br />` tag - a line break. Some use two, some use one, and because it's a common tag, it's used elsewhere on the page too. There are 82 of them in total.

Create a new scraper for this page with the *marker before* of:

```
>
```

...and the *marker after* of:

```
<br
```

And click **Execute**. This should give you four irrelevant lines followed by 42 containing the data you want. We could decide to stop here and clean this data up elsewhere - but I'm going to use this as a way to develop some knowledge of what regex can do.

What's the pattern?

Our data may not have much structure in its HTML - but it does have structure in the text itself. Let's take the longest of the lines:

```
<br />4 April 2011 &ndash;John Hannett (USDAW),
Michael Leahy (Community), Len McCluskey (UNITE),
Paul Kenny (GMB), Gerry Doherty (TSSA), Billy Hayes
(CWU)&ndash; Dinner, House of Commons<br />
```

This is what we have:

- A `<br />` tag.
- A number, which can be one or two digits.
- A space - even invisible characters are important, as we'll see.
- A month - twelve possibilities, aside from typos.
- Another space.
- Four digits: a year.
- A space.
- HTML code for a dash: `–` followed by:
- Names - normally in the format forename-space-surname. But we mustn't rule out the possibility of titles, middle names and extra surnames elsewhere in the data.
- Affiliations in brackets. These are not present for all entries.
- Commas separating multiple guests.
- HTML code for another dash: `–` followed by a venue.

Listing these yourself is a useful exercise as you can then write possible regex next to each item - or refer to them as you construct a regular expression.

In fact, based on what we know already we can start to construct the beginnings of some regex to extract the data. We already know that our *marker before* is this:

```
<br />
```

And we know that our data begins with either one number or two - the first part of a date.

We can express this in regex as follows:

```
/[0-9]+/
```

This, broken down, says:

- `/` - the identifier that says 'this is regex'. We don't need these in the *Format* column because anything in this is assumed to be regex - but it's a good habit to get into.
- `[0-9]` - any digit between 0 and 9. In other words, any number. This is our *pattern.*
- `+` - the plus sign *modifier* specifies not just any number but 'one or more'.
- `/` - the identifier that indicates the end of the regular expression

Save, and click **Execute** and look at the results. You should now have a column full of numbers - the first parts of our dates. Remember that the *Format* column specifies exactly what you want, and all we've specified so far is 'one or more digits' after the marker `<br />`

We can now continue to build our regular expression to look for the rest of the date. But to do that, we need to be able to identify invisible characters like spaces.

Matching non-textual characters

As well as simple characters and digits such as those identified above, regex can also identify spaces, punctuation, and line breaks.

How do you identify a character that doesn't exist? By combining certain characters with a backslash (*note that this is different to the more common forward slash used in web addresses and as an identifier above*). Here's a list from OutWit's own documentation on regex[4]:

\r line break (carriage return)

\n Unix line break (line feed)

\t tab

\f page break (form feed)

\s any space character (space, tab, return, line feed, form feed)

\S any non-space character (any character not matched by \s)

\w any word character (a-z, A-Z, 0-9, _, and certain 8-bit characters)

\W any non-word character (all characters not included by \w, incl. returns)

\d any digit (0-9)

\D any non-digit character (including carriage return)

\b any word boundary (position between a \w character and a W character)

\B any position that is not a word boundary

---

[4]http://www.outwit.com/support/help/hub/helpcenter/quickRegexp.xhtml

If some of this doesn't make sense, look it up. For example, what makes a Unix line break different to a standard line break? There's an answer on Stack Overflow[5] - in short: it's unlikely to be useful to you.

The one we need is right there: a space. This is indicated by the following regex:

```
\s
```

We can now add that to our regex to create this:

```
/[0-9]+\s/
```

Already you can see why regex looks like gobbledegook to people when they first come across it: it's easier written than read.

So breaking it down again, this says:

- `/` - the identifier that says 'this is regex'. We don't need these in the *Format* column because anything in this is assumed to be regex - but it's a good habit to get into.
- `[0-9]` - any digit between 0 and 9. In other words, any number. This is our *pattern.*
- `+` - the plus sign *modifier* specifies not just any number but 'one or more'.
- `\s` - followed by any space character
- `/` - the identifier that indicates the end of the regular expression

Now we should be able to pick up speed a bit. We have

---

[5]http://stackoverflow.com/questions/1279779/what-is-the-difference-between-r-and-n

the numbers followed by a space. After the space in our data we expect the name of a month - a word.

Rather than be too clever about this, we can start by simply saying 'any series of letters' and if we get dodgy results we can always tweak the regex.

You can say 'any lower case letter' with the regex [a-z] and 'any upper case letter' with [A-Z]. You can combine the two with [a-zA-Z].

And of course we can use a + modifier in the same way as we did with the numbers. Adding this to the end of our current regex creates the following:

```
/[0-9]+\s[a-zA-Z]+/
```

In your *Format* column add to your regex in the same way, and click **Execute**. Your scraper should now be grabbing both the number and the month, and thankfully every entry is as we hoped. So no tweaking is needed.

Cracking on, we now need to add another space before we specify an expression to represent a year. The space is, once again:

```
\s
```

And the year - well, we could say four digits, or one or more digits, or even the digits '20' followed by two more digits. As always, let's start simple and see what happens, and re-use the regex for 'one or more digits':

```
[0-9]+
```

Adding those two elements makes our regex look like this:

```
/[0-9]+\s[a-zA-Z]+\s[0-9]+/
```

Yep. Looks horrible. But once you've adapted your

regex to look like that, and clicked **Execute** once more, you'll see it's worked: we have the years; no need to tweak.

In fact, we now have the whole date, which feels like a good point to take a break and look at some other parts of regex.

What if my data contains full stops, forward slashes or other special characters?

If you're the inquisitive sort, you might have wondered what happens if you want to use regex to grab data that includes characters that regex uses for other purposes. For example, so far we've used a full stop to indicate 'any character', a question mark to indicate 'zero or one', forward slashes and backslashes, asterisks and plus signs, square brackets and and curly ones.

If you want to specify any of these *as characters* rather than as the special instructions regex normally uses them for, they need to be '**escaped**' - with a backslash, just as the invisible characters were above.

'Escaping' is basically telling the computer '*Don't treat this as you normally would*'. To confuse things, this works both ways. The character 's', on its own, normally means 'the letter s' in regex. Simple. But put a backslash before it and regex knows you mean 'any space character'.

The character '.' on its own, however, normally means 'any character'. But put a backslash before it and regex

knows you mean 'a full stop'.

The best way to remember the difference is this:

- If the backslash comes before a letter, assume it doesn't mean that letter
- If the backslash comes before a symbol, assume it means that symbol, literally

Escaping is common to many other tools and languages, so it's worth getting your head around.

This is how you escape a full stop:

\.

If this wasn't escaped, then it would be treated as a 'special character'. Specifically, if not escaped the full stop would be treated as meaning 'any character'. But when escaped, the full stop means 'look for a full stop here'.

There are plenty of other special characters to escape too, including, of course, the backslash itself:

\\

And the forward slash:

\/

The asterisk, full stop and question mark need to be escaped:

*

\+

\-

As do normal, curly and square brackets:

\(

\)

\{

```
\}
\[
\]
```

And the dollar sign and what's called the caret symbol:

```
\$
\^
```

A quicker way is to list them like so: .$*+-^(){}[]/

'Anything but that!' - negative matches

You may have noticed that some of the expressions above - such as \S and \D - are negative matches, such as *not* a space, or *not* a digit. You can specify other negative matches using the caret character - ^ - within square brackets (it has to be placed first), like so:

- [^a-z] - any character *not* lowercase a to z.
- [^A] - any character *other than* a capital A.

This or that - looking for more than one regular expression at the same time

The pipe symbol - | - can be used to separate two expressions that you want to match. For example, taking our regular expression to find the date:

```
/[0-9]+\s[a-zA-Z]+\s[0-9]+/
```

We can add a pipe symbol at the end of that (just before the last forward slash):

`/[0-9]+\s[a-zA-Z]+\s[0-9]+|/`

And then write a second regular expression in case we were looking for dates written a different way, e.g. '2nd July' instead of '2 July':

`/[0-9]+\s[a-zA-Z]+\s[0-9]+|[0-9]+[nsrt][drh]\s[a-zA-Z]+\s[0-9`

(In the second regex we still start with one or more numbers, but these would be followed by 'st' (e.g. '1st', '21st'), 'nd', 'rd' or 'th'. The two square brackets specify the possible first and second letters of those.)

Try that regex in your scraper - you'll see it still picks up the dates with the first part, even though none match the second part.

Only *here* - specifying location

You can also use the caret symbol *outside of square brackets* to specify that you are looking for something *at the start of a string*.

To specify that you're looking at the *end of a string*, you can use the $ sign.

Here are some examples of both in action:

`^[aeiou].*` - match any string beginning (ˆ) with a vowel [aeiou], and all text after (.*).

`.*[0-9]$` - match any string ending ($) with a number [0-9], and all text before (.*).

With all that covered, we can return to our scraper.

Back to the scraper: grabbing the rest of the data

Where were we? Oh yes, we'd written some regex in the first line of our scraper to grab the date, based on a *marker before* of:

>

And a *Format* of:

`/[0-9]+\s[a-zA-Z]+\s[0-9]+/`

Broken down, that regex means:

`/` - regex follows...

`[0-9]` - we want to match: a digit from 0 to 9

`+` - (one or more)

`\s` - followed by a space

`[a-zA-Z]+` - followed by a lowercase or upppercase letter (one or more)

`\s` - followed by a space

`[0-9]+` - followed by a digit from 0 to 9 (one or more)

`/` - regex ends.

Now we can create a second line in our scraper to grab the names.

So, in that second line:

1. Give it a *Description* of 'Names'.
2. We can use regex in the *marker before* and *marker after* columns too - so the simple option here is to copy the regex we created in the line before and use it as our *Marker before*: `/[0-9]+\s[a-zA-Z]+\s[0-9]+/`
3. In *Marker after* put: ‹

4. Click **Execute**.

Because we're not specifying the *format* in this line, we don't need to be too specific about what we're grabbing. All we're specifying is that comes before and after. OutWit allows a number of combinations of these:

- A *marker before* and a *marker after*
- A *marker before* and a *format*
- A *marker after* and a *format*
- A *marker before*, a *marker after*, and a *format*
- A *marker before*
- A *format*

Having clicked **Execute** you should see two columns: Date, and Names. The names have dashes before them, which is probably best cleaned in a spreadsheet tool. But if you did want to remove them, you could change your regex to the following:

```
/[0-9]+\s[a-zA-Z]+\s[0-9]+\s(&ndash;|-)/
```

After an extra space - \s - the extra regex here uses what's called a **sub-pattern**. A sub-pattern is a piece of regex *within* your regex within brackets:

```
(&ndash;|-)
```

This says match *either* – *or* - with the *either* part specified by the 'pipe' symbol between the two possibilities:

```
|
```

We've used this because although most lines use the code &ndash, a couple just use a dash (-) so we need to be able to match on either.

Which dash? Negative matches in practice.

There's a final bit of data we need to grab: in addition to the date of each meeting and the names of the people there, some entries have a venue: 'Dinner at the House of Commons'.

Looking closer, we see that the venue is always preceded by a dash. So we need an extra line to grab that.

For *Description* put 'Venue'.

As for the *marker before*: look in the HTML for one of the venues such as 'Dinner at Ed Miliband's House' and you'll see – - the code for an 'en dash'. Put this as your *marker before*:

```
&ndash;
```

And put an opening HTML chevron as your *marker after*:

```
<
```

Click **Execute** and you'll see mixed results: it's grabbing a bunch of names as well as the venue, because – is also used before names.

We need to work out what's different about the two dashes. One difference is that the first dash always comes after a date: specifically, the year. So we can specify: a dash that *does not* come after a number. In *marker before* then, type the following regex:

```
/\D\s&ndash;/
```

This breaks down like so:

/ - this is regex

\D - any character *other than* a digit

\s - followed by a space

– - followed by – in the HTML code

/ - regex ends

For *marker after* put:

<

And click **Execute**

This has now worked for all but one of the entries
that mention venues (April 4 2011: Dinner at the House of
Commons). Why?

Try reading out the regex (or writing it down) liter-
ally as you go through the HTML code that's not being
matched:

- Not a digit? Yes, that's right.
- A space? No.

That's why the regex doesn't create a match: there is
no space before – in the code before 'Dinner at the
House of Commons', there is a closing bracket:)

We need to adapt our regex to be a little less fussy.
So, instead of \s for 'any space character', use . for 'any
character', which should make your regular expression
look like this:

/\D.–/

Click **Execute** and the scraper should now work.

For more on regex look at Amit Arora's introduction
to regular expressions[6] - one of clearest you will find

---

[6]http://www.digitalamit.com/article/regular_expression.phtml

online. And The Bastards Book of Ruby has a good guide
to Regex[7] too - although bear in mind that there may be
slight differences in the way it is used with other languages.
More broadly, Regular-Expressions.info[8] is one of the most
comprehensive resources out there.

Recap

Regex is particularly intimidating to approach, and this
chapter has represented a steep learning curve in trying
to take it all in. As with so much code, you shouldn't
expect to learn it all by heart, but rather know what is
broadly possible, where to go to remind yourself how to
do it, and what techniques to use in thinking creatively
around phrasing your regular expression. So here are the
key points:

- **Regular expressions** allow you to describe a **pat-
  tern** of characters - including invisible ones - in order
  to find a match.
- You can be more or less **strict** in what you ask for:
  you can specify a particular character, or a range.
  You can specify one, or at least one, or a particular
  number, or a minimum and maximum, or from zero
  to many. You can specify *anything but* a particular
  range or type of character.

---

[7]http://ruby.bastardsbook.com/chapters/regexes/
[8]http://www.regular-expressions.info/

- On paper, **break up the code you're trying to scrape to identify those patterns**. Then build your regex up part by part to match each part: a date, a space, a month, and so on.
- If you get more than one match on part of the pattern, then **use negative matches** (e.g. not a digit or not a space, etc.) to rule out the other match.
- Regex consists of **an identifier**; **a pattern**; **and a modifier**. The first identifies it as regex, the second identifies what you're looking for, and the last says how fussy you are about it.
- You can use brackets to specify **sub-patterns** - regex within regex, such as a range of possible matches separated by the pipe symbol.
- Finally, remember that **trial and error** is especially useful when using regex. Your skill is in solving the problems that many attempts will inevitably generate - not in getting it right first time.

Tests

- If you have the Pro version of OutWit, for those entries containing quite a few names see if you can separate them using the **Separator** column in the scraper (each name is separated by a comma: ', '). Note that in the data some dates are repeated for multiple meetings, suggesting that each meeting is treated separately, rather than each date, so the entries with multiple names are still a single meeting.

- Try to extract the names of the organisations that each person belongs to.
- Imagine you had a large dataset but were only interested in results containing a particular organisation. How would you use regex to do that? Try it with UNITE in this data.
- Go back to the consortia scraper from the previous chapter and see if you can use regex to solve some of the problems there. For instance, you could use the following code in the *format* column to solve your `</strong><br />` problem: `/.*<\/strong><br \/>|.*<\/strong>\)<br \/>|.*\sAlliance<br/`(*note how the forward slashes are escaped and how* `.*` *is used to grab all characters between your tags*).

11 Scrapers #10 and #11: Scraping hidden and 'invisible' data on a webpage: icons and 'reveals'

Sometimes the data that a webpage contains won't be immediately obvious to you: either because it's *represented* in icons rather than words, or because it's literally hidden until you click on something.

A good example of each is provided by the Office of the Independent Adjudicator's page of Older Case Studies, at http://www.oiahe.org.uk/decisions-and-publications/recent-decisions-of-the-oia/older-case-studies.aspx[1] - and Inclu-

---

[1] http://www.oiahe.org.uk/decisions-and-publications/recent-decisions-of-the-oia/older-case-studies.aspx

sive London's guide to accessibility at Olympic venues, at
http://www.inclusivelondon.com/search/what/0/37523/921/0/London/Olym

The Office of the Independent Adjudicator seems to
only list three pieces of information on each old case: its
number, the issues concerned, and the outcome. But if you
click on any case, you will see it expand to reveal more
detailed data: A text summary of a few paragraphs; some
reasons for the case; and the recommendations.

The URL doesn't change - and if you view the source
HTML for the page you'll see that all this information is
actually there. It's just not revealed unless you click on a
particular case. As a result, it can all be scraped.

The Inclusive London data is subtler. Each venue has
a series of badges indicating whether a venue has partic-
ular facilities: 'Assistance dogs welcome', for example, or
accessible toilets.

But how do you scrape the data that these images
represent?

The key, again, is to look at the page in terms of its
HTML. To display an image, a webpage has to load it using
HTML. Each different image will have a different filename.
If any image is being used repeatedly to indicate the same
thing: that's structure. And data.

Confused? Let's scrape it to demonstrate.

---

[2]http://www.inclusivelondon.com/search/what/0/37523/921/0/London/
Olympic%20And%20Paralympic%20Venues/0/0/0/0/0/search.aspx?tbclr=1

Scraping accessibility data on Olympic venues

So our story is how accessible the venues for the Olympics
are. Who are the heroes and villains, and so on (bearing in
mind that reality may not reflect the badges).

Open Inclusive London's guide to accessibility at Olympic
venues[3] in Firefox, and launch OutWit Hub.

Under *automators* on the left, click on **scrapers** and
click **New**. Start by grabbing some simple data, such as the
name of each venue (*marker before* of ‹h2›) and address
(*marker before* of ‹div class="general-content"› and
marker after of ‹/div›).

Now we know that each venue has a series of images
representing different aspects of accessibility - but how are
they described in the HTML? It turns out to be a list like so:

```
‹ul›
‹li id="ctl01_MainBodyContentPlaceHolder_CompanyBasicInfoCont
SearchResultsRepeater_ctl00_DisabledIconListControl_-
List_GDOG"›‹img id="ctl01_MainBodyContentPlaceHolder_-
CompanyBasicInfoControl_SearchResultsRepeater_ctl00_-
DisabledIconListControl_GDOG" title="Assistance dogs
welcome" src="/images/site/portals/inclusivelondon/default/search
off.png" alt="Assistance dogs welcome" style="border-width:0px;"
/›‹/li›
```

There are a number of ways we can grab the details
from this:

- We can grab the whole `<ul>` list, and use the *separator* column to split it by `<li>`
- We can grab the `title="` attributes of each image, or the `alt="` attributes
- We can grab the codes (e.g. GDOG) from the end of one of the `id="` attributes, which seem to have a consistent naming pattern
- We can grab the name of the image

As the `alt="` attribute seems the most literal, we'll try that first.

Create a new line, then, called *Accessibility* and in *marker before* type:

```
alt="
```

In *marker after* type:

```
"
```

And click **Execute**.

You should now have a whole bunch of columns containing accessibility information. But their order is inconsistent: 'Assistance dogs welcome', for example, appears in different columns for different venues. That can be sorted out in cleaning, but we can also sort it out here.

Untick the box at the start of the line you just created, and start a new one with the *description* 'Assistance dogs'.

Instead of *marker before*, this time we'll just use **Format**, as all we're looking for is an exact match: does the image exist, or not.

The *format*, then, is this:

```
alt="Assistance dogs
```

If you click **Execute** now you'll get a column full of `alt="Assistance dogs` where there's a series of matches.

You can count these in spreadsheet software if you need to (using the function COUNTIF) but if you've got the Pro version of OutWit Hub, you can tidy things up a bit, by using the **Replace** column.

The *Replace* column specifies what you want to replace your scraped text with, or add to it.

In this case, if you type `YES` then that will appear instead of `alt="Assistance dogs` where there's a match. You could also choose '1' so you can total up the results.

Once you find out the alt tags for all of the other icons (simply hover over one with your cursor and it should appear - the best place to do this is the set of icons across the search area where it says *Filter your results by their accessible features*), you can repeat this process for the other alt tags.

? Is scraping legal?

You need to make a distinction in scraping between the *acquiring* of the data and any *reproduction* or publication of it. *Publication* of data which you have scraped (in its raw form) may break copyright law or database rights, both of which protect the creators of databases.

A useful analogy is the football fixture list: republishing football fixtures for a whole year requires (in many countries) the permission of the sporting body that holds those rights. But saying what match is happening this weekend would not.

So, if your intention is to scrape the data in order to establish a fact, a pattern or a lead, and that is what you are *publishing*, then you are not breaking copyright law.

What you may be breaking, however, are the *terms and conditions* of the site. Breaking these may mean you are banned from the site, so read them carefully.

Hidden HTML

Now to scrape the cases heard by the Office of the Independent Adjudicator: http://www.oiahe.org.uk/decisions-and-

publications/recent-decisions-of-the-oia/older-case-studies.aspx[4]

The process here is a little simpler: again it means looking at the source HTML and finding the elements you want.

The first two are easy to find and work first time. Here is the *marker before* and *marker after* for each one:

- `<br />Issues:` before and `<` after
- `<br />Outcome:` before and `</` after

Some mention a "Course/Professional Body" - this is a little tricky to identify because the formatting is less consistent (in some cases there's a colon immediately after; in others there's a HTML tag between the last word and that colon). The best way to drill down is to:

- Search for part of the phrase in the HTML and see how many results there are.
- Compare how many results you have in the scraper.
- Simplify your *marker before* as much as possible and if it grabs too many results, progressively add to it until it works.

This is the code I ended up with that worked:

- `Course/Professional Body` before and `</p` after

---

[4]http://www.oiahe.org.uk/decisions-and-publications/recent-decisions-of-the-oia/older-case-studies.aspx

The next part to grab is the summary. Here again there
is inconsistent HTML, as well as the problem of summaries
having varying numbers of paragraphs. As we explored
in chapter 8, a useful technique here is to step back. The
marker before that works best isn't a HTML tag (because
we have multiple paragraph tags) but rather the title of the
next section: 'Reasons'.

Here is the code I used:

- `Summary` before and `Reasons` after.

This still leaves two entries empty: case studies 46 and
3.

Case Study 46 has an obvious problem: they've given
the summary section the header of 'Reasons' and vice
versa. Case Study 3 has no 'Reasons' section at all.

We could put together some regex to account for Case
Study 46 but this is one of those cases where just making
a note to add that record manually will be much quicker.
Case Study 3 is not a problem, because there is genuinely
no data.

You'll find similar problems with the next line: 'Reasons'.

Here the *marker before* might reasonably be first tested
with:

`Reasons:`

And the *marker after* as:

`Recommendations`

The first problem is that the term 'Recommendations'
isn't used consistently. You could try replacing it with

`<strong>` which comes before it - but there's at least one instance where this is used earlier. The tag which comes before *that* and eventually seems to work best is:

`</div>`

Still then we have one missing entry which, on closer inspection, turns out to place a tag between 'Reasons' and the colon ':', so a slight simplication of the *marker before* is needed - to:

`Reasons`

This works for all entries on that page with data apart from Case Study 30, where there is no `</div>` at all.

Once again, we can add this manually quicker than trying to solve the problem, so make a note and move on.

The next line to scrape is 'Recommendations' itself. An obvious *marker before* is exactly that:

`Recommendations`

Recommendations seem to come as a list, so a first guess might be to use the closing tag for a bullet list:

`</ul>`

Clicking **Execute** shows this works for many lines, but not all. The first to be missed is Case Study 58: in the HTML for that we can see that 'Recommendations' is not used at all, but 'Suggestion'.

Once again, the quickest solution (assuming that time is always going to be an issue in journalism) may be not to try to catch them all in the same line, but rather to create a new line to scrape those cases using 'Suggestion', and then merge the two at the cleaning stage.

This still leaves many lines with neither recommen-

dations or suggestions, but an inspection of these finds that this is because those cases often reached a decision of 'Not justified' or similar, and so no recommendations were made.

Finally, some entries have 'observations'. These need to be grabbed with a further line with a *marker before* of:

```
Observations
```

And a *marker after* of:

```
</div>
```

You can create further lines to look for other details. For example, if you wanted to bring out any mention of 'fees' you could create a new line with the following **regex** in the **Format** column (*See previous chapter for more on regex*):

```
/.*fees.*/
```

This, quite simply, says:

- `/` - start of regex
- `.*` - match zero or more of any character...
- `fees` - ...occurring before the series of characters 'fees' (this doesn't have to be a separate word - 'toffees' would create a match here - although it's unlikely to make an appearance in this context).
- `.*` - if zero or more of any character appears after 'fees', match them too.
- `/` - end of regex

About a dozen cases contain a match against this regular expression - some of them more than one (which generates extra columns).

You can adapt that regex to look for any other similar keyword.

Once you've got the results you want (tick *Clean text* if you want HTML stripped out), click **Export** and save as a CSV file to clean and analyse in a spreadsheet tool.

Recap

This chapter has largely been about exercising some of the techniques used in the previous two, introducing some extra problem-solving approaches, and the concept of scraping invisible or hidden data. The key points to take away:

- **Data doesn't always appear in obvious tables or lists**. It may be represented by icons, or only revealed when you click somewhere on a page.
- If you're scraping data represented by icons, look for their alt tags or image URLs.
- **Look at the HTML code for clues**. What you see in a browser is only one perspective on that page (equally a page may be generated by cookies or other technologies which mean the data is not visible in the HTML itself - we'll deal with this situation in later chapters relating to scraping *forms and databases*).
- Use the **Replace** option to change the source data into a representation of it that is easier to work with, e.g. 'YES' or '1'.
- If a part of a scraper only misses one part of your data, **don't feel you have to spend hours making**

it work perfectly, when it will be quicker to add the missing line manually later - but don't forget to.

- Single characters can make all the different to a scraper picking something up or not. **Try leaving out punctuation such as colons**, **commas**, **dashes**, **plurals**, etc. but check that you don't get extra results you don't want.
- If different terms are used for the same thing, **try creating two lines that scrape each and then merging the two sets of results in a spreadsheet**.
- **Use regex to find mentions of key terms and the text surrounding them**.

Tests

- Write more lines to find mentions of key terms in the case studies
- Find out how to merge the two columns 'Recommendations' and 'Suggestions' in a spreadsheet package (tip: create a new column and use a formua like =B2&C2 to grab the contents of the other two)
- Download the results and add the missing lines manually. How would you make sure you don't forget to do this?
- Think about information in a field of interest which might be represented by icons (such as star ratings, thumbs, grade, facilities etc.). See if you can scrape that.

12 Scraper #12: Scraperwiki intro: adapting a Twitter and Google News scraper

A version of this chapter was originally published when both Scraperwiki and Twitter were different. The code no longer works, so you will not be able to adapt the code yourself and see working results. However, the lessons of this chapter are still useful.

Now we're back where we started with importHTML - we're going to adapt a scraper without knowing exactly how it works.

The great thing about Scraperwiki *was* that you could search and adapt other users' scrapers. So if, for example, you wanted to scrape Twitter during a big newsworthy event such as a riot, you might be able to find an existing scraper to adapt.

Scraperwiki's changes

In July 2013 Scraperwiki launched a new version of the site, with old scrapers and documentation hosted at the `classic.scraperwiki` domain. The new site makes it easier to write scrapers for websites like Twitter and Last.fm - with quick start guides hosted at https://scraperwiki.com/help[1]. However, scrapers are private by default on the new site, so it may be harder to find others.

If you already have scrapers on the old site there is a way to move them all over detailed here[2].

You can still browse old scrapers using the search box on Scraperwiki Classic[3]'s home page (Scraperwiki Classic is the version where you could see each other's code - the new Scraperwiki may yet introduce changes to allow this again). The results page will have a URL like this:

---

[1]https://scraperwiki.com/help

[2]http://rdjcode.wordpress.com/2013/07/03/moving-from-scraperwiki-classic-to-scraperwiki-awesome/

[3]https://classic.scraperwiki.com/

https://classic.scraperwiki.com/search/twitter/[4] - at the top of the page you will find results from users, but below that you will find scrapers, including the following:

- Example Twitter hashtag user friendship network[5]
- Basic Twitter Scraper[6]
- Twitter Public Mood Scraper[7]
- Twitter hashtag search[8]
- Candidate Twitter Test Scraper[9]

If you click through to one of these, look to see whether the scraper worked (if not there will be no data underneath in *This scraper's datastore", and sometimes an error code in red).

Also check the 'This scraper in context' area (see image below), which will tell you if the scraper has been **cloned** from another. If it has, click on that other scraper to find the original: even if the scraper you're looking at doesn't work, it may have been cloned from one that does.

This view in context

Based on: Tony Hirst / Example Twitter hashtag user friendship network

The this view in context section

---

[4]https://classic.scraperwiki.com/search/twitter/

[5]https://scraperwiki.com/views/example_twitter_hashtag_user_friendship_network_1/

[6]https://scraperwiki.com/scrapers/basic_twitter_scraper_380/

[7]https://scraperwiki.com/scrapers/twitter_public_mood_scraper/

[8]https://scraperwiki.com/scrapers/twitter_hashtag_search/

[9]https://scraperwiki.com/scrapers/candidate_twitter_test_scraper/

Why old Twitter scrapers no longer work

The Twitter scrapers above were written when Twitter was set up differently. They no longer work - in the jargon of programming they are "broken". This is because when Twitter first launched the company made it very easy for developers to create interesting applications with their data. But over the years they changed things so that their data was less and less open, and scraping it became harder and harder.

Thankfully, the *new* Scraperwiki makes it easier than ever to scrape tweets on Twitter, or the followers of a particular user - as long as you have a Twitter account. Just click on the buttons to choose that scraper and enter your criteria in the box. No coding required!

Just click on the buttons to choose that scraper

The Twitter scraper - enter your criteria in the box. No coding required!

Forking a scraper

Now I want to explain how to clone (what's called **forking**) a scraper. In this case, the one called 'Basic Twitter Scraper'. If we look at the '**This scraper in context**' area we can see

that it's based on a previous scraper (the URL is a clue -
there's a number added to 'basic_twitter_scraper') - and in
fact, it's likely you'll follow that trail back through several
different scrapers.

I'm going to work on the last one with that name in the
trail I followed: https://classic.scraperwiki.com/editor/raw/basic_-
twitter_scraper[10] (notice there's no number on the URL
too).

Click on **View Source** and you'll see code like this:

```
###############################################################
# Twitter API scraper - designed to be forked
and used for more interesting things
###############################################################
import scraperwiki
import simplejson
import urllib2
# Get results from the Twitter API! Change QUERY
to your search term of choice.
# Examples: 'newsnight', '#newsnight', 'from:bbcnewsnight',
'to:bbcnewsnight'
QUERY = 'wiki'
RESULTS_PER_PAGE = '100'
LANGUAGE = 'en'
NUM_PAGES = 1
for page in range(1, NUM_PAGES+1):
 base_url = 'http://search.twitter.com/search.json?q=%s&rpp=%s
\
```

---

[10]https://classic.scraperwiki.com/editor/raw/basic_twitter_scraper

```
    % (urllib2.quote(QUERY), RESULTS_PER_PAGE, LANGUAGE,
page)
    try:
    results_json = simplejson.loads(scraperwiki.scrape(base_-
url))
    for result in results_json['results']:
    data = {}
    data['id'] = result['id']
    data['text'] = result['text']
    data['from_user'] = result['from_user']
    print data['from_user'], data['text']
    scraperwiki.sqlite.save(["id"], data)
    except:
    print 'Oh dear, failed to scrape %s' % base_url
```

The first thing you need to look at in any scraper is the
comments. These are indicated by hash signs (#) and are
coloured grey in Scraperwiki's view.

You can see from these that this scraper is actually
"designed to be forked". You can also see instructions on
how to adapt it (change the word after QUERY = ' to
what you want to search for - make sure you **include the
quotation marks** to indicate that you are searching for a
string, e.g. QUERY = '#ddj').

To clone this code you'll need to set up an account with,
and be logged in to, Scraperwiki first:

1. In the **View source** view for that scraper[11], copy all
   the text.

_____

[11]https://classic.scraperwiki.com/editor/raw/basic_twitter_scraper

2. Create a new scraper in Scraperwiki (once logged in, click **Create a new dataset**), and then the '**Code in your browser**' button ("*Just like you used to in ScraperWiki Classic*") from the options presented. Choose the Python language when prompted. Then paste the copied text below the line `#!/usr/bin/env python`

3. Change the parts of the code you want to adapt to your own purposes (in this case, change the part `QUERY = 'wiki'` to a search string of your choice such as `QUERY = 'corruption'` - keep the inverted commas).

4. The scraper will automatically be saved, but you can change its name from *Untitled scraper* by clicking the gear icon to the right of the name and selecting **Rename**.

As explained earlier, this scraper won't actually find any information because Twitter no longer provides information in the format this is trying to scrape. If you click the **RUN** button along the bottom left to test your scraper it will actually run without generating errors - but it will also not print any results in the **Console** view underneath.

The reason it doesn't generate any errors is because the scraper is well written enough not to stop when it has an error. The key lines are these:

```
try:
results_json = simplejson.loads(scraperwiki.scrape(base_-url))
```

and these:

```
except:
    print 'Oh dear, failed to scrape %s' % base_url
```

The first lines say `try:` something. But if that something doesn't work, then it runs the `except:` lines instead. So even if it doesn't work, it doesn't generate an error.

Scraping Google News

Although that's a useful lesson, it's not very satisfying. So, here's another scraper you can adapt to your own needs by changing a single word.

A search on Scraperwiki Classic for 'google news python'[12] brings up this scraper by Ross Jones[13], which you can see in full by clicking on **View Source**[14].

Copy the code and once again paste it into a new scraper on the new Scraperwiki. Again, make sure you leave in the line `#!/usr/bin/env python` which says what language the scraper is in.

Click **Run** underneath the window containing the code. You should see a `Started run:` line with the time and date you ran it, then below that a line like this:

```
[<DOM Element: link at 0x7f9c64a3d9e0>, <DOM
Element: ...
```

And then below that two numbered lines with some headlines relating to Apple, before a `Finished run:` line with the time and date the scraper finished.

---

[12]https://www.google.com/search?q=site%3Aclassic.scraperwiki.com+google+news+python

[13]https://classic.scraperwiki.com/scrapers/google_news_scrape/

[14]https://classic.scraperwiki.com/editor/raw/google_news_scrape

Hopefully the scraper should have worked - and you can work out what it does: scrape two headlines, with accompanying URLs, from Google News about Apple.

Knowing that, you can try changing one thing about the scraper to see if it will scrape news about something else, or more headlines.

Before you read on, take a look at that code and try to guess which line determines that the scraper will look for 'apple'. You may want to circle a few possibilities, and then try each one in turn.

Here it is in full to give you some time - again, remember you don't need to *understand* the code; at this stage you just need to be willing to try to *adapt* one word in it (and to fail):

```python
#!/usr/bin/env python
import urllib2
from xml.dom.minidom import parseString
def get_google_new_results( term, count ):

    results = []

    obj = parseString( urllib2.urlopen('http://news.google.com/
   % term).read() )

    elements = obj.getElementsByTagName('title')[2:]
   # To get rid of unwanted title elements
   in XML doc

    links = obj.getElementsByTagName('link')[2:]

    print links
```

```
    for element in elements[:count]:

        headline = element.childNodes[0].data

        for link in links:

            url = link.childNodes[0].data.split('=')[-1]

        newssearch = headline + ' ->
        ' + url

        results.append( newssearch )

    return results

items = get_google_new_results( 'apple', 2 )
for i,e in enumerate(items):

    print '%d: %s' % (i+1,e,)
```

Done it?

OK, here's the line:

```
items = get_google_new_results( 'apple', 2 )
```

If you change `'apple'` to another word (only one word) then you will get results for that. And if you change 2 to another number, you will get that many results (up to a maximum of 10).

> ⓘ If you want to use a two word search, make sure you put %20 or + between them. A search for 10 results for Bayern Munich, for example, would look like this:
>
> ```
> items    =    get_google_new_results(
> 'bayern+munich', 10 )
> ```

If you get an error message, check that you've included quotation marks (single or double? Try both and see!) and not changed anything else about the original working scraper.

Recap

This has been an intentionally simple introduction to Scraperwiki - all you've had to do is change one word. But there's a lot to be learned:

- You can **use the search facility to find scrapers that perform a similar function** to that which you need. This process isn't limited to Scraperwiki - you can find scrapers all around the web.
- Look at the '**This view in context**' area to trace the origins of any scraper on Classic Scraperwiki, and other versions of it.
- **Look for the comments** in code to give you clues to what each part of it does - in Python these are indicated by hashes (#).
- **Forking** is the process of copying (cloning) another scraper to adapt them to your own purposes.
- **Adapt** other *working* scrapers to your purposes if you can - it's quicker, and a good way to learn. **Look for clues** in the comments, and **bits that you recognise** (such as URLs).
- **Strings** of characters should be placed in quotation marks - we covered this with the first scraper, but it's reared its head again here.

Tests

- Try out the simple Twitter scrapers on Scraperwiki: scraping by keyword, or scraping a user's followers.
- Search for other scrapers in Scraperwiki Classic, or browse through by tag, and read the comments on any scraper. Does it help explain the code that follows?
- See if the scraper is based on others, and click back to those.
- Fork one of the simpler scrapers that you find, and try to change one small piece of the code, such as a string. Does it still work?

13 Scraper #13: Tracing the code - libraries and functions, and documentation in Scraperwiki

Now we've adapted a scraper, we're going to look at the **documentation** on Scraperwiki for some help.

As in the last chapter, create a new scraper in Scraperwiki by logging in, clicking **Create a new dataset**, and then the '**Code in your browser**' button ("*Just like you used to in ScraperWiki Classic*") from the options presented. Choose the **Python** language when prompted.

You can access Scraperwiki's documentation and tutorials from within any scraper using the 'Help' button in the bottom right. The first option - "**Quick start guide**" is a good one to start with, and the second option "Scraping tutorials" provides more.

The tutorials link in a Scraperwiki code view

However, we're going to use one of the Scraperwiki
Classic tutorials, because it allows us to explore some
troubleshooting along the way.

Here's the first tutorial](https://classic.scraperwiki.com/scrapers/tutoria
1):

```
#########################
# START HERE: Tutorial 1: Getting used to the
ScraperWiki editing interface.
# Follow the actions listed with -- BLOCK
CAPITALS below.
#########################
---------------------------------
# 1. Start by running a really simple Python
script, just to make sure that
# everything is working OK.
# -- CLICK THE 'RUN' BUTTON BELOW
# You should see some numbers print in the
'Console' tab below. If it doesn't work,
```

```
   # try reopening this page in a different browser
- Chrome or the latest Firefox.
   --------------------------------
   for i in range(10):
    print "Hello", i
   --------------------------------
   # 2. Next, try scraping an actual web page and
getting some raw HTML.
   # -- UNCOMMENT THE THREE LINES BELOW (i.e. delete
the # at the start of the lines)
   # -- CLICK THE 'RUN' BUTTON AGAIN
   # You should see the raw HTML at the bottom of
the 'Console' tab.
   # Click on the 'more' link to see it all, and
the 'Sources' tab to see our URL
   # you can click on the URL to see the original
page.
   --------------------------------
   #import scraperwiki
   #html = scraperwiki.scrape('https://scraperwiki.com/hello_-
world.html')
   #print html
   --------------------------------
   # In the next tutorial, you'll learn how to
extract the useful parts
   # from the raw HTML page.
   --------------------------------
```

Searching for Scraperwiki documentation

If you search for 'Scraperwiki documentation' on Google you're likely to find the documentation pages from Scraperwiki Classic[1]), which includes a link to *Live tutorials* (https://scraperwiki.com/docs/python/tutorials/[2]). The links no longer work, and will take you to instead back to Scraperwiki.com.

However, you can find other people's scrapers *based* on the tutorial templates by doing the following:

1. Hover over one of the tutorial links with your mouse, then right-click and select **Copy link address** (or the equivalent for your browser).
2. Paste into your address bar but *do not press Enter*. You're going to edit this URL to change it from a template for a new scraper (which you can no longer create) to an existing scraper.
3. Delete the `new/python?template=` part from the URL. For example, the first tutorial link should be changed from `https://classic.scraperwiki.com/scrapers/new/python?template=tutorial-1` to `https://classic.scraperwiki.com/scrapers/tutorial-1`. You should now be able to see the scraper description page, and click '**View source**' to see the original code.

---

[1] https://scraperwiki.com/docs/

The tutorials on Scraperwiki are very useful for getting simple scripts working quickly - but they're not the most accessible if you don't have a programming background. You can follow the instructions and get the script working, but why and how is not always clear.

So, here is the part of that same scraper once you've followed the instructions and **uncommented** three lines of code. (*Comments* in code - intended to explain the working code that followed - are normally distinguished from *working* code by a symbol of some sort. 'Uncommenting' means removing that symbol (a hash sign in Python) so that anything after it becomes active code, rather than inactive 'comment'.)

These are the first lines that the tutorial scraper says need to be uncommented. We're going to relate this to some of the principles you've already learned:

```
import scraperwiki
html = scraperwiki.scrape('https://scraperwiki.com/hello_-world.html')
print html
```

This code does 3 things:

The first line (`import scraperwiki`) brings in (***import***s) a collection of **functions**. A *collection* of functions is called a **library** (think of a collection of books that help you do different things).

We encountered a function in our very first scraper: importHTML (this works in Google spreadsheets, but not in Python scrapers, by the way). A function, you'll remember,

---

[2]https://scraperwiki.com/docs/python/tutorials/

does something useful.

Importing the `scraperwiki` library (not to be confused with the website of the same name) means we can use a number of functions in that library for scraping text. In fact, we're going to use one in the next line.

The name of that function is `scrape` - and you can tell it belongs to the scraperwiki library because it's named after it with a full stop (period) connecting them: `scraperwiki.scrape`

Parent/child relationships

Full stops (periods) often indicate a relationship: in this case, a parent-and-child relationship. So `scraperwiki` contains, or is the parent of, `scrape`.

They're also sometimes used in scraping when you want to dig down to find the child of a particular parent object - just as we used slashes in the importXML scraper to look for `councils/council` or `//div//@href`. (in a scraper which used this formatting the equivalent might look something like `councils.council` or `div.@href`).

Parameters (again)

Now we know from the first scraper in this book that a function is followed by parentheses, which contain **parameters**: the ingredients needed for a function to work, so `scraperwiki.scrape('https://scraperwiki.com/hello_-world.html')` is performing the function `scrape` with that parameter: i.e. the URL.

In that first scraper, however, the results of using that
function on that URL would be shown on the Google Docs
spreadsheet. In *this* chapter's scraper, we need to **store
those results** first. That's what the first part of that line
does:

```
html =
```

This is something new: a **variable**. Variables are central
to programming, so it's worth taking a detour to explain
what they are and how they work.

Detour: Variables

Variables are names for things. You choose the name - you
could replace `html` above with anything you like: `beans`,
`custard`, `elephant_poo` - whatever. It will still work (as
long as you don't pick a name that has another meaning
in the script or Python - such as `print` or `import`.

Try it, and see what happens. If it breaks, you can
always change it back. You will also notice that words that
have a pre-defined meaning such as those will be colour-
coded red in Scraperwiki.

The 'thing' that the variable refers to can be anything:
it can be:

- A number, such as 10 (whole numbers are called
  integers; those with decimal points, e.g. 10.1, are
  called **floating points**, or **floats** for short);
- A **string** of characters, such as `'Paul Bradshaw'`
  or `"Scraping for Journalists"` (note the inverted

> commas - you can use either single or double *straight* quotes but be careful to type them in directly and *not* to copy and paste them from word processing software which sometimes formats them as *curly* quotes);
>
> - A **list**, such as `[5, 10, 20, 100]` or `['Paul', 'Jane', 'Brian', 'Roy']` (note that each string is in inverted commas, whereas each number is not);
> - A **dictionary** (a list of pairs - think of them as words and definitions, like 'adjective' : 'describing word', 'noun' : 'object', 'verb' : 'action');
> - And it can be other things, even other variables.

You assign a property to a variable by **initialising** it with the = operator like so:

```
Name = 'Paul'
Age = 17
```

You can also change the value of a variable like so:

```
Age = Age+1
Name = 'Laura'
```

Now Age is 18, and Name has changed from Paul to Laura. What a busy year!

To see this in action, try adding the following line to your code under the line initialising the html variable, and run it:

```
sausages = html
print sausages
```

The first line puts the variable html into a new variable sausages (if we just wanted to put the *string* 'html' into the

variable `sausages` then we'd put quotation marks around
it).

   The second line prints it. Because `sausages` is the same
as `html` the result is the same.

Scraping tip #2: read code from right to left

This second line provides the first opportunity to
demonstrate something counter-intuitive (at first)
about code: it makes more sense when you read it
from right to left.

Here's that line of code:

```
html = scraperwiki.scrape('https://scraperwiki.com
world.html')
```

And here's the same line described as a process:

1. Take   the   URL
   `https://scraperwiki.com/hello_-`
   `world.html`
2. Scrape it using the `scrape` function
3. In the `scraperwiki` library
4. And put the result into a variable called `html`

See what I mean?

Back to Scraper #9

So, if we look at that line of code:

```
html = scraperwiki.scrape('https://scraperwiki.com/hello_-
world.html')
```

The = operator is a key thing to look out for. This means that the result of the process on the right of that = sign (scraping a URL using the `scrape` function from the `scraperwiki` library), is given to the variable `html`.

The final line, then:

```
print html
```

...shows us what that variable `html` actually *contains*:

Print doesn't literally send the results to a printer - it just means 'display'. If you click **RUN** to run the code and look in the **Console view** beneath, you'll see the series of lines generated by the first `print` command (which we're not going to explain here), and underneath, the result of `print html`:

```
<!doctype html> <!-- HTML5 Boilerplate -->
```

You may have to scroll quite a bit to get to those first lines: there's a lot of HTML in the scraped webpage.

So, you've scraped a page. Next we need to know how to store the results and do some other things besides, which we'll tackle in the next chapter.

Recap

All that for just 3 lines of code? Yes, because there's a lot to learn in those lines:

- **Libraries** are collections of **functions**
- You **import** a library by, er, using the `import` command
- A library's function is often **called** into action by using the name of the library and function, with a period (full stop) between, such as `scraperwiki.scrape`
- Functions also need **parameters** to be **passed** by putting them in parentheses after the function
- A **variable** contains information you need to manipulate, use, or store. The name of a variable can be anything you like, as long as it isn't the name of another function, library or variable (don't worry: you'll get an error telling you this). A variable is sometimes also referred to as an **object**.
- Variables are **initialised** and changed with the `=` operator, which assigns the value to the right of the `=` (or the results of a function) to the variable on the left.
- The **print** command followed by the variable name can show you the contents of a variable at any particular time.

By the way, you've just learned your first bit of the programming language **Python**. Which is nice.

Tests

These key concepts deserve a good deal of experimentation to become comfortable with the way that they work. So make some time to do the following:

- Replace the URL in the code you used above with another one. What happens when you run the script now?
- Change the name of the variable from 'html' to something else. Does it still work? What if you use 'def' instead. Can you work out why?
- Try to add two lines which change the value of 'html' and then print it again.
- Try to initialise a new variable with a new name. Give it a number value. Then print it.
- Now see if you can write a new line that adds one to that number. Then another line that prints it.
- Search for documentation online that explains how to create a **list variable** in Python, or a **dictionary variable**. What happens when you print that?
- Search for documentation that explains how to print just the first item in that list, or dictionary.

14 Scraper #13 continued: Scraperwiki's tutorial scraper 2

Back to Scraperwiki Classic's documentation page for tutorials[1] and this time the code for the second tutorial here[2] (again, you'll need to copy and edit the URL to remove the `new/python?template=` part).

If you were particularly observant, you might have noticed that the URL `https://scraperwiki.com/hello_-world.html` used in the scraper code no longer exists (this is a good tip to try when adapting someone else's scraper). Instead, the scraper actually ends up scraping the page you are redirected to - a 404 error page.

You'll need to replace that URL with another page containing a table, such as http://inmo.ie/6022[3] (this is the site for a site called Trolley Watch, which monitors bed availability in Irish hospitals). In other words, change `html = scraperwiki.scrape('https://scraperwiki.com/hello_-world.html')` to `html = scraperwiki.scrape('http://inmo.ie/6022')`

Here's the code in full, with that change made, and after

---

[1] https://scraperwiki.com/docs/python/tutorials/
[2] https://classic.scraperwiki.com/editor/raw/tutorial-2
[3] http://inmo.ie/6022

you've followed the instructions in the comments (if you run this you'll get an error at the end - more on this later):

```
#######################################
# START HERE: Tutorial 2: Basic scraping and
saving to the data store.
# Follow the actions listed in BLOCK CAPITALS
below.
#######################################
import scraperwiki
html = scraperwiki.scrape('http://inmo.ie/6022')
print "Click on the ...more link to see the whole
page"
print html
---------------------------
# 1. Parse the raw HTML to get the interesting
bits - the part inside <td> tags.
# -- UNCOMMENT THE 6 LINES BELOW (i.e. delete
the # at the start of the lines)
# -- CLICK THE 'RUN' BUTTON BELOW
# Check the 'Console' tab again, and you'll see
how we're extracting
# the HTML that was inside <td></td> tags.
# We use lxml, which is a Python library
especially for parsing html.
---------------------------
import lxml.html
root = lxml.html.fromstring(html) # turn our
HTML into an lxml object
```

```
    tds = root.cssselect('td') # get all the <td>
tags
    for td in tds:

       print lxml.html.tostring(td) # the full
      HTML tag

        print td.text # just the text inside
      the HTML tag

    ---------------------------
    # 2. Save the data in the ScraperWiki datastore.
    # -- UNCOMMENT THE THREE LINES BELOW
    # -- CLICK THE 'RUN' BUTTON BELOW
    # Check the 'Data' tab - here you'll see the
data saved in the ScraperWiki store.
    ---------------------------
    for td in tds:

       record = { "td" : td.text } # column
      name and value

       scraperwiki.sqlite.save(["td"], record)
      # save the records one by one

    ---------------------------
    # Go back to the Tutorials page and continue to
Tutorial 3 to learn about
    # more complex scraping methods.
    ---------------------------
```

You should now understand the first bunch of four lines where a library is imported, function used, variable initialised with the results, and printed.

The second collection of code has another library imported (lxml.html) which the comments tell us is good for "parsing" (analysing) HTML. If we didn't know that we could still google "library lxml.html" to find the documentation (this is why jargon helps).

We also can work out that `.fromstring` is a function from that library, and that the ingredients it uses are in parentheses: our variable, `html`.

Interestingly, the results are put into a new variable: `root`. Which allows us to learn something new…

Scraping tip #3: follow the variables

Many scrapers are a gradual 'whittling down' of data from full webpages or PDF documents to specific elements within those.

If you're looking at a scraper by someone else, and want to understand how it works, try to trace those variables:

- `html` is initialised first, with the results of a

function being used on a URL. **Search** for

'html' and you find it being used to initialise

another variable:

- `root` - the results of the `.fromstring` function

  being used on `html`. Search for 'root' and you

  find it in the line which initialises...
- `tds` - the results of the `.cssselect` function being used on `root` (more details later). Search for this and you find it on the other side of an = operator initialising...
- `td` - the results of a **for loop** (more on these later) going through the contents of `tds` (a **list**)

You don't (yet) need to know what those functions do, or even what the variables contain at each stage (although you can add `print` commands to show you).

The point is to illustrate the process of tracing an object (in this case, a webpage) as it passes through your scraper, being transformed, whittled, listed and separated along the way.

What are those variables?

You'll notice that the comments on the code talk about an "lxml **object**"...

```
root = lxml.html.fromstring(html) # turn our
HTML into an lxml object
```

...that object is our **variable** `root` at the start of the line (remember I said variables were sometimes referred to as objects?). Turning something into an lxml object allows us to use lxml *functions* on that - which is exactly what happens in the following line:

```
tds = root.cssselect('td') # get all the <td>
tags
```

We can understand this whole line based on what we've learned so far.

- We know that `tds` is a variable because of the `=` operator, and we know it will be given the results of the function on the right.
- We already know that `root` is a variable too
- We can guess that `cssselect` is a function, because it's followed by a parenthesis...
- ...which contains the **parameter**: the information the function needs to work - a **string** (because of the quotation marks).

Put another way: we know there is *a **function** doing something with the **string** 'td' and the **variable** `root`*, and putting the results into a new variable called `tds`.

If you know that '*td*' is a HTML tag (`<td>`), and `root` is a scraped HTML webpage, then you can probably work it out: `cssselect` grabs the contents of all the `<td>` tags in the HTML page contained in the variable `root`, and puts them in the new variable `tds`.

Because there is *more than one* `<td>` tag, that new variable is a **list**.

If we want to see the contents of that variable to find out more and check if it is a list, we can use the print command again by adding a line after the `tds` variable is created as follows:

```
tds = root.cssselect('td') # get all the <td>
tags
    print tds
```

...and then **RUN** the script. The part where it prints that variable `tds` will look like this:

```
[<Element  td  at  0x28cf2f0>,  <Element  td  at
0x28cf290>]
```

> You may want to *comment out* `print html` so that your results aren't dominated by the code of the scraped page. **Commenting out** a line simply means putting a hash sign at the start so it is treated as a comment rather than working code. The result in this case looks like this:
>
> ```
> #print html
> ```

Because `tds` is an lxml object, you can't actually see the details (later code converts it back into an object we

can understand), but you can see that this is a list, because it has the following qualities:

- **A list is contained in square brackets**
- Each item in the list is **separated by a comma**

So a list of names might look like this:
```
['Paul', 'Kevin', 'Barbara']
```
And a list of ages might look like this:
```
[21, 32, 85]
```
And you can create a variable containing a list like so:
```
their_names = ['Paul', 'Kevin', 'Barbara']
their_ages = [21, 32, 85]
```
The final part of this code is a **for loop**:
```
for td in tds:

  print lxml.html.tostring(td) # the full
 HTML tag

  print td.text # just the text inside
 the HTML tag
```

...and this means another detour

Detour: loops (for and while)

If you see a line which begins with the command **for**, or **while**, then you know this is a **loop**.

A loop is a way of doing something over and over again - looping over and over until something happens to

break the loop (in a **for loop**, when it reaches the end of the list; in a **while loop**, when the condition set at the start is no longer met, i.e. 'while this is the case, do that').

It's great for doing something over and over again *to* or *with* each item in a list, or doing something over and over again *while* a particular condition is met (for example: while something exists, or while a value is above or below a particular level).

What the loop actually does is described in the lines that come under the for command and after a **colon** (the colon is important - it won't work if it's missing):

```
for td in tds:

  print lxml.html.tostring(td) # the full
HTML tag

  print td.text # just the text inside
the HTML tag
```

You'll notice that these are also indented. That's important - if you don't indent, then it will be treated as outside of that loop. You can indent a line by either pressing the TAB button, or pressing the space bar four times.

In this case you can read it as follows:

for each item in the tds *list:*

print the results of using a particular function on that item

print a particular property of that item (.text)

There are two items in this list, so that loop runs twice: it prints those two pieces of information for the *first* item, and then prints them for the *second* item.

If you really want to blow your mind, think about this: as the for loop goes round and round, it does the following: creates a td variable, gives it the value of the first item of the tds list, then executes the command, then gives it the value of the second item of the tds list, and so on. When you're dealing with long lists, that td variable can be changing many times per second.

PS: If you want to find out more about loops you can, of course, google 'loops Python' or something similar. The free ebook *Learn Python The Hard Way* has two simple chapters introducing the for and while loops at http://learnpythonthehardway.org/book/ex32.html[4] and http://learnpythonthehardway.org/book/ex33.html[5] - the book as a whole is recommended too.

Back to scraper #13: Storing the data

Now we can understand the beginning of the final chunk of code, which is also a **for loop**:

```
for td in tds:

    record = { "td" : td.text } # column
  name and value
```

---

[4]http://learnpythonthehardway.org/book/ex32.html
[5]http://learnpythonthehardway.org/book/ex33.html

```
 scraperwiki.sqlite.save(["td"], record)
# save the records one by one
```

Whereas the last loop only printed each item in the list, this one does something else with each item in the list, and we can again use the knowledge gained so far to make some educated guesses about it:

- `record` is a **variable** (it's followed by the = operator) so we could call it anything we wanted (try it - note that it's used again in the next line, so if you change the name in one place, you need to change it anywhere else that it's used).
- `save` is a **function** (it's followed by parentheses) from the `scraperwiki` library that does something specific. We could search the documentation to find out more.

But the rest needs some explanation;

```
record = { "td" : td.text } # column name and
value
    scraperwiki.sqlite.save(["td"], record) # save
the records one by one
```

When I introduced variables earlier I mentioned the different types: strings, integers and floating points, lists, and dictionaries. The variable being created on the first line is a **dictionary**.

We know this because of two clues:

- **curly brackets** - just as lists use square brackets, a dictionary uses curly ones like so: {

- **a pair of values separated by a colon** - a dictionary, as explained earlier, is a way of storing pairs (just as a physical dictionary contains pairs: words, paired with their definition).

Once you know that you can spot a dictionary variable.

In this case the pair consists of a string (`"td"`) and the `.text` of another variable: `td`. There's potential for confusion here, because the string and the variable both look the same.

The quotation marks are key - *if it has quotation marks then it's a string, and stands alone.* If it doesn't, then it's something different.

What we're really doing here is filling a column in a table. "td" is just the header for that column - a label. But the text contained within the td variable (`td.text`) - rather than the whole contents including the HTML tag surrounding the text - is what goes in that column.

The final line stores the results in a database:

```
 scraperwiki.sqlite.save(["td"], record) # save
the records one by one
```

Again, most of this we can work out by using the tips introduced earlier: look for the documentation. A google for "scraperwiki documentation scrape function" will take you to this page: https://scraperwiki.com/docs/python/python_-help_documentation/[6] - and searching within that page for ".save" takes you to the part of that documentation which specifically deals with the `save` function.

---

[6] https://scraperwiki.com/docs/python/python_help_documentation/

This is the same process that we followed in looking at the Google Docs Help pages (documentation) for our first function, importHTML. It confirms what was pretty obvious: that this function saves something, but it also explains where it's saving, and what the parameters mean.

```
scraperwiki.sqlite.save(unique_keys, data[,
table_name="swdata", verbose=2])
```

"Saves a data record into the datastore into the table given by table_name.

"data is a dict object with field names as keys; unique_keys is a subset of data.keys() which determines when a record is overwritten.

"For large numbers of records data can be a list of dicts.

"verbose alters what is shown in the Data tab of the editor."

The first parameter, then, is *unique_keys*. This corresponds to `["td"]` in our code.

The second parameter is *data*, which corresponds to `record`, the variable containing the dictionary.

The documentation assumes you know what a key is. I won't.

Detour: Unique keys, primary keys, and databases

Databases often have what's called a **primary key**: this is a column in your data where no two entries will be the same. For instance employee ID codes might be used as a primary key (no two employees have the same one), but employee surnames wouldn't be (two employees might have the same surname). Sometimes, if the raw data doesn't include anything unique, an arbitrary primary key (such as row or entry number) might be added to help distinguish data later. A table can only have one primary key.

A **unique key** is very similar, as the name suggests, but you can have more than one in a table.

So `["td"]` is the key for our data. It corresponds to the "`td`" in the previous line: the header of a column of data (that column now becomes the container of our unique key, i.e. each value in it must be unique).

How can we be sure? Trial and error. Try changing one or the other and the script generates an error when you run it (something like `sqlite3.OperationalError: table swdata has no column named dog`).

Change both, and the error disappears - so you know they're the same thing.

However, you may have tried to run the code where both the key and our data are called `"td"`, and still got an error. We'd better sort that out.

A unique key can't be empty: fixing the ValueError

As the code stands, it generates the following error when run:

```
ValueError: You passed no sample values, or all the values you passed were null.
```

If you ever see an error like this, it is likely because you are trying to store 'null' data - that is, no data at all - against a key which is being used as the unique key.

For some extra clues, look at the lines above that error message to the one that starts

```
Traceback (most recent call last):
```

This details the lines that generated the error. The first is this:

```
File "./code/scraper", line 39, in <module>
```

That means that line 39 has generated our error. And rather helpfully it also shows us that:

```
scraperwiki.sqlite.save(["td"], record) # save the records one by one
```

So we know it's the line that saves our data with the unique key. There are some other lines, but these refer to code *outside* of our scraper - for example this one:

```
File "/usr/local/lib/python2.7/dist-packages/scraperwiki/sqli
line 31, in save dt.create_table(data, table_name
= table_name, error_if_exists = False)
```

Refers to the code 'called' by `scraperwiki.sqlite`. You cannot change this, so don't worry about it. Likewise `File "/usr/local/lib/python2.7/dist-packages/dumptruck/dumptruck.py",`

```
line 204
```

So, back to line 39. As explained, our problem is trying to store 'null' data. Because we're only storing one thing - `td.text` - we can at least isolate the problem. If this wasn't the case, you might try printing each element to identify which one was 'null', or reducing what was saved until you can isolate the culprit.

So how do we stop `td.text` being null?

Actually, that may be the wrong question. We might also ask: How do we *stop an error being generated by* `td.text` being null?

The answer to that comes from our first experiences with Python two chapters ago: the word (**statement**, in programming jargon) `try:`.

Here's how we might add a `try:` statement to our code so that it *tries* to run the lines that save `td.text`, but also has an option if it hits a problem:

```
for td in tds:

  record = { "td" : td.text } # column
name and value

  try:

      scraperwiki.sqlite.save(["td"],
     record) # save the records one
     by one

  except:
```

```
 record = { "td" : "NO ENTRY"
} # column name and value

 scraperwiki.sqlite.save(["td"],
record) # save the records one
by one
```

The `try:` is added as a line before the line that saves the data. Note that `try:` ends in a colon, and **anything to be *tried* - i.e. the next line - needs to be indented**, just as with `for:`.

Because `try:` is being used *within* `for:` (and so is indented itself) the line `scraperwiki.sqlite.save(["td"], record)` # save the records one by one is actually indented twice.

Note also that **we need an `except:` *after* the `try:`** (but not indented within it) to specify what happens if the thing being tried generates an error.

Once again, all lines *within* the `except:` (that is, all lines to be executed if it is triggered) need to be indented.

To that end, we add a new line which changes the value of "td":

```
 record = { "td" : "NO ENTRY"
} # column name and value
```

...before saving *that*.

In other words, this section of code says: *try* to save the record, *except* when its value would generate an error, in which case give it a new value and save that.

There are other solutions we could try, such as:

- Testing whether td.text is null before saving
- Changing `td.text` to grab something less specific.

On the latter, for example, we could have instead used `lxml.html.tostring(td)`. This actually grabs the full HTML within `td`, and does not grab any 'null' values. `td.text` on the other hand, grabs any text immediately *after* the `<td>` tag, but also *before* any other tag, as we'll see in the next chapter.

Trial and error, partnered with a reading of the documentation around the functions and methods being used, will help you to narrow down things until you understand the culprit for any errors.

This may feel like failure at first - but it's not: it's *valuable experience.* You'll encounter these errors many times and they're part of the process. Solving them eventually becomes part of the enjoyment.

Summing up the scraper

After that detour into keys we now have everything we need to understand the scraper - and more importantly, others in future. Here's what the scraper is doing:

1. First, it scrapes the whole page using the `scraperwiki.scrape` function, and stores it in `html`
2. Then it turns `html` into an lxml object (because we need to use an lxml function), and stores it in `root`

3 Then it finds the ‹td› tags in that page, and their contents, using the lxml function cssselect. A list of the results are stored in the variable tds.

1. Then it loops through each result in that tds list, and (temporarily) stores it in a variable called td.
2. While it's stored in that variable, it extracts the text (not the code) and stores it in a variable called record.
3. And saves that record in a database, before continuing the loop with the next item in the list (extracting and storing again), until the list is finished.

We can remove all the print commands (they're only there to show us what's happening), and the **for loop** which only consists of print commands and comments and there would only be a dozen lines of code:

```
import scraperwiki
html = scraperwiki.scrape('http://inmo.ie/6022')
import lxml.html
root = lxml.html.fromstring(html) # turn our
HTML into an lxml object
tds = root.cssselect('td') # get all the ‹td›
tags
for td in tds:

  record = { "td" : td.text } # column
 name and value

  try:
```

```
    scraperwiki.sqlite.save(["td"],
  record) # save the records one
  by one

except:

   record = { "td" : "NO ENTRY"
  } # column name and value

   scraperwiki.sqlite.save(["td"],
  record) # save the records one
  by one
```

One final change: the sqlite.save function dates from Scraperwiki Classic. It still works for now, but it's best to use the newer scraperwiki.sql.save function in case the old one ever stops working.

The *two* lines in our code, then, should be changed to this:

```
   scraperwiki.sql.save(["td"],
  record) # save the records one
  by one
```

There's just one more question: where has the data been stored to?

After you've **RUN** the scraper, click on **View in a table** (in the upper middle area of the scraper) and you'll see the data you just scraped.

There is one column - labelled "td" - and your data: the contents of the ‹td› tags in the page you scraped (where there was not another tag immediately after it).

If you have more than one column it may be because you ran the scraper more than once and changed the label (with a column for each label - both will contain the same data). If this is the case, go back to your code, click on the orange **Clear** button around the bottom right corner, and then select **Clear dataset** to empty the datastore. You can go back to your scraper and run it one more time to re-fill with cleaner data.

Once you're happy with the data, click on **Download as spreadsheet** above it to get two download links: an Excel spreadsheet (ending in .xlsx) or as CSV (Comma Separated Values), which will work in Excel, Google Docs and any other spreadsheet package. You can also copy the links to these files and share publicly online if you want others to be able to download them.

Recap

Once again, we've spent a lot of time on understanding just a few lines, because this will pay off later on. Here are the key things to take away:

- One of the best ways to understand a scraper is to **follow the variables** as they are used to create new ones (typically, each one containing more and more specific parts of your data)
- **Lists** use square brackets and commas, e.g. ['David Cameron', 'Gordon Brown', 'Tony Blair']
- **Dictionaries** store lists of **pairs**, and use curly brackets, colons and commas, e.g. {'Party' : 'Conservative', 'Age' : 60}
- A **for loop** repeats an action for every item in a list, for example extracting further data, and/or storing them in a dataset. It repeats this (loops) until it finishes the list.
- The actions come after a **colon** at the end of the line containing the for command (it won't work without it), and are **indented** below it (again, if you get an error message, it may be because the indents are missing)
- The `try:` **statement** allows you to cope with errors generated by the code indented afterwards. If an error is generated then the code skips to the `except:` **statement** which follows, and executes the code indented after that instead.

- Data is saved using the `scraperwiki.sql.save` function - in the Classic Scraperwiki a slightly different `sqlite.save` function was used instead. It still works, but it's best to use the newer option in case the old one ever stops working.
- A unique **key** is the column which is most unique in your data (ideally one that won't have empty records, as this can trip up the scraper)
- And, once saved (by running the script), you can download the data from the **Download as spreadsheet** link.

Now we can start to adapt this for our own scraper.

Tests

- Try changing the key in the line where the data is saved. What happens? Why?
- Can you adapt the code so that it grabs another property of `td`? Tip: look through the documentation to see what other properties there are, and what other properties `td` might have.
- Can you adapt the code to grab a different HTML tag? Tip: you'll need to look at the raw HTML of the page you're scraping to see what other tags are used.
- Write down on a piece of paper what happens to the webpage as it gets passed from variable to variable. Change the names of a variable in one place and see where else you'll have to change it. Try clicking **RUN**

and see if a new error is generated because you forgot
to rename it in every place.

15 Scraper #14: Adapting the code to scrape a different webpage

We may have scraped all the cells from that page - but we haven't exactly got them in any order. At the moment, they're all being put in the same column. So let's fix that.

Here's the code when we last left it:

```
import scraperwiki
html = scraperwiki.scrape('http://inmo.ie/6022')
import lxml.html
root = lxml.html.fromstring(html) # turn our
HTML into an lxml object
tds = root.cssselect('td') # get all the <td>
tags
for td in tds:

  record = { "td" : td.text } # column
 name and value

  try:

    scraperwiki.sqlite.save(["td"],
    record) # save the records one
    by one

  except:

    record = { "td" : "NO ENTRY"
    } # column name and value

    scraperwiki.sqlite.save(["td"],
    record) # save the records one
    by one
```

Look through that code and **see if you still understand
what it's doing**. Don't worry about knowing what every
function or library does, or even the specific jargon of
dictionaries and lists.

The key thing is that you have a general idea of what's happening to the contents of the webpage, and how that's being whittled down by various functions in the scraper before being saved.

Dealing with errors

Can we do better with the error that made us add a `try:`? We have a few options:

- **Read the documentation** for the libraries we're using to see if we can tweak the code to avoid that problem - particularly the one that extracts data (because it's a lack of data that's our problem): **lxml.html**.
- **Use trial and error** on possible solutions. For example, what about using a different value for our unique key?
- **Find what causes the error**. We know it's an entry which is 'None' that's causing the error. But where is that? Can we somehow skip it?

It's likely to be a mix of all the above.

I said in the last chapter that `td.text` grabs any text immediately *after* the `<td>` tag, but also *before* any other tag.

But there is text inside the tag, isn't there? Well, it depends how fussy you are. The `<td>` tag is followed instantly by another tag *before any text* appears on a number of occasions, for example:

```
<td><a href="/0">Home</a></td>
<td><input type="text">
```

- And that's our problem.

We could **test this theory** by looking at other similar examples and comparing them to tags where there is text after `<td>`.

In the documentation (http://lxml.de/lxmlhtml.html[1] - search for ".text") we can look for a different option to `.text` that wouldn't be so fussy (there's also a guide to "parsing HTML" on Classic Scraperwiki[2].)

Don't worry if you don't understand all of the documentation - use trial and error to *see* what each option does. For example, you could change `print td.text` to:

- `print td.tag`
- `print td.attrib['href']`
- `print td.tail`
- `print td.text_content()`

...and then compare the results to the original HTML.

If you do, one in particular might stand out: `.text_content()`

This is the one that solves your problem, because it doesn't just grab any text immediately after `<td>`, but "the text content of the element, including the text content of its children, with no markup." (http://lxml.de/lxmlhtml.html[3].)

---

[1] http://lxml.de/lxmlhtml.html
[2] https://scraperwiki.com/docs/python/python_css_guide/
[3] http://lxml.de/lxmlhtml.html

'Children' in HTML means tags within other tags. In our example above, ‹a› is a child of ‹td›.

Here's the full code without comments once you've made that change to both the `print` command and the creation of each `record` in the for loop:

```
import scraperwiki
html = scraperwiki.scrape('http://inmo.ie/6022')
import lxml.html
root = lxml.html.fromstring(html) # turn our
HTML into an lxml object
tds = root.cssselect('td') # get all the ‹td›
tags
for td in tds:

   print td.text_content() # just the text
   inside the HTML tag

for td in tds:

   record = { "td" : td.text_content() }
  # column name and value

   scraperwiki.sql.save(["td"], record) #
  save the records one by one
```

This solves the problem. But it took longer than using Google Docs for the same purpose in our first scraper (and the results still need tidying). So it also teaches a final lesson to sum up alongside the others...

Recap

- **Use the fastest tool to get the job done**. If your data is in a simple table on a public webpage, don't over-complicate it: just use a Google Docs =importHTML scraper.
- **Adapt existing scrapers as a way of learning** - even if the scraper itself doesn't work properly in the end.
- If you hit an error, use a range of techniques to try to solve it: reading **documentation** to solve the error with code; **trial and error**; printing variables at each stage to trace the **cause of the problem in the scraped content**.

We're going to move on now to another tutorial which will help us solve some of the remaining problems with this experiment in adaptation.

Tests

- If you haven't already, change `print td.text` to: `print td.tag` or to `print td.attrib['href']` or `print td.tail` or `print td.text_content()`. What is happening? Does the documentation help explain?
- Try scraping the table rows (`<tr>`) - how would you do that? What happens when you do?
- Look at the HTML of the page - are there other attributes that you can grab (other than href)?

- Try to adapt the scraper for another webpage - perhaps one that lists the addresses for race courses.

16 Scraper #15: Scraping multiple cells and pages

Top-selling albums of all time: Page 1

Wikipedia's list of the top-selling albums in history, ready to scrape.

Album	Released	
Thriller	1982	Pop /
Back in Black	1980	Hard roc

The third tutorial on Scraperwiki's Live tutorials page[1] covers another key set of skills.

Open it up by right-clicking on **How to write a screen scraper 3**, and copying the link. Once again you'll have to paste it and remove the `new/python?template=` part of the URL so that it's not trying to create a new scraper on Scraperwiki Classic itself.

That should leave you with the following URL[2]: `https://classic.scraperwiki.com/scrapers/how-to-write-a-screen-scraper-3` (*don't copy and paste from this text as I've left a space for formatting reasons*).

Here's the full code you see when you click **View**

---

[1] https://scraperwiki.com/docs/python/tutorials/

[2] https://classic.scraperwiki.com/scrapers/how-to-write-a-screen-scraper-3/

source. There's a lot more of it than we've been used to, and lots of extra elements we're going to cover:

```
# START HERE: Tutorial 3: More advanced scraping.
Shows how to follow 'next'
# links from page to page: use functions, so you
can call the same code
# repeatedly. SCROLL TO THE BOTTOM TO SEE THE
START OF THE SCRAPER.
import scraperwiki
import urlparse
import lxml.html
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):

   rows = root.cssselect("table.data tr")
  # selects all <tr> blocks within <table
  class="data">

   for row in rows:

   # Set up our data record - we'll need
  it later

       record = {}

       table_cells = row.cssselect("td")

       if table_cells:

           record['Artist'] = table_-
          cells[0].text
```

```
            record['Album'] = table_-
          cells[1].text

            record['Released'] =
          table_cells[2].text

            record['Sales m'] =
          table_cells[4].text

            # Print out the data
          we've gathered

           print record, '------------'

            # Finally, save the
          record to the datastore
          - 'Artist' is our unique
          key

           scraperwiki.sqlite.save(["Artist"],
          record)

   # scrape_and_look_for_next_link function: calls
the scrape_table
   # function, then hunts for a 'next' link: if one
is found, calls itself again
   def scrape_and_look_for_next_link(url):

      html = scraperwiki.scrape(url)

      print html
```

```
    root = lxml.html.fromstring(html)

    scrape_table(root)

    next_link = root.cssselect("a.next")

    print next_link

    if next_link:

        next_url = urlparse.urljoin(base_-
        url, next_link[0].attrib.get('href'))

         print next_url

         scrape_and_look_for_next_link(next_-
         url)

# ----------------------
# START HERE: define your starting URL - then
# call a function to scrape the first page in
the series.
# ----------------------
base_url = 'http://www.madingley.org/uploaded/'
starting_url = urlparse.urljoin(base_url, 'example_-
table_1.html')
scrape_and_look_for_next_link(starting_url)
```

Let's start with what we should recognise:

- **Libraries** being imported at the start - two we've
  used; one is new

- **Functions** from those libraries - and other functions - being used with parameters in brackets. Most of these we've used before.
- **Variables** being created with the = operator, and a **for loop**

What's new?

- **def**
- **if**
- numbers in square brackets
- Colour coding?

One thing we haven't mentioned so far is the colour coding used in Python code. This gives us extra clues as we write:

- **Red** indicates a python **statement**. `For`, `import` and `print` are ones you've already used. Statements perform particular actions, as you've already seen. The colour coding helps you avoid using the same terms for variables or anything else.
- **Yellow** indicates a **string**. This helps you avoid problems when you forget to add a closing quotation mark at the end of a string (which would cause all the code afterwards to go red).
- **Purple** indicates a **number** (an **integer** or **floating point**). This is helpful if for any reason you want a number to be treated as a string (in which case you would add quotation marks around it).

Now that you know that anything in red is a statement, you can google "def statement python" or similar for any other statement. You can also get a full list by googling 'Python statements'.

If you did that, you could find out that `def` creates a new function and `if` allows you to do something only if a test is met (for example, something is over or under a particular value, or exists at all). Let's explain both:

Creating your own functions: `def`

So far we've dealt with functions that other people have created: `scraperwiki.scrape` to grab HTML pages; `lxml.html` to convert those into lxml objects; `cssselect` to extract parts of HTML; and `scraperwiki.save` to save the results.

But if we want to do something specific, we'll have to create our own functions.

You create a new function with **def**, like this:

```
def the_name_of_the_function(parameters):
```

Like variables, you can give a function any name you want, as long as it doesn't happen to be the same as one of the basic python statements such as print, etc. (You'll see if it is because it will turn red - but best to pick names for your functions that are unique and describe what they do, like scrape_first_page)

When you create a new function, you also **name any parameters** that it will need, in parentheses. Again, these can be anything you want: the parameters will be used in the lines following **the colon at the end** of the `def` line (the

colon is important - if you miss it out you'll get an error).

The lines that tell your function what to do will all be **indented** underneath (use a tab, or four spaces). So it's easy to tell where the bit defining your function ends and the rest of the code begins, because the code will *not* be indented: it will start from the beginning of the line again.

Here's the second function created in our script, then:

```
def scrape_and_look_for_next_link(url):

    html = scraperwiki.scrape(url)

    print html

    root = lxml.html.fromstring(html)

    scrape_table(root)

    next_link = root.cssselect("a.next")

    print next_link

    if next_link:

        next_url = urlparse.urljoin(base_-
        url, next_link[0].attrib.get('href'))

        print next_url

        scrape_and_look_for_next_link(next_-
        url)
```

The comments help in describing some of what the function does:

```
# scrape_and_look_for_next_link function: calls
the scrape_table
# function, then hunts for a 'next' link: if one
is found, calls itself again
```

You'll also be able to use your previous experience to figure out what the function does. For example, **following the variable** and **reading from right to left** is particularly useful here:

1. The variable `url` is created as the function's only parameter
2. `url` is scraped with the `scraperwiki.scrape` function, and the results put into a new variable called `html`
3. `html` is printed so we can see its contents
4. `html` is then used as the parameter for the `fromstring` function, and the results put into a new variable, `root`.
5. The function `scrape_table` is used to do something with `root`...

...Hold on - what's the `scrape_table` function? Where's that come from?

The answer is: it's *another* function created in this script. If we want to keep following the variable, we need to **go over to that** and *only* return to the rest of this function once we've followed *that function* through.

It's essentially a function-within-a-function, which can hurt your head if you think about it too much, so here's a simplified version:

```
DEF FUNCTION 1
> FIRST PART OF FUNCTION 1
> FUNCTION 2 TAKES SOMETHING FROM IT (PARAMETERS)
>> FUNCTION 2 RUNS, FINISHES
> SECOND PART OF FUNCTION 1 RUNS
```

The simplest way to trace this new function to use the browser's Find facility to search for it. When you do find it, you'll notice that that, too, is created with the `def` statement:

```
def scrape_table(root):
```

Here's the code for the whole function, including the comment preceding it, so we can follow from there:

```
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):

  rows = root.cssselect("table.data tr")
 # selects all <tr> blocks within <table
 class="data">

  for row in rows:

      # Set up our data record -
     we'll need it later

     record = {}

     table_cells = row.cssselect("td")
```

```
if table_cells:

    record['Artist'] = table_-
    cells[0].text

    record['Album'] = table_-
    cells[1].text

    record['Released'] =
    table_cells[2].text

    record['Sales m'] =
    table_cells[4].text

    # Print out the data
    we've gathered

    print record, '------------'

    # Finally, save the
    record to the datastore
    - 'Artist' is our unique
    key

    scraperwiki.sqlite.save(["Artist"],
    record)
```

And here's our list following the variable with that function now included:

1. The variable url is created as the function's only parameter

2. `url` is scraped with the `scraperwiki.scrape` function, and the results put into a new variable called `html`

3. `html` is printed so we can see its contents

4. `html` is then used as the parameter for the `fromstring` function, and the results put into a new variable, `root`.

5. The function `scrape_table` is used to do something with `root`

- `root` is used with the `cssselect` function, which extracts all the HTML in `<table class="data"><tr>` tags and puts them into a **list variable** called `rows`

- A **for loop** takes each individual item in that list, storing it as `row`,

- Creates a new (empty) variable called `record`,

- And a new variable called `table_cells`, which uses the `cssselect` function again to extract all `<td>` tags.

- If there are any `<td>` tags (we'll explain the **if** statement below):

- Then the text between each tag is put in the new variable called `record`, (under four different labels)

- And saved to the datastore with Artist as the unique key

1. Now, resuming the first function where we left off: the `root` variable is also created to initialise another variable called `news_link`, (containing the contents of `<a class="next">`). This is printed.
2. If the variable isn't empty, then it's used as one of the parameters for the `urlparse.urljoin` function to create a new variable: `next_url`,
3. Which is then used as the parameter for the function `scrape_and_look_for_next_link`
4. Which is actually this function - so in other words, it's looping back to the start of the function and running all over again. It will only stop when the answer to that final if question is 'No, it's empty'. So time to explain that...

If statements - asking a question

Our script contains two sections that start with the word `if`. It's coloured red, so we know it's a **statement**, and we can **search the web to find out more**, e.g. 'if python' or 'if statement python'. There's lots out there, so if you find one explanation too opaque, search for others, or add extra words like 'for beginners' or 'made simple'.

'**If**' is pretty self-explanatory: it says **if** something is the case, then do something. For example:

```
if age = 40:
```

```
print "You are 40"
```

You'll notice that the if statements in our code don't appear to ask anything:

```
if next_link:
```

And:

```
if table_cells:
```

Actually, there *is* a question there, but it's implied. The question is "if (the variable) next_link (contains anything)", or rather "If this exists:"

It helps to write the code another way, like this:

```
if next_link is not None:
```

But it will work both ways - and you should make sure you can understand either if you come across them.

As with the **for** and **def** statements, the line beginning `if` should **end with a colon**, and everything in **the indented lines after that colon** will contain **the things that you want to code to do if the condition is met**.

Look at the two if statements in our code and try to work out what they do. Here:

```
if next_link:

    next_url = urlparse.urljoin(base_-
    url, next_link[0].attrib.get('href'))
```

```
print next_url

scrape_and_look_for_next_link(next_-
url)
```

And here:

```
if table_cells:

    record['Artist'] = table_-
    cells[0].text

    record['Album'] = table_-
    cells[1].text

     record['Released'] =
    table_cells[2].text

     record['Sales m'] =
    table_cells[4].text

     # Print out the data
    we've gathered

     print record, '------------'

      # Finally, save the
    record to the datastore
    - 'Artist' is our unique
    key
```

```
scraperwiki.sql.save(["Artist"],
record)
```

The more you exercise this problem-solving, the better you will get at coding scrapers. Reading books like this will give you some of the wider knowledge, but the actual skills come from practice.

Write what you understand - and questions on what you don't - as comments in the code before each line you're commenting on (start comments with the hash sign # so they don't run as code).

Here's my explanation included as comments before each line:

```
#If there's anything in the variable next_link:

if next_link:
```

```
#Create a new variable called next_url. Use the
urlparse.urljoin function on the variables base_url
and next_link - we need to look up the documentation
for attrib.get('href') to find out what that does

        next_url = urlparse.urljoin(base_-
        url, next_link[0].attrib.get('href'))
```

```
#print the contents of that variable next_url

        print next_url
```

```
#run a function called scrape_and_look_for_-
next_link on the variable next_url
```

```
scrape_and_look_for_next_link(next_-
url)
```

And here:

```
if table_cells:
```

```
#add to the variable record (a dictionary) these
pairs: label 'artist': the text in the first [0]
table_cells item...
```

```
record['Artist'] = table_-
cells[0].text
```

```
#... label 'album': the text in the second [1]
table_cells item...
```

```
record['Album'] = table_-
cells[1].text
```

```
#... label 'released': the text in the third
[2] table_cells item...
```

```
record['Released'] =
table_cells[2].text
```

```
#... label 'sales m': the text in the fifth [4]
table_cells item...
```

```
record['Sales m'] =
table_cells[4].text
```

```
    # Print out the data we've gathered into the
record variable, followed by '-----'

            print record, '------------'

             # Finally, save the
            record to the datastore
            - 'Artist' is our unique
            key

            scraperwiki.sql.save(["Artist"],
            record)
```

Quite often, after the if statement is finished, you will see an **else** or an **elif** statement (short for *else if*). The **else** and **elif** statements are what the script should do if the original if statement is not met (which also helps in avoiding errors that would be generated otherwise). For example:

if the age is above 40:

   print "you are older than 40"

elif the age is below 40:

   print "you are younger than 40"

else:

   print "you are 40"

Taken as a whole, these are called **if-else statements** and you see them in all sorts of programming situations, for obvious reasons: they allow you to take different actions depending on what you have.

There are a couple of elements still to explain...

Numbers in square brackets: indexes again

One part of that code which is new is where there are numbers in square brackets:

```
 record['Artist'] = table_-
cells[0].text

 record['Album'] = table_cells[1].text

 record['Released'] = table_-
cells[2].text

 record['Sales m'] = table_-
cells[4].text
```

These are **index**es - we came across these in our very first scraper using importHTML in Google Docs.

An **index**, you might remember, refers to an item's position. So in that Google Docs code:

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

The number 1 at the end meant 'the first' (reading right to left: the first, table, at that URL)

Google Docs, however, is a bit unusual: an index of 1 in a Google spreadsheet means 'first'. But in most programming, **indexes begin at 0**, so 'first' is 0, 'second' is 1, and so on. This might sound counter-intuitive (which presumably is why Google Docs start from 1) but it does make logical sense. And programming is logical by nature.

If you're dealing with lists of things then **indexes allow you to grab particular parts of the list**. For example (and you can add this code to a scraper to see it working):

```
myfamily = ['brother', 'sister', 'mother', 'father']
print myfamily[0]
```

...would print the first item in the `myfamily` list variable, specified with `[0]`.

You can now guess how to adapt this to print the other items.

It also works from the back with negative numbers. Try this:

```
myfamily = ['brother', 'sister', 'mother', 'father']
print myfamily[-1]
```

[-1] indicates the last item in a list; `[-2]` the second to last, and so on (just to confuse things, these numbers don't start from -0, but again this does make logical sense, as there's no difference between -0 and 0).

That's particularly useful to know when you are scraping pages where the information you want always comes last.

Looking at our code, then, we can now understand what those numbers in square brackets are doing:

```
 record['Artist'] = table_-
cells[0].text

 record['Album'] = table_cells[1].text

 record['Released'] = table_-
cells[2].text

 record['Sales m'] = table_-
cells[4].text
```

table_cells[0] is the first item in the table_cells list, table_cells[1] the second, and so on. Adding .text to the end simply grabs just the text from that item (not the HTML tag), by the way.

Attributes

This now gives us an extra piece of information for the final piece of code to explain:

```
next_url = urlparse.urljoin(base_-
url, next_link[0].attrib.get('href'))
```

We now know that next_link[0] must mean the first item in the next_link list variable. And .attrib.get('href') is probably extracting a particular part of that, just as .text did.

Again, the tip here is to try different searches involving the phrase `.attrib.get` to see what comes up. You'll quickly see mentions of lxml.html in the results and, indeed, if you remembered that `.text` was from that library (and that the whole library is used to extract bits of HTML) then you might have guessed it was from that.

Actually, the specific combination `.attrib.get` isn't mentioned in the documentation at http://lxml.de/tutorial.html[3], but `.attrib` and `.get` are mentioned separately, and you can work it out by using the print command somewhere after `next_link` has been created:

```
print next_link[0].attrib.get('href')
```

...and then see what the console at the bottom prints at that point and comparing it with the HTML in the page being scraped:

`example_table_2.html`

What `.attrib.get('href')` grabs, then, is the contents of the *attribute* href=" in the first item in the list variable `next_link`, which (if we follow the variable back in the code) was created by grabbing all the contents of `<a class="next">` tags.

So, again, it's all about drilling further and further into the content we're scraping.

---

[3]http://lxml.de/tutorial.html

Scraping tip #4: follow the functions

Like variables, once you see a function being created with **def**, you can search for the name of a function to see where it's used, and what parameters are used (programmers say the parameter (or 'argument') is '**passed**' or that we are '**passing**' a parameter (or 'argument') to the function) when it is.

In this case, one function is first used ('**called**') at the bottom:

```
# START HERE: define your starting URL - then

# call a function to scrape the first page in
the series.

# -----------------------

base_url = 'http://www.madingley.org/uploaded/'

starting_url  =  urlparse.urljoin(base_url,
'example_table_1.html')

scrape_and_look_for_next_link(starting_url)
```

And we already know that the other function is used *within* the first one to be called.

Because functions are often used with variables (as their parameters) it sometimes help to list them on a separate piece of paper like this:

Function/method/etc used	Variable(s) used?	Variable created
n/a	no - a string: 'http://www.madingley.org/uploaded/'	base_url
urlparse.urljoin	base_url, 'example_table_1.html'	starting_url
scrape_and_look_for_next_link	starting_url	url (when def is used to create function, parameter is renamed)
scraperwiki.scrape	url	html
lxml.html.fromstring	html	root
scrape_table	root	root (when def is used to create function, parameter uses name root but could be anything)
.cssselect	root	rows

for	rows	row
.cssselect	row	table_cells
.text	table_cells[0] etc.	record['Artist'] etc.
scraperwiki.sqlite.save	record	
.cssselect	root	next_link
.attrib.get('href')	next_link[0]	next_url (created by function below)
urlparse.urljoin	base_url, next_link[0]	next_url
scrape_and_look_for_next_link	next_url	url (when def is used to create function, parameter is renamed)

Scraping tip #5: Read from bottom to top

If you went through the process of 'following the functions' you may have noticed something else about reading scrapers: this one started working at the bottom, and moved towards the top, through the second and then the first functions to be created.

This is quite common, and here's why:

A scraper is like a recipe: it's a set of instructions for making something, with various ingredients. Some ingredients come ready-made (functions from libraries, and Python's own statements like if, def, print, and for), and others you need to prepare yourself (which you do with the def statement).

Before you start cooking, you have to prepare those ingredients. You have to make the pizza base before you cook the pizza; or marinade the meat before you cook that.

(*Don't worry: I'm almost done with this metaphor.*)

Your scraper works the same way: most of it consists of preparation: you need to import your libraries before you can use functions from them (think of it as your trip to the supermarket to get the ready-made ingredients); and you need to prepare your own custom-made functions before you can use them.

A code runs from top to bottom, but a function won't work unless it's been written first. So it's only in the final lines of your scraper that you can say 'Go cook', and all that preparation starts to pay off.

If you're trying to understand a scraper *written by someone else*, then, the bottom is a good place to

start: it's where you can see that cooking start, and
as ingredients are used, you can trace those back to
the lines higher up where they had been prepared.

Recap

We've now covered the last key elements of Python and
scraping. Here's a recap:

- You can create your own functions (that only exist
  within your scraper) with **def**
- You can test if something exists, or if it meets partic-
  ular criteria with... **if**
- ...and you can add **elif** and **else** to perform alterna-
  tive actions based on other conditions being met or
  not met.
- Both def and if **should be followed by a colon and
  at least one indented line** that contains code to be
  executed within the function or if the criteria is met.
- Scraperwiki colour-codes some words to help you
  avoid mistakes: red for existing Python statements;
  yellow for strings and purple for numbers.
- **A number in square brackets is probably an index**.
  Indexes specify what position something is in a list.
- **Follow the functions**: find where each function
  is *called*, and then where it's *defined* (it must be
  defined *before* it is called).

- **Read other people's code from bottom to top**: the first parts will be preparing the ingredients - the cooking begins near the end.

From hereon, it's largely about learning new libraries and functions for new problems - but the core way that Python and scraping works should be familiar. The more you scrape, the better you'll get.

Tests

- Try grabbing different indexes in your code. What happens?
- Try changing `.text` to some other property mentioned in the documentation. What happens?
- Explore the documentation to see what else is possible, and explore that in your scraper.

17 Scraper #16: Adapting your third scraper: creating more than one column of data

Here's the code from that live tutorial 3 again:

```
import scraperwiki
import urlparse
import lxml.html
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):

   rows = root.cssselect("table.data tr")
  # selects all <tr> blocks within <table
  class="data">

   for row in rows:

    # Set up our data record - we'll need
   it later

       record = {}

       table_cells = row.cssselect("td")
```

```
if table_cells:

    record['Artist'] = table_-
    cells[0].text

    record['Album'] = table_-
    cells[1].text

    record['Released'] =
    table_cells[2].text

    record['Sales m'] =
    table_cells[4].text

    # Print out the data
    we've gathered

    print record, '------------'

    # Finally, save the
    record to the datastore
    - 'Artist' is our unique
    key

    scraperwiki.sqlite.save(["Artist"],
    record)

# scrape_and_look_for_next_link function: calls
the scrape_table
# function, then hunts for a 'next' link: if one
is found, calls itself again
def scrape_and_look_for_next_link(url):
```

```
html = scraperwiki.scrape(url)

print html

root = lxml.html.fromstring(html)

scrape_table(root)

next_link = root.cssselect("a.next")

print next_link

if next_link:

    next_url = urlparse.urljoin(base_-
    url, next_link[0].attrib.get('href'))

    print next_url

    scrape_and_look_for_next_link(next_-
    url)

# START HERE: define your starting URL - then
# call a function to scrape the first page in
the series.
base_url = 'http://www.madingley.org/uploaded/'
starting_url = urlparse.urljoin(base_url, 'example_-
table_1.html')
scrape_and_look_for_next_link(starting_url)
```

Now that we know how it works, we can adapt it to scrape our horse racing deaths page properly.

Before you look at your code, write down what you need the scraper to do, in steps, so that you're absolutely clear. Here's what we need from the horse deaths page:

1. Grab the page
2. Find the cells
3. There are four columns, each of which needs to be stored separately - how do we distinguish them?

You may have to come back to the list and add to it as you realise extra stages are needed.

Now start at the bottom of the code and follow the functions. As you go, ask yourself whether a piece of code is relevant to your specific problems. Here's the relevant chunk of code:

```
# START HERE: define your starting URL - then
# call a function to scrape the first page in
the series.
base_url = 'http://www.madingley.org/uploaded/'
starting_url = urlparse.urljoin(base_url, 'example_-
table_1.html')
scrape_and_look_for_next_link(starting_url)
```

Firstly, we need to tell it what URL to start with. So you could change the first working line of code to this:

```
base_url = 'http://inmo.ie/6022'
```

The next line...

```
starting_url = urlparse.urljoin(base_url, 'example_-
table_1.html')
```

...joins that variable to another, to form the first URL we're going to scrape - but we know we're only scraping one page, not a series, so we can strip out the code that joins them. In fact, we could simply put:

```
base_url = 'http://inmo.ie/6022'
```

```
starting_url = base_url
```
Or simpler still:
```
starting_url = 'http://inmo.ie/6022'
```
Finally for this chunk of code, we need to update the comments:
```
# START HERE: define your starting URL - then
call a function to scrape it
starting_url = 'http://inmo.ie/6022'
scrape_and_look_for_next_link(starting_url)
```
Now, as it's called, we need to trace up to a second chunk of code: the `scrape_and_look_for_next_link` function:
```
def scrape_and_look_for_next_link(url):

    html = scraperwiki.scrape(url)

    print html

    root = lxml.html.fromstring(html)

    scrape_table(root)
```

We don't need to change any of this before the part where it calls the `scrape_table` function, as it's only really scraping our page into something that's ready to be parsed with that other function. So let's trace up to *that*:
```
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):
```

```
 rows = root.cssselect("table.data tr")
# selects all <tr> blocks within <table
class="data">

 for row in rows:

  # Set up our data record - we'll need
it later

     record = {}

     table_cells = row.cssselect("td")

     if table_cells:

        record['Artist'] = table_-
        cells[0].text

        record['Album'] = table_-
        cells[1].text

         record['Released'] =
        table_cells[2].text

         record['Sales m'] =
        table_cells[4].text

         # Print out the data
        we've gathered

         print record, '------------'
```

```
 # Finally, save the
record to the datastore
- 'Artist' is our unique
key

 scraperwiki.sqlite.save(["Artist"],
record)
```

Now this is where the real customisation takes place. The second line grabs anything in the HTML tags ‹table class="data"› ‹tr› - but our page doesn't have a ‹table class="data"› - it has ‹table class="Trolley"›.

We need to change it, then, to:

```
 rows = root.cssselect("table.Trolley
tr") # selects all ‹tr› blocks within
‹table class="Trolley"›
```

> *i* If we only had one table, because the ‹tr›
> tag only appears in tables, we could simplify
> it further to:
>
> ```
>  rows = root.cssselect("tr") # selects
> all ‹tr› blocks
> ```

The next couple of lines are fine. We still need to then grab each <td> cell in turn, so this line stays as it is:

```
table_cells = row.cssselect("td")
```

But as we hit the code that stores them...

```
 record['Artist']  =  table_-
cells[0].text

 record['Album'] = table_cells[1].text

 record['Released'] = table_-
cells[2].text

 record['Sales m']  =  table_-
cells[4].text
```

...we need to specify and name the cells better. If we look at the HTML behind the page we're scraping we can see that the first ‹td› in each row is the date, so for the line using index position 0 - [0] - we change the name to 'Date'.

We repeat that for the next few lines, adding an extra line for the fourth column (index 3):

```
 record['Date'] = table_cells[0].text

 record['Hospital']  =  table_-
cells[1].text

  record['Region']  =  table_-
cells[2].text

 record['Trolley total'] = table_-
cells[3].text

 record['Ward total'] = table_-
cells[4].text
```

And finally, we need to specify the right entry to be the unique key. Because a date, region, or number could all appear more than once, the best one to choose is the name of the hospital:

```
scraperwiki.sql.save(["Hospital"],
record)
```

We also change `scraperwiki.sqlite.save` to `scraperwiki.sql.save`. Now click **RUN**.

Because all cells are formatted the same, we don't get any errors this time from ‹td› being followed immediately by another tag - but we know if we did we could solve this by swapping `.text` for `.text_content()`

```
record['Hospital'] = table_-
cells[1].text_content()
```

Now, after scraping this page, the code does come *back* to the second part of the `scrape_and_look_for_next_link` function *after* `scrape_table` was called, as shown below:

```
scrape_table(root)

next_link = root.cssselect("a.next")

print next_link

if next_link:

    next_url = urlparse.urljoin(base_-
    url, next_link[0].attrib.get('href'))
```

```
 print next_url

 scrape_and_look_for_next_link(next_-
 url)
```

The second line of that code section above was originally written to find the ‹a class="next" ...› link so that it could run the scraping functions all over again.

But as we're only scraping one page, we don't need any of this, so you can delete it.

Here's the code in full once all of that customisation has been done:

```
#!/usr/bin/env python
import scraperwiki
import urlparse
import lxml.html
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):

  rows = root.cssselect("table.Trolley
 tr") # selects all ‹tr blocks within
 ‹table class="data"

  for row in rows:

  # Set up our data record - we'll need
 it later

    record = {}
```

```
table_cells = row.cssselect("td")

if table_cells:

    record['Date'] = table_-
    cells[0].text

    record['Hospital'] =
    table_cells[1].text

    record['Region'] = table_-
    cells[2].text

    record['Trolley total']
    = table_cells[3].text

    record['Ward total']
    = table_cells[4].text

    # Print out the data
    we've gathered into the
    'record' variable, followed
    by '---'

    print record, '------------'

    # Finally, save the
    record to the datastore
    - 'Hospital' is our
    unique key
```

```
           scraperwiki.sql.save(["Hospital"],
        record)


  # scrape_and_look_for_next_link function: calls
the scrape_table function
  def scrape_and_look_for_next_link(url):

    html = scraperwiki.scrape(url)

    print html

    root = lxml.html.fromstring(html)

    scrape_table(root)


  # START HERE: define your starting URL - then
call a function to scrape it
  starting_url = 'http://inmo.ie/6022'
  scrape_and_look_for_next_link(starting_url)
```

You could simplify this further. For example, we're not
actually using the `urlparse` library now (it was used for the
function creating the initial URL), so we can delete the line
importing it. We could also try to merge the two functions
into one.

You could also add to it. For example, it may be that
each hospital's name also links to a page with more infor-
mation on it (it doesn't, but in other instances it might). We
could grab those links by adding the following lines under
our `if table_cells:` statement (don't forget the indent):

```
        if table_cells:
```

```
table_cellsurls = table_-
cells[1].cssselect("a")

record['HospitalURL']
= table_cellsurls[1].attrib.get('href')
```

See if you can explain what this does by *following the variable...*

By the way, note that we're using the same `.attrib.get('href')` that was used in the previous scraper to fetch the 'next' link - we're just using it in a different place to find a link in a different bit of HTML.

There are other things we can do, which we'll return to. But for now the scraper works, so let's move on to a new problem to learn another key technique.

Recap

This scraper was largely about re-using techniques we'd already learned in adapting a scraper to another page. Here are the key points to take away:

- **Make a list of what your scraper needs to do, step by step**. And update it if you find new stages.
- **Start from the bottom**, and adapt the scraper as it goes through each function
- Change the **starting URL**
- **Change the comments** to reflect the new code

- **Change the code specifying the HTML to be extracted**: there will be different tags, with different classes or ids, and different positions.
- **Change the code labelling each field** in your new dataset
- **Strip out code that is no longer necessary**. You can always test it to see if it's definitely not necessary, and revert to your previous code if you find that removing the lines breaks it.

At the end of our scraper we added a new field for URLs. What if we had such a list and needed to scrape each page? That's our next problem.

Tests

Before that, have a play around with this scraper, and try some of the following:

- Change the URL to another page with a basic table to scrape (such as at UKjockey.com) - how can you adapt the rest of the scraper to work with the slightly different HTML on that page?
- Can you change `.text` to `.text_content()` on other lines? What happens?
- What happens if you choose other indexes?
- What happens if you choose a different unique key?
- As always, browse the documentation for some of these elements and see what other things are possible.

18 Scraper #17: Scraping a list of pages

In Scraper #6 I went through a scraper that was written for The Scotsman newspaper, which used Google Refine to grab free school meals data for Scottish schools. This introduced the idea of structure in URLs.

But using Google Refine is slow and clunky for such a task - we can do that much faster in Scraperwiki, and also introduce a useful technique.

You can see the original scraper code at https://scraperwiki.com/scrapers/ school_meals_scotland/[1] (you'll notice a key difference which I'll mention below) and read more about the context at http://helpmeinvestigate.com/education/2012/01/free-school-meals-in-scottish-primary-schools-data-visualisation/[2].

```
# Data stored on individual pages not accessible
from one single page
# typical URL is http://www.ltscotland.org.uk/scottishschoolso
# Need to cycle through a list of those codes
#If you want to understand this scraper - start
at the bottom where it says 'base_url' (line 52 or
so)
import scraperwiki
import lxml.html
```

---

[1]https://scraperwiki.com/scrapers/free_school_meals_scotland/
[2]http://helpmeinvestigate.com/education/2012/01/free-school-meals-in-scottish-primary-schools-data-visualisation/

```
    #Create a function called 'scrape_table' which
is called in the function 'scrape_page' below
    #The 'scrape_page' function also passed the
contents of the page to this function as 'root'
    def scrape_table(root):
     #Use cssselect to find the contents of a
particular HTML tag, and put it in a new object
'rows'
     #there's more than one table, so we need to
specify the class="destinations", represented by
the full stop

     rows = root.cssselect("table.destinations
     tr")

      for row in rows:

          #Create a new empty record

          record = {}

          #Assign the contents of <td>
          to a new object called 'table_-
          cells'

          table_cells = row.cssselect("td")

          #If there's anything

          if table_cells:
```

```
                 #Put the contents of
                 the first <td> into a
                 record  in  the  column
                 'FSM'

                 record['FSM'] = table_-
                 cells[0].text

                  #this  takes  the  ID
                 number, which has been
                 named item in the for
                 loop below

                 record['ID'] = item

                 print record, '------------'

                  #Save  in  the  SQLite
                 database, with the ID
                 code to be used as the
                 unique reference

                 scraperwiki.sqlite.save(["ID"],
                 record)

    #this  creates  a  new  function  and  (re)names
whatever parameter is passed to it - i.e. 'next_-
link' below - as 'url'
    def scrape_page(url):

        #now 'url' is scraped with the scraperwiki
        library imported above, and the contents
        put into a new object, 'html'
```

```
        html = scraperwiki.scrape(url)

        print html

         #now  we  use  the  lxml.html  function
      imported  above  to  convert  'html'  into
      a new object, 'root'

         root = lxml.html.fromstring(html)

         #now we call another function on root,
      which we write - above

         scrape_table(root)

    #START HERE: This is the part of the URL which
all our pages share
    base_url = 'http://www.ltscotland.org.uk/scottishschoolsonline
    #And these are the numbers which we need to
complete that URL to make each individual URL
    #This list has been compiled using the =JOIN
formula in Google Docs on a column of school codes
    schoolIDs =['5237521','5244439','5245044','5237629','5237823'
...]
    #go through the schoolIDs list above, and for
each ID...
    for item in schoolIDs:

        #show it in the console

        print item
```

```
 #create a URL called 'next_link' which
adds that ID to the end of the base_url
variable

 next_link = base_url+item

  #pass that new concatenated URL to
a function, 'scrape_page', which is
scripted above

 scrape_page(next_link)
```

One line in the code above needs clarifying before we go any further: *=['5237521','5244439','5245044','5237629','5237823','5234026', ...]*

The "..." at the end is not in the original code. For the purposes of this illustration, it just indicates that there is more in the original code (if we were to copy the whole list from the script it would take up quite a few pages alone).

You'll note that this scraper is also adapted from the third live tutorial on Scraperwiki, so you should be able to follow most of the code. The main difference is that instead of adding a 'next' link to the base URL, it adds an ID code. The key thing to note here is how it uses a for loop to do this:

```
for item in schoolIDs:

   print item

   next_link = base_url+item
```

```
scrape_page(next_link)
```

You'll remember that **for** goes through a list, and does something with each item in turn. This is called '**iterating**'; the **for loop** *iterates* through a *list*. As always, search for 'for loops Python' to find out more if that helps, and read more than one resource until you find it starts to make sense.

In this case, our list is schoolIDs, which we created a few lines earlier:

```
schoolIDs =['5237521','5244439','5245044','5237629','5237823'
```

Note that these numbers are placed in quotation marks to indicate that they are **strings**. Why are they strings? Because they will be used in a URL, not as part of a calculation. (once again: the "..." above is used to indicate another few hundred items in this list - we won't show them all here)

So the first time the for loop runs, it:

1. takes the first item in the list (5237521)
2. prints it
3. creates a variable called `next_link`, which adds that (with the + operator) item to the end of the variable `base_url` (initialised a few lines earlier)
4. and runs the function `scrape_page` with that new variable
5. Once that function finishes with that URL, the *for loop* runs again for the second item in the list, and so on.

The rest of the script should be familiar enough to understand: scrape_page turns the page into an object that is then used by another function - scrape_table - to extract particular parts of HTML. In other words, exactly the same as the last couple of scrapers.

There's only one line which warrants particular explanation:

```
if table_cells:

    record['FSM'] = table_-
    cells[0].text

    record['ID'] = item

    print record, '------------'

    scraperwiki.sqlite.save(["ID"],
    record)
```

Here you'll see that one of the fields stored in the data uses an object called item. What is this? Search in the code and you'll see it's the name of the variable created when the for loop iterates through the list of school ID codes: in other word it's the ID code of the particular page that's being scraped.

As an ID code is unique, it's a good choice for our unique key - and also allows us to match up our data with other data later if we need to.

It also demonstrates that the data you're storing doesn't need to come from within the page itself.

Creating the list of codes

Of course, to use a scraper like this you need a list of codes to start with. We've already covered how we found a spreadsheet of the codes earlier, in our chapter on structures in URLs and using Google Refine for scraping. But making a list from that spreadsheet is a separate process. Here's how:

If your codes are all in one column, you can use the =JOIN function in Google Docs or Excel to join them all together into one list in one cell.

If your codes were in cells A2 to A300, for example, you would need to write the following formula in any other cell:

```
=JOIN("', '", A2:A300)
```

(Type this yourself rather than copying and pasting it, as pasted quotation marks often cause problems by not being 'straight' quotation marks)

The two parameters for the JOIN function are: a delimiter, and the range of cells you want to join. The delimiter is what goes between each item being joined, and in this case we want a single quotation mark, a comma, and another single quotation mark, to produce a result where the first two values look like this (note what's between them):

```
1234567', '7654321
```

The comma separates each item, and the single quotation mark identifies it as a string.

Once we've done that, we'll have a list which is OK to copy and paste into Python *apart from the beginning and end*, which will be missing single quotation marks (because

the delimiter only goes between each item).

So you can either add those manually or write another formula like so (replace B2 with the cell reference of your JOIN formula):

```
="'"&B2&"'"
```

(…this does the following: put a single quotation mark, then the contents of B2, then another single quotation mark.)

You now have a list of those codes ready to copy and paste into your Scraperwiki scraper.

Recap

This scraper introduced **lists** as a starting point for a scraper. Here are the key points:

- If you have a column of elements that can be used to complete URLs you need to scrape, **create a list using =JOIN** in Google Docs or Excel.
- Use a **for loop** to **iterate** through a list, using each item to create a URL that you then scrape
- Use the + operator to **join two parts of a URL** into one
- If a number is being used as part of a URL, it needs to be **formatted as a string** with quotation marks.
- You can **store elements from outside your HTML page** in the data, as long as it's stored and named elsewhere in your code.

Tests

- Try using the =JOIN function some more in Google Docs to join the contents of a row or column of cells. Search for documentation on the function, and make sure you're confident with using it - and can sort out any missing formatting before and after the resulting string.
- Search for guidance on lists in Python - how they can be generated, looped through, and accessed. Think of other ways they might be used in scrapers.

19 Scraper #18: Scraping a page - and the pages linked (badly) from it

NEWS POLITICS

Home | World | UK | England | N.Ireland | Scotland | Wales | Business | **Politics** | Health | Education | Sci/En

12 March 2014 Last updated at 01:39   Share 📘 🐦 ✉ 🖨

David Cameron's prime questioners

By Ed Lowther
Political reporter, BBC News

This article by Ed Lowther about who asked the most questions in Parliament was written based on data he scraped using techniques from *Scraping for Journalists*

At the end of Scraper 16 (the hospital Trolley Watch page) we touched on the ability to grab the links from the HTML for details on each hospital.

Links like these open up a whole new level of data that we could be grabbing - and this scraper will show how to do that.

In addition, in the case we're going to be dealing with, the links are also badly written, which allows us to tackle another key skill in writing scrapers: tackling bad HTML.

Here's one I prepared earlier which scrapes a page detailing ice rinks:

```python
#!/usr/bin/env python
#import the libraries containing our functions
import scraperwiki
import lxml.html
import urllib
# scrape_table function: gets passed an individual
page to scrape
def scrape_table(root):

    rows = root.cssselect("table#AutoNumber2
   tr") # selects all <tr blocks in <table
   id="AutoNumber2"

    for row in rows[0:-1]:

        # Set up our data record -
       we'll need it later

        record = {}

        table_cells = row.cssselect("td")

        #if there's anything in the
       table_cells variable (a list)
```

```
if table_cells:

    #grab the text contents
    of the first and put in
    'record' under the key
    'Rink':

    record['Rink'] = table_-
    cells[0].text_content()

    #grab the <a> tags in
    the first:

     rinkurls  =  table_-
    cells[0].cssselect('a')

    #if there are any <a>
    tags:

    if rinkurls:

        #grab the first
        href=" attribute
        (the URL of the
        link)  and  put
        into 'rinklink'

        rinklink = rinkurls[0].attrib.get('href')

         #combine with
        base  url  and
        scrape into 'rinkhtml'
```

```
rinkhtml = scraperwiki.scrape('http://www.eiha

 #convert into
lxml object and
put into 'rinkroot'

 rinkroot = lxml.html.fromstring(rinkhtml)

 #grab all the
<td><a> tags from
'rinkroot' and
put into 'rinktds'

 rinktds = rinkroot.cssselect('td
a')

 #print a string
followed by the
3rd href=" attribute
of that list of
<td>s:

  print  "rink
url from linked
page:", rinktds[2].attrib.get('href')

  #store   that
value in 'record'
under key 'rinklink'

 record['rinklink']
= rinktds[2].attrib.get('href')
```

```
  #these lines repeat
the process earlier using
the <td> tags from the
main page <tr>s

  record['Address'] =
table_cells[1].text_content()

 #in some rows there is
no table_cells[2].text_-
content(), so this tests
the length first

 if len(table_cells[2].text_-
content())>0:

     record['Area']
     = table_cells[2].text_-
     content()

 record['County'] = table_-
cells[3].text_content()

 record['Phone'] = table_-
cells[4].text_content()

 #grab all the <a> tags
in these two <td>s and
put in a list variable

 showteamsurls = table_-
cells[5].cssselect('a')
```

```
mapurls = table_cells[6].cssselect('a')

 #test the length of
the list before storing
anything - some have
no links

 if len(mapurls)>0:

    record['Maplink']
    = mapurls[0].attrib.get('href')

 else:

    record['Maplink']
    = "NO MAP LINK"

 if len(showteamsurls)>0:

    record['teamslink']
    = showteamsurls[0].attrib.get('href')

 else:

    record['teamslink']
    = "NO MAP LINK"

 # Print out the data
we've gathered

 print record, '------------'
```

```
                # Finally, save the
            record to the datastore
            - 'Rink' is our unique
            key

             scraperwiki.sql.save(["Rink"],
            record)

    # scrape_and_look_for_next_link function: scrapes
page,  converts  to  lxml  object  then  calls  the
scrape_table function
    def scrape_and_look_for_next_link(url):

        html = scraperwiki.scrape(url)

        print html

        root = lxml.html.fromstring(html)

        scrape_table(root)

    # START HERE: define your starting URL - then
call a function to scrape it
    starting_url = 'http://www.eiharec.co.uk/rink_-
list.php'
    scrape_and_look_for_next_link(starting_url)
```

As always, take some time to look through it using
the techniques we've explored: *following the variables and
functions* (writing them down helps); *reading from right
to left* and from *bottom to top*; and looking at the colour
coding.

As this is based on the live tutorials and our previous scrapers, most of it should be familiar. Here's one section I've added:

```
#grab the <a> tags in
the first:

rinkurls = table_-
cells[0].cssselect('a')

#if there are any <a>
tags:

if rinkurls:

    #grab the first
    href=" attribute
    (the URL of the
    link)  and  put
    into 'rinklink'

    rinklink = rinkurls[0].attrib.get('href')

     #combine with
    base  url  and
    scrape into 'rinkhtml'

    rinkhtml = scraperwiki.scrape('http://www.eiha

     #convert into
    lxml object and
    put into 'rinkroot'
```

```
rinkroot = lxml.html.fromstring(rinkhtml)

 #grab all the
<td><a> tags from
'rinkroot' and
put into 'rinktds'

 rinktds = rinkroot.cssselect('td
a')

 #print a string
followed by the
3rd href=" attribute
of that list of
<td>s:

  print   "rink
url from linked
page:", rinktds[2].attrib.get('href')

   #store   that
value in 'record'
under key 'rinklink'

 record['rinklink']
= rinktds[2].attrib.get('href')
```

The comments explain what it does: in short, grab the
links (which is only a relative, i.e. partial, link so we need to
add it to the base URL to complete it), scrape it, turn it into
an lxml object so we can extract the <td><a> tag contents

from it, and save the href="" attribute (i.e. the URL) of the third `<a>` tag in the resulting list.

It's looking for the third `<a>` tag because the pages containing each rink's details have the rink's website URL there (although some have no link at all, in which case the third link is the map URL).

Using ranges to avoid errors

There are some other lines in this scraper which warrant further explanation. Here's the first:

```
for row in rows[0:-1]:
```

We've seen `for` loops before, and we've seen indexes before, but this one is a little different. `[0:-1]` is a **range** - it basically means *from index 0 to just before index -1*.

Put another way: when this `for` loop runs, it's not going to run through every item in the list, but rather every item up to but *not including* the last item.

We could adapt this to loop through a different range like so:

```
for row in rows[0:5]:
```

This would start at index 0 in that list and end at index 4 (confusingly, index 5 is used to indicate the first item it's *not* going to loop through).

Why do we need this? Change the line to remove the range, **Run** the scraper and see what happens:

```
for row in rows:
```

This generates an error - on the *last row* - because there is no second `<td>` tag on that row (it's a sort of footer at the

end of the table, with no data).

Using a range allows us to say 'ignore that last row', or 'ignore the first row' or any other instruction.

Using `len` to test lists

There are three parts of the code when an `if` test is used with a function called `len`.

`len` is short for 'length' - it can be used to measure the length of a string (i.e. how many characters there are, including spaces) or the length of a list (i.e. how many items it contains).

The function is pretty straightforward: put what you want measured in parentheses immediately afterwards. (In fact, you can use the same function in spreadsheets too.)

It is particularly useful for avoiding errors when we have no data.

On the first occasion `len` is used to measure a string: the result of `table_cells[2].text_content()`:

```
if len(table_cells[2].text_content())>0:
```

The value of that `table_cells[2].text_content()` is only stored *if* that condition (that its length if greater than 0) is met.

On the other two occasions that `len` is used to measure a list - both `mapurls` and `showteamsurls` are lists (if you're ever in any doubt just add a `print` command to show you the variable. If it's a list it will use square brackets).

The result of using `len` to measure that list will be a number. This `if` test runs if that number is higher than 0:

```
if len(mapurls)>0:
```

In other words, if the list has *at least one item* then we run the next, indented line:

```
record['Maplink'] = mapurls[0].attrib.get('href')
```

Otherwise - that is, if it *is an empty list* - we store a string of text in that record's 'Maplink' key instead:

```
record['Maplink'] = "NO MAP LINK"
```

The same process is repeated for the list `showteamsurls`.

If you want to see why we need those lines, try removing one of the `if` and `else` tests and moving the first indented line back in line with the others (deleting the second). You should get an error when the scraper hits the first empty list.

Problems with URLs

In a [previous scraper](#)[a] we had a problem where some links gave us an `Error code 400`. In these situations a good technique is to compare the ones that work with those that don't: what's different?

In this case the lines that worked were all single word names. The others - generating the error code - had spaces in them.

URLs don't have spaces. Have you ever tried to type a space in a browser's address bar? The browser will replace it with something else - normally `%20` or the + sign (If there's a question mark in the URL, the + replaces spaces after it. Before the question mark, or in URLs without one, a space is indicated by `%20`).

But we don't have a browser correcting our mistakes. We just have Python, and we need to tell it to correct the mistakes first.

There is more than one way to correct the problem. Here's the first:

```
        testingreplace        =
'http://www.horsedeathwatch.com/'+table_-
cellsurls[0].attrib.get('href')

 print testingreplace.replace(" ", "%20")

  horselink  =  testingreplace.replace("  ",
"%20")
```

This needed to be written in the code instead of the line

```
horselink = 'http://www.horsedeathwatch.com/'+tab
cellsurls[0].attrib.get('href')
```

The second line prints the results of using the `.replace` method. This replaces something in our variable with something else - the specifics are detailed in the parameters: replace any spaces - " " - with "%20".

The third line does the same thing again, but instead of printing it, puts the result into a new variable, `horselink`. That variable name is already used in the subsequent code, so we don't need to change that.

Methods for changing text

We've encountered methods for doing things with text before - particularly `.text` for grabbing the text

between two tags and `.text_content()` for grabbing all text within a particular tag - but we've not really gone into detail about methods generally.

A **method** is like a function: it does something. You can generally spot a method because it is preceded by a full stop (period), which attaches it to an object (normally a variable). The key difference is:

- A function works with the parameters inside parentheses
- A method works with the object it's attached to with the full stop (period)
- A method uses any parameters to work with that object

So our `.replace` method takes the *object on the other side of that full stop*, and *replaces* something in it.

The parentheses contain *the parameters that specify how it does that*: first, what it's replacing, then, after a comma, a second parameter: what it's replacing it with.

Other useful methods for dealing with text (strings) include:

- `.capitalize()` - turns all characters into upper case
- `.lower()` - turns all characters into lower case
- `.lstrip([chars])` - strips out any leading characters specified in parentheses. If the parentheses are left empty, then any empty spaces are stripped out.

You can also use methods to test if something is true or false, e.g.:

- `.isalnum()` - returns 'true' if the string consists entirely of numbers. Otherwise, returns 'false'
- `.islower()` - returns 'true' if the string consists entirely of lower case characters. Otherwise, returns 'false'
- `.isspace()` - returns 'true' if the string consists entirely of spaces (and there's at least one). Otherwise, returns 'false'

True and false can be used with if statements like so:

```
testing = " "

if testing.isspace():

        print "spacey"

else:

        print "not spacey"
```

Those are just a few to give you a flavour. The documentation for Python at http://docs.python.org/library/stdtypes.html#string-methods[b] gives a comprehensive list of methods for strings (text). The same page also contains information on other methods, including those for working with numbers (integers and floats).

---

[b]https://scraperwiki.com/scrapers/horsedeaths_deeplinkscraping_broken/

A second way of fixing bad URLs

There are other ways of fixing a URL problem, however. One is to use a function from a library which is designed to deal with URLs. And there's one in particular called **urllib**:

First you need to import `urllib` in the first lines of code:

```
import urllib
```

Then you can use it later on with the `.urlopen` function:

```
    #Use   the
urllib.urlopen
function   to
open the URL
'properly' so
it works

 horselinkurl
=
```

```
urllib.urlopen('http://www.horsed
cellsurls[0].attrib.get('href')).
```

In the code above, the `.urlopen` function works with the URL that caused problems originally. It deals with those problematic spaces much better than our original code did: the spaces don't cause it a problem.

We add the `.read()` method at the end to 'read' the contents of that opened URL into the new variable `horselinkurl`.

This means that `horselinkurl` basically contains the whole of the webpage at that URL, so we don't need to use `scraperwiki.scrape` to do the same thing, and can skip straight to `lxml.html.fromstring` to turn that string into an lxml object so we can extract data from it.

You can read more about the `urllib` library in the documentation: http://docs.python.org/library/urllib.html[a]

---

[a]http://docs.python.org/library/urllib.html

Other workarounds

These are just two options - the more experience you gain with solving scraping problems, the more solutions you'll discover. Broadly, this problem of unwelcome characters in URLs is called 'escaping'. You can find a series of '**escape**

codes' that are used for this at http://www.december.com/html/spec/esccode

You also have another way of scraping all the links that Scraper 16 gathered: download the data and open in a Google spreadsheet. Google spreadsheets have a =JOIN function which can be used to join the values in a series of cells together to create a list for a *new scraper* to loop through. =JOIN has two parameters: a delimiter (i.e. a character that you want to insert between each value - most likely a comma) and the range of cells with the values to be joined. So the formula =JOIN(",",A2:A200) will join all the values in the cells A2 to A200, with a comma between each one.

Sometimes this will be faster and/or simpler. In this example, however, it has one important limitation: the list does not get updated when the page does, so if you're planning to run the scraper regularly, it's not likely to return anything new on the second and third runs.

If your list of links is likely to change, then, you're going to need to incorporate grabbing those links into the same scraper that scrapes their contents.

Recap

This chapter deals with two problems in scraping: dealing with bad URLs, and scraping a further level of linked pages. Here are the key points to remember:

---

[1]http://www.december.com/html/spec/esccodes.html

- If some pieces of data scrape correctly, and others don't, **look for the differences between their sources**. In this case it was spaces in links, but it may be the destination pages or something else.
- **URLs shouldn't have spaces in them**.
- If they do, you can **fix it with .replace or with urllib.urlopen()**
- You can also download a list of links and **make a list of them** to loop through, using =JOIN
- **Methods** are like functions, but they **work on objects** rather than parameters. Spot them by the *full stop (period) preceding it*, which attaches it to an object (normally a variable).
- `len` will give you the length of a string - or the number of items in a list.
- Using `len` with `if` allows you to only store data if it exists - that is, if its length is greater than zero.
- There are a range of **methods available for changing**, **extracting or testing text and numbers**.
- Resolving problematic characters in URLs is called **escaping**: each character has an *escape code* which represents that character, i.e. %20 represents a space.

Scraper tip: a checklist for understanding someone else's code

We've covered a lot of techniques for looking for particular types of code, such as variables and functions, but the following checklist is useful if you're trying to trace someone else's code:

1. **Search for the word elsewhere in the code**:

- If it appears before an = sign, then it's most likely a **variable** initialised at that point.
- If it appears after `def`, then it's a **function** created at that point (and you can follow the indented lines after that to see what it does).
- If it appears after `import` then it's a **library** (and you can search for documentation to find out what it does)
- If it's *purple* in the code then it's probably a Python **statement** - search Python documentation to find out more about what it does
- If it is preceded by another word and joined by a full stop (period) then search for *that* word:

    - if that word appears after `import` then it's a library, and the word joined to it that you're trying to figure out is probably a **function from that library**. Search

the library documentation to find out more.

- If that word is a *variable* (initialised with =), then the word you're trying to figure out is probably a **method** used to find out something about that variable. Search Python documentation to find out more.

Some other options:

1. **Search for the word in documentation**. Rather than go through the steps above, you might skip straight to searching for the word in the documentation for any libraries used in that scraper (with **import**). You can also search Python's own documentation for the word.
2. **Search for the word in the code of the page or document being scraped**. This is the best place to look if you're looking for a string (coloured red and in quotation marks). If you do find it there then it's likely the code is specifying it because that's what it's looking for.

20 Scraper #19: Scraping scattered data from multiple websites that share the same CMS

There are a couple of challenges which are worth covering before we move on from the basic scraper libraries. So far we have dealt with pages where the data is encoded in the HTML consistently: i.e. all the data is contained in `<tr>` tags, or all the data is in `<p>` tags.

In those cases, scraping that data has largely consisted

of putting them all into a list and then extracting items based on their position in that list, e.g. the first item is the age; the second item is the name; and so on.

If the data is scattered around, however - a name in a `<h4>` tag; a biography in the second `<p>` tag, and so on - then you'll need to jump around the page before you store the data.

This chapter shows you how.

Finding websites using the same content management system (CMS)

The example I'm using for this is a collection of websites for clinical commissioning groups (CCGs), health bodies in the UK which are being given new powers to commission services.

Each site has a list of board members, roles and biographies. Rather than put these into a spreadsheet manually, I wanted to automate the process. I'm lazy like that.

The process of compiling the list of websites is quite interesting itself. If you have a situation where you want to scrape a collection of websites and they seem quite similar - or are likely to have been designed by the same company or use the same content management system (CMS) - then there's a chance the data might be encoded in the same way too.

That was the case here. Brent[1] and Ealing[2] CCGs' 'About Us' pages shared a similar design - and URL. I used the following advanced search on Google[3] to look for webpages that shared the same section of URL and mentioned 'CCG':

```
inurl:about-us/board.aspx CCG
```

...and indeed, there were seven of them.

Writing the scraper: looking at HTML structure

The HTML containing data on each board member looks like this:

```
<div class="block size3of4">
<h4>Philip Portwood<span>Board Member</span></h4>
<p><p>Philip was an East Acton Councillor for
twenty two years, being elected in six successive
elections. He has served longer as an East Acton
Councillor than any other person in the history of
local government in Acton (which began in 1895).
Philip was the Millennium Mayor of the Borough. He
is a Director of the Acton Community Forum, and a
```

[1] http://www.brentccg.nhs.uk/

[2] http://www.ealingccg.nhs.uk

[3] https://www.google.co.uk/search?q=ccg+nhs+uk+board&channel=linkdoctor&safe=on#hl=en&safe=active&sclient=psy-ab&q=inurl:about-us%2Fboard.aspx+CCG&oq=inurl:about-us%2Fboard.aspx+CCG&gs_l=serp.3...8135.8741.2.9010.4.4.0.0.0.0.92.351.4.4.0...0.0...1c.1.xmE9ViOJ14U&pbx=1&bav=on.2,or.r_gc.r_pw.r_cp.r_qf.&fp=4a762d82895734c9&bpcl=35466521&biw=1058&bih=531

```
Trustee of both the Acton (Middlesex) Charities and
the Imperial College Healthcare Charity.</p></p>
    </div>
```

We have a name contained within ‹h4› tags, a position within ‹span› tags, and a biography within a second ‹p› tag. The whole is contained within ‹div class="block size3of4"› - and in fact if we look outwards there's also an ‹article› tag containing all this data plus an image URL to boot.

A first run at the scraper might look like this:

```
import scraperwiki
import urlparse
import lxml.html
# scrape_divs function: gets passed an individual
page to scrape
def scrape_divs(url):

    html = scraperwiki.scrape(url)

    print html

    root = lxml.html.fromstring(html)

    #line below selects all <div class="block
    size3of4"> - note that because there is
    a space in the value of the div class,
    we need to put it in inverted commas as
    a string

    rows = root.cssselect("div.'block size3of4'")
```

```
for row in rows:

     # Set up our data record -
    we'll need it later

     print row

     record = {}

     #grab all <h4> tags within our
    <div>

     h4s = row.cssselect("h4")

     #put the text from the first
    <h4> tags into variable membername

     membername = h4s[0].text

     #repeat process for <h4><span>

     spans = row.cssselect("h4 span")

     membertitle = spans[0].text

     #repeat process for <p> tags

     ps = row.cssselect("p")

     #this line puts the contents
    of the last <p> tag by using
    [-1]
```

```
         memberbiog = ps[-1].text_-
    content()

     record['URL'] = url

     record['Name'] = membername

     record['Title'] = membertitle

     record['Description'] = memberbiog

     print record, '------------'

     # Finally, save the record to
    the datastore - 'Name' is our
    unique key

     scraperwiki.sqlite.save(["Name"],
    record)

   #list of URLs with similar CMS compiled with this
advanced search on Google: inurl:about-us/board.aspx
CCG
   ccglist = ['www.brentccg.nhs.uk/', 'www.ealingccg.nhs.uk/',
'www.hounslowccg.nhs.uk/', 'www.westlondonccg.nhs.uk/',
'www.centrallondonccg.nhs.uk/', 'www.harrowccg.nhs.uk/',
'www.hammersmithfulhamccg.nhs.uk/']
   #loop through the list and for each one, convert
into the full URL and run the scrape_divs function
created earlier
   for ccg in ccglist:
```

```
scrape_divs('http://'+ccg+'about-us/board.aspx')
```

You can find this scraper at https://scraperwiki.com/scrapers/ccgboards_examplebroken/[4]. The comments in the code above explain most of what needs to be understood about it, but here it is in full, reading from the bottom up:

- The last two lines of code loop through a list of URLs and run them through a function created earlier on.
- That function grabs the contents of each of the `<div class="block size3of4">` tags and puts it in a list.
- Then it loops through that list, and grabs the contents of any `<h4>` tags and puts those in a list - then grabs the first item in that list and stores it as 'member-name'.
- It grabs the contents of any `<h4><span>` tags and puts those in a list - then grabs the first item in that list and stores it as 'membertitle'.
- It grabs the contents of any `<p>` tags and puts those in a list - then grabs the *last* item in that list and stores it as 'memberbiog'.
- All three are then stored in a **dictionary** variable called `record`, and that is stored in the Scraperwiki datastore, with the 'Name' field used as the unique key.

If you click **RUN** on that scraper it will work for 17 records before hitting an error: `list index out of range`. In other words, we're asking for an index that doesn't exist.

Here's more detail on the line generating the error:

```
Line 20 - membername = h4s[0].text -- scrape_-
divs((url='http://www.brentccg.nhs.uk/about-us/board.aspx'))
```

So it's the first ‹h4› tag that doesn't exist.

If we look at the data we can see that we've grabbed all the names on that page so what's the problem?

Work through the scraper to find out what it's doing. It's looking for the ‹h4› tag in the tag ‹div class="block size3of4"› and after that last name there's another one of those - *without any ‹h4› tag in it*:

```
<div class="block size3of4">
 <ul class="clearfix nobox"><li class="parent"><a
href="/" title="Home">Home</a></li><li class="parent"><a
href="/contact-us.aspx">Contact Us</a></li><li class="parent"><a
href="/about-us.aspx">About Us</a></li><li class="parent"><a
href="/news.aspx">News</a></li><li class="parent"><a
href="/publications-and-policies.aspx">Publications
and Policies</a></li></ul>
 </div>
```

How do we deal with this? Well, we *could* ask it only to grab the data *if* it exists...

Using if statements to avoid errors when data doesn't exist

If our error is generated by the data not existing, we need to adapt the scraper to *only* grab the contents when the data exists.

You might imagine that the scraper should already do that - but code is purely logical: if you tell it to get something, and that something doesn't exist, it just shouts 'error!'

This isn't such a bad thing: we need computers to tell us when they hit something we didn't expect. In some cases that's because the data is formatted differently, and we need to adapt the scraper to grab that data too. But in this case, we need to give the scraper permission to move on if the data doesn't exist.

We can do either with an **if statement**.

We've already talked about **if statements** in the previous chapter, in relation to testing if a something is 'true' (i.e. a condition is met, such as a variable being all numbers or spaces). But we first introduced **if statements** in Chapter 16, as a way of testing if something exists. And that's how we need to use them here.

Instead of the line:

```
membername = h4s[0].text
```

...we need to write the following:

```
 #if there are any, grab the
first and put it in membername
variable

 if h4s:

    membername = h4s[0].text
```

If you click **RUN** now your scraper will be OK... until it hits that same error - `IndexError: list index out of range` - this time with the `<span>` tag:

`Line 25 - membertitle = spans[0].text -- scrape_divs((url='http://www.brentccg.nhs.uk/about-us/board.aspx`

So, off we go again, this time to replace this line:

```
membertitle = spans[0].text
```

...with these:

```
if spans:

    membertitle = spans[0].text
```

And, yes, the next line will also generate the same problem, so that:

```
    memberbiog = ps[-1].text_-
content()
```

...is replaced with this:

```
if ps[-1]:

    memberbiog = ps[-1].text_-
    content()
```

Now if you **RUN** that, it works OK... for a few seconds. Then we hit a new error:

`UnboundLocalError: local variable 'membername' referenced before assignment`

The variable that doesn't exist

The first thing we recognise is the variable 'membername'. It's being "referenced before assignment" apparently. But what does that mean?

It means it's being called upon before anything has been assigned to it. In other words, at this point as the script runs, the variable *does not exist.*

But how can that be? We've just stored 50 records using that variable.

To help answer that, *trace the code as it runs* on the piece of data being scraped.

Looking in the **Data** tab below the scraper, we can see that the last piece of data to be successfully scraped was the details for Sarah Cuthbert on this page[5]. Sarah is the last person listed on that page.

We can also read in the **Console** just above the error message the following information: `record['Name'] = membername -- scrape_divs((url='http://www.westlondonccg.nhs.uk/a`

That's a different URL, so we now know that our error was generated *on the first piece of data on this page.*

Now, to do a bit of detective work on that page. Look at the raw HTML by right-clicking and selecting 'View source' (or use the CTRL+U shortcut in Firefox, or a plugin like Firebug).

Now try to do what the scraper is doing:

- We start with the line `rows = root.cssselect("div.'block size3of4'")`. Is there a `<div class="block size3of4">`

---

[5] http://www.hounslowccg.nhs.uk/about-us/board.aspx

in that HTML? Yes. So that, and all others, will be put in the `rows` variable.

- Next, the code loops through a list of those: `for row in rows:`
- Then we move to this line:`h4s = row.cssselect("h4")`. For the first of those `<div class="block size3of4">` sections, then, it selects any `<h4>` tags and puts them in a variable called `h4s`. Are there any `<h4>` tags within that first `<div class="block size3of4">`? No, there are not, so our `h4s` variable has nothing in it.
- The next line: `if h4s:` is a test that is not met. There is no `h4s`, so the next line:
- `membername = h4s[0].text` is not executed. So *the variable `membername` is never initialised.*
- That means that by the time we get to the line that says `record['Name'] = membername` there is no `membername` and the computer is asking "What is that?"

We need to make sure that `membername` exists *whether or not there is any data to put in it.*

Initialising an empty variable

This is done by initialising - creating - the variable before that part of the script runs.

So, in the line before `h4s = row.cssselect("h4")`, add this line of code:

```
membername = ""
```

...And, as we know this is likely to happen with the other variables too, you may as well add another couple of lines to cover those too:

```
membertitle = ""

memberbiog = ""
```

Now, then, when the code runs, all three variables will be initialised before we hit the **if statements** that test if there's something to put into them. If there is nothing, no problem, that part of the data will be filled with nothing - "" - rather than falling over because you're asking it to find a variable that was never initialised.

If there *is* something, however, the nothing will be replaced with that.

Now the scraper works for all seven websites. It doesn't fall over when it doesn't find the tags in the place it expects them; and it doesn't fall over if some of the data is missing. You can see the final working scraper here[6] if you're interested.

This technique is quite common in scrapers: if a website has switched from one content management system to another, or one user uses it in a slightly different way, then you can adapt the scraper to accommodate that.

---

[6]https://scraperwiki.com/scrapers/ccgboards/

Recap

This chapter focused on some error-handling techniques and ways to adapt your scraper to HTML that may vary a little from page to page. Here are the key points:

- Use the `inurl:` operator on Google searches to find sites that might share the same content management system and therefore HTML structure for data.
- If your scraper is falling over when it hits something unexpected, **look at the code in the page at the point when it had the problem** and try to rewrite your code to accommodate what it is that's different.
- Use **if statements to only run parts of your scraper when data fits a particular pattern** so that it doesn't fall over if it doesn't.
- **Initialise variables before the if statements** so that you're not relying on the if statement to initialise them. Obviously leave them empty or initialise them as = `"no data"` so you can identify those parts of your data.
- **Put yourself in the position of the scraper as it goes through a page** in order to understand why it's having a problem.

Tests

- You can actually avoid all the if statements by not using `<div class="block size3of4">` as your general 'container'. See if you can alter the scraper to use

a different tag containing all your details - and can
you remove the if statements and it still works?

- How can you grab the URL of the board member's
photo as well?
- How can you add that relative URL to the base URL
so you have a full one?

21 Scraper #20: Automating database searches (forms)

Quite often you will want to scrape data which is only accessible through searching a database. Depending on the way that the search is set up, here are some things to consider:

- Can you get everything in the database by leaving all fields empty (or sometimes using an asterisk: *) and clicking 'Search'?
- If not, does the search require certain fields to be filled with information which you could supply from a list, e.g. regions, postcodes, countries etc? Or does it only use drop-down lists and tick boxes?

- Do the results have their own URL which works when someone goes directly to it without a search (you can use another browser to test this yourself)?
- Is there another way to access the data, e.g. a browse facility, API or Freedom of Information request?

Checking the above can help you more quickly identify the best way to access the data. For example: an empty search on the Chief Fire Officers Association's Enforcement Register produces results at this URL:

http://www.cfoa.org.uk/11823?pv=search&premises_type_-id=&premise_id=&frs_id=&organisation_name=&address=&address_-postcode=&status_id=[1]

Likewise, once upon a time, if you wanted to scrape the Social Care Register an empty search at the now-dead GSCC website - http://www.gscc.org.uk/registerSearch.php[2] (now no longer live) - would generate a whopping 110,000 results. The results page URL[](http://www.gscc.org.uk/registerSearch.php# did not work in another browser where the search had not been conducted.

But if you clicked to the *second* page of results you would actually find a URL that worked in another browser: http://www.gscc.org.uk/registerSearch.php?o=11#results

The GSCC's replacement is the Heath and Care Professions Council. A search for social workers called 'Wilson' on that now produces a URL like this: http://www.hpc-

---

[1]http://www.cfoa.org.uk/11823?pv=search&premises_type_id=&premise_id=&frs_id=&organisation_name=&address=&address_postcode=&status_id=

[2]http://www.gscc.org.uk/registerSearch.php

uk.org/search-results/?searchOption=1&search=wilson&profession=SW³

To understand more about all of these URLs, we need to take a detour...

Understanding URLs: queries and parameters

When you conduct searches on a website, information about your search is sometimes stored in the URL. This is called a **query** - it is *your* query, in fact - and you will find it after a question mark in the URL.

For a better example of this, let's look at the page for searching European Investment Bank loans.

The search page for loans made by the EIB is at http://www.eib.org/projects/loans/list/index.htm⁴ - change the *Country* drop-down menu to 'United Kingdom' and *Region* to 'European Union', select the years 2007 and 2012 in the *From* and *To* options, and click '**Go**', and you will be taken to this URL:

http://www.eib.org/projects/loans/list/index.htm?start=2007 &end=2012&region=european-union&country=united+kingdom&sector=⁵

The question mark is where the **query string** begins - so let's pick that query string apart:

```
start=2007&end=2012&region=european-union &country=united+kin
```

³http://www.hpc-uk.org/search-results/?searchOption=1&search=wilson&profession=SW

⁴http://www.eib.org/projects/loans/list/index.htm

⁵http://www.eib.org/projects/loans/list/index.htm?start=2007&end=2012&region=european-union&country=united+kingdom&sector=

Helpfully, this is pretty easy to understand - especially as the drop-down lists are still visible on the page.

- `start=2007` correlates to the 'From' drop-down menu on the form.
- `&end=2012` correlates to the 'To' drop-down menu.
- `&region=european-union` correlates to the region we selected
- `&country=united+kingdom` correlates to the country we selected
- `&sector=` correlates to the drop-down menu that we didn't change. There's *nothing after the = sign* because we didn't specify anything here.

In programming jargon, these are each called a **field-value pair**, because the first part is the name of the field in a database (e.g. 'region'), and the second part is the value of that field you're asking for.

Each of these pieces of information is basically a **parameter**, used by the code serving up the results page in much the same way as a function uses parameters.

Each field-value pair is separated by an ampersand (&) or a semi-colon (;).

As always, trial and error is useful in establishing how changing the drop-down menus affects the URL - and also whether you can manually change the URL and still get results.

If you try this URL, for example (where we remove the country and region values), you'll get over 3,000 results:

http://www.eib.org/projects/loans/list/index.htm?start=2007 &end=2012&region=&country=&sector=[6]

But removing the value for the start field will prevent you getting any results at all: http://www.eib.org/projects/ loans/list/index.htm?start=&end=2012&region=&country=&sector=[7]

Knowing all this, we could have looked back at that second page of GSCC results and try to change it so that it starts from the first result, not the 11th like so (note the field-value pair of o=1): http://www.gscc.org.uk/registerSearch.php? o=1#results[8]

This did indeed work. Its replacement is not so easy to bulk-search, but we can at least replace the name like so:

http://www.hpc-uk.org/search-results/?searchOption=1 &search=bradshaw&profession=SW[9]

- which means we could at least cycle through a list of names if we wanted to check those.

By the way, the query string you probably encounter most often without realising it is in a Google search. Look at all the information contained in a typical string: https:// encrypted.google.com/search?q=understanding+query+strings &ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-

---

[6] http://www.eib.org/projects/loans/list/index.htm?start=2007&end= 2012&region=&country=&sector=

[7] http://www.eib.org/projects/loans/list/index.htm?start=&end= 2012&region=&country=&sector=

[8] http://www.gscc.org.uk/registerSearch.php?o=1#results

[9] http://www.hpc-uk.org/search-results/?searchOption=1&search= bradshaw&profession=SW

a[10]

   To know more about what all this means, search for 'Google URL parameters' or similar - you'll find lists like this[11]

When the URL doesn't change

Unfortunately, this isn't the only way that a site stores information about the data you're searching for.

   Another way is to use cookies, which only exist on your browser, and for a limited time.

   To demonstrate this, try an empty search for school reports on the Independent Schools Inspectorate website[12]. The page fills up with results - but the URL doesn't change. This is because when you submit the search the webpage stores a cookie on your browser specifying what you were searching for, and the page then calls on that to display the appropriate information.

---

[10]https://encrypted.google.com/search?q=understanding+query+strings&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-a

[11]http://www.blueglass.com/blog/google-search-url-parameters-query-string-anatomy/

[12]http://www.isi.net/reports/

Open up the same URL in another browser or some time later (some cookies expire after a particular length of time, typically when you leave the site) and you'll find it empty of results once more...

...and that's how your scraper will see it. Because a scraper is not a browser.

Or is it?

Solving the cookie problem: Mechanize

The **Mechanize** library is a collection of functions designed to tackle problems around filling in **forms**, storing **cookies**, and generally anything where pretending to be a **browser** (which is what it does) comes in useful.

As always, Scraperwiki's live tutorials are a useful place to start in understanding it. Here's the code for the live tutorial around scraping pages behind forms (search results)[13]

```
# START HERE: Tutorial for scraping pages behind
form, using the
# very powerful Mechanize library. Documentation
is here:
# http://wwwsearch.sourceforge.net/mechanize/
import mechanize
import lxml.html
lotterygrantsurl = "http://www.lottery.culture.gov.uk/Advanced
br = mechanize.Browser()
response = br.open(lotterygrantsurl)
print "All forms:", [ form.name for form in
br.forms() ]
br.select_form(name="aspnetForm")
print br.form
br["ctl00$phMainContent$dropDownAwardDate"] = ["Between"]
br["ctl00$phMainContent$txtGrantDateFrom"] = "01/01/2004"
```

---

[13]https://scraperwiki.com/scrapers/new/python?template=tutorial-mechanize

```
br["ctl00$phMainContent$txtGrantDateTo"] = "20/01/2004"
response = br.submit()
print response.read()
```

We can understand the lines that import two libraries and the initialisation of a variable called `lotterygrantsurl`...

The two lines after that are where we start to use Mechanize.

```
br = mechanize.Browser()
response = br.open(lotterygrantsurl)
```

We can use what we've learned so far to decode some of this:

- `Browser` must be a **function** (because it's followed by parentheses) of Mechanize (because it's joined to it with a full stop, or period). We can search the documentation for Mechanize to find out more - and indeed it is mentioned there.
- `br` is a variable because it's initialised by the = sign.
- `br` is then used again with the `.open` method (we know it's a method because it's joined to the variable with a full stop), which is combined with the variable `lotterygrantsurl` created earlier, and the results passed into a new variable called `response`

Even if you don't understand what exactly is happening, or the documentation for Mechanize (which isn't the best), you can at least follow what happens to the webpage that's being scraped, as it's passed from variable to variable, and you can use trial and error to work out what's being done to it.

In this case, `br` is probably short for 'browser' (change it if it helps) and that 'browser' is used to open the URL (represented by the variable), and pass the results into `response`

Now it gets confusing:

```
print "All forms:", [ form.name for form in
br.forms() ]
```

Reading from right to left is particularly helpful here. We can guess that:

- `.forms()` is a **method** for getting infomation about...
- the object `br` (an educated guess would say `.forms` finds any forms in the HTML now contained in `br`)
- And the `for` loop goes through each of those forms, and puts it into an object called `form`
- Each time it loops, the `.name` method is used to find out the name of that form
- And the whole process is in square brackets, which makes it a **list**...
- Which is printed, after the string "All forms:"

From a programmer's perspective, this one line is well written because it condenses a number of processes within it, rather than spreading them across separate lines of code.

From a the perspective of someone new to this code, however, it can be harder to read because you're having to understand a number of processes all at the same time.

If you **RUN** the scraper and look for the line where "All forms:" is printed out, you'll see the following:

```
All forms: ['aspnetForm']
```
What's aspnetForm? Try searching the HTML of the page you're scraping. Here it is:

```
<body onload="javascript:hideElements();">
    <form name="aspnetForm" method="post" ac
id="aspnetForm">
<div>
```

So far, so literal. Working back, we can confirm what we thought:

- `.forms()` did indeed grab anything in the HTML with the tag `<form ...>`
- the `.name` method grabbed the bit in that tag with the attribute `name="`
- You'll also notice that the printed results were contained in square brackets.

We could go further and test this on another URL with more than one form to see if the print line showed us all of the form names (note: some forms might not have a name).

But for now we'll return to the rest of the code:

```
br.select_form(name="aspnetForm")
print br.form
```

So now we know why we were looping through the form names and printing them. We needed to know in order to use the right name in this next line (we could of course have simply looked at the HTML).

This name is used as a parameter for another method being used on `br` - `.select_form` - and then we print the

form, selected with the `.form` method.

This is where Mechanize is very useful. There should be a line in Console where that form is printed, like this:

```
<aspnetForm POST http://www.lottery.culture.gov.uk/AdvancedSea
application/x-www-form-urlenc ...more
```

Click on '...**more**' to see the full HTML that is being printed. Here it is:

This shows all the options for the search form. You can compare that to the HTML of the search page and find the same information, more spread out.

The key thing you're looking for here is the name of each field for the search form. The first one shown above is `"ctl00$phMainContent$lbGoodCause"` - quite a mouthful.

Search for that in the HTML and you can find it too:

```
<select size="4" name="ct100$phMainContent$lbGoodCause" mu
width:380px;">
        <option selected="selected" value="-1">Any</option>
        <option value="A">Arts</option>
        <option value="C">Charitable Expenditure</option>
        <option value="E">Health, Education, Environment</
        <option value="D">Health, Education, Environment a
        <option value="H">Heritage</option>
        <option value="M">Millennium</option>
        <option value="S">Sports</option>
```

This is the name of the <select> tag part of the form
where a user would select the type of good cause they
wanted to explore: arts, sports, and so on.

Note that each <option> has a value=" attribute: A, C,
E, and so on. **These values are what you need to know for
your scraper -** *not* **the text that is displayed to website
users**.

Our next 3 lines of scraper code, then, start to make a
lot more sense:

```
br["ct100$phMainContent$dropDownAwardDate"] = ["Between"]
br["ct100$phMainContent$txtGrantDateFrom"] = "01/01/2004"
br["ct100$phMainContent$txtGrantDateTo"] = "20/01/2004"
```

Each line names a particular part of the form (indicated
by the string in the first square brackets) and sets it to the
value on the right.

Why is the value on the end of the first line in square
brackets? Look at the page's HTML to work it out:

- The part of the form with a name of "ct100$phMainContent
  $dropDownAwardDate" has 3 <option> tags, with val-
  ues of "After", "Before" and "Between" respectively.
- The parts with the names "ct100$phMainContent
  $txtGrantDateFrom" and "ct100$phMainContent$txtGrantDateTo"

have no option values at all. It's also worth pointing out that they have a type= value of "text", which means they are open text fields rather than drop-down or tick box menus.

So ["Between"] in the first line of the scraper code above uses square brackets because it is selecting a particular option. The others don't because they are not.

You can add another line now to make this clearer if it helps:

```
print br
```

RUN the code now and click on '...more' at the point where br is printed and you'll be able to see how that object has been set to select a particular form at the URL and set parts of it accordingly - particularly this bit (note the asterisk):

```
<SelectControl(ctl00$phMainContent$dropDownAwardDate=[After, Before, *Between])>
<TextControl(ctl00$phMainContent$txtGrantDateFrom=01/01/2004)>
<TextControl(ctl00$phMainContent$txtGrantDateTo=20/01/2004)>
```

The final two lines finish things off:

```
response = br.submit()
print response.read()
```

Reading from right to left once again: the first line uses the .submit() method to submit the form using all the information we've stored in br (which form; what options and text in which fields).

The results of that are put in the variable response

And the .read() method is used to grab the results, which are printed

You can view those results then in the Console - they are different to the webpage you were looking at before, because they now represent a results page.

And, now that they are stored in that variable, you can start to store parts of it using the same techniques you have with other scrapers. There's also a cheat sheet on Scraperwiki[14] which includes further instructions on how Mechanize works, and how to solve particular problems such as selecting forms without names.

Recap

- If you're scraping searches, look to **see if the URL changes when you search**, and if the results URL works in another browser
- If it does, look after the question mark in the URL: this is where the **query string** begins, and you can customise this to scrape different results pages
- If it doesn't the website may be using **cookies** to store your search criteria and you'll need the **Mechanize** library to mimic a browser
- **Find the name of the right form** in your search page's HTML - and what **options** (if any) you'll need to 'select' in each part of the form. You may also need to enter text.
- **Customise the code above** for a different search page with a different form name and different values.

---

[14]https://scraperwiki.com/views/python_mechanize_cheat_sheet/

Once you're happy with that, you're ready to combine that knowledge with your previous scraping experience to store the results.

Tests

- Find another form-filling search - or use one of those listed at the beginning of this chapter. Look in the HTML to identify the names of the fields for that form.
- Search the European Investment Bank or Enforcement Register database and try to alter the URL to change the field-value pairs. Which ones work and which ones don't?
- Adapt the scraper so it lists fields for that URL
- Adapt the scraper so that it fills in particular fields with particular values
- Rewrite the line `print "All forms:", [ form.name for form in br.forms() ]` as multiple lines that break the code down into its separate actions. This make the script bulkier and less efficient, but it will help you understand it better. As you gain experience, you may start to combine multiple lines into one.

22 Scraper #21: Storing the results of a search

In the previous chapter we used the **Mechanize** library to imitate a browser that conducts a search on a webpage and prints the results. But if we want to use those results we need to extract and store them somehow.

For that we can use the **lxml** library we used in previous scrapers

Here's the code in full - you can also find it here[1]

```
#import libraries
import scraperwiki
```

---

[1]https://scraperwiki.com/scrapers/mechanize_lxml_example/

```
import mechanize
import lxml.html
#now create a function called scrape_table which
isn't run until the end...
#this gets passed an individual page (root) to
scrape - look to see where it is created lower down
def scrape_table(root):

    #grab all table rows <tr>

    rows = root.cssselect("tr")

     #create a variable to count the entries,
    set at 0

     idno = 0

     #create a record to hold the data

     record = {}

     #for each row, loop through this

     for row in rows:

          #create a list of all cells
         <td> in that row

         table_cells = row.cssselect("td")

         if table_cells:
```

```
 #if there is a cell,
record the contents in
our dataset, the first
cell [0] in 'Name' and
so on

 record['Name'] = table_-
cells[0].text_content()

 record['Town'] = table_-
cells[1].text_content()

 #add 1 to the counting
variable

 idno = idno + 1

 record['ID'] = idno

  record['Country']  =
table_cells[2].text_content()

 record['Year'] = table_-
cells[3].text_content()

 record['Ages'] = table_-
cells[4].text_content()

 record['Gender'] = table_-
cells[5].text_content()
```

```
                table_cellsurls = table_-
            cells[0].cssselect("a")

             record['URL'] = table_-
            cellsurls[0].attrib.get('href')

              # Print out the data
            we've gathered

             print record, '------------'

              # Save the record to
            the datastore - 'ID'
            is our unique key -

             scraperwiki.sqlite.save(["ID"],
            record)

   #SCRAPER STARTS WITH MECHANIZE HERE
   #set the URL containing the form we need to open
with mechanize
   starting_url = 'http://www.isi.net/reports/'
   #start using mechanize to simulate a browser
('br')
   br = mechanize.Browser()
   # Set the user-agent as Mozilla - if the page
knows we're Mechanize, it won't return all fields
   br.addheaders = [('User-agent', 'Mozilla/5.0
(X11; U; Linux i686; en-US; rv:1.9.0.1) Gecko/2008071615
Fedora/3.0.1-1.fc9 Firefox/3.0.1')]
```

```
   #open the URL previously defined as 'starting_-
url'
   br.open(starting_url)
   #find out and display (print) the names of any
forms in the HTML
   #i.e. <form ... name="
   print "All forms:", [ form.name for form in
br.forms() ]
   #as it happens, the name of the form in this
page is... "form"
   br.select_form(name="form")
   #submit the form and put the contents into
'response'
   response = br.submit()
   #read contents of 'response' and print
   #read contents of 'response' and put into 'html'
   html = response.read()
   print html
   root = lxml.html.fromstring(html)
   # START scraping by running scrape_table function
created above
   scrape_table(root)
```

Most of this should be familiar, as the code is used from our previous scrapers, so I'll just explain the new lines of code.

If we start with the part using `mechanize` to submit the form on the starting URL (http://www.isi.net/reports/), this is new:

```
   br.addheaders = [('User-agent',  'Mozilla/5.0
```

```
(X11; U; Linux i686; en-US; rv:1.9.0.1) Gecko/2008071615
Fedora/3.0.1-1.fc9 Firefox/3.0.1')]
```

The comments above this line of code explain it: this line adds some extra qualities to our fake browser (br) that makes the website think it's a Mozilla browser, among other things.

Why? Well, some websites will hide information or not work if they see that mechanize is being used.

Here's the next new piece of code:

```
html = response.read()
print html
root = lxml.html.fromstring(html)
```

We've used response.read() before - to read the contents of response. However, where before we just printed those contents, now we put them into a new variable, html *(note: don't include the original print response.read() line as well, or it will generate an error).*

The final line converts (using the function fromstring) that html object into something that lxml can work with: a new variable called root.

Root is then used with the scrape_table function we created earlier, and which is almost identical to the function of the same name in earlier scrapers, with these differences:

```
    idno = 0

            ...

            idno = idno + 1

            record['ID'] = idno
```

The first line of code, early on in the function, creates a new variable called `idno`, and sets it to a value of 0. Later, the `if` statement within the `for` loop adds 1 to that value each time it runs (*see the Tests at the end of this chapter for a simpler way of doing this*), and stores the value in `record` under the label `'ID'`.

Here's another two new lines that may need explaining:

```
table_cellsurls = table_-
cells[0].cssselect("a")

record['URL'] = table_-
cellsurls[0].attrib.get('href')
```

These two lines: grab the `<a>` tags (links) and put it into a variable called `table_cellsurls`, and then grab the `href="` **attribute** of the first tag (specified with the index `[0]`) and put it into a field in our dataset labelled `'URL'`.

So, following our data from initial search page URL right through to storing particular parts of it in our database, this is how it goes:

We start with our search URL (http://www.isi.net/reports/) which is stored in a variable. We then create a browser using the mechanize library, and give it various instructions, such as which browser to imitate (Mozilla Firefox), which URL to open (stored in our variable), and which form to submit (`<form name="form">`).

The results of submitting that form are read and stored in another variable, `html`. This is then converted into an lxml object (using `fromstring`), and we run a new function (`scrape_table`) on that object.

scrape_table grabs all the table rows and puts them into a new list variable called rows. It loops through those rows and for each one grabs all the table cells, putting those in another list variable called table_cells. Each item in that list (first, second, etc.), and one tag's link attribute, is stored under a different label in record, which is saved in the datastore. Along the way we also created a new variable called idno - just a number that is increased by 1 for each row - to act as our unique key for each record.

For more on ways of using mechanize to solve problems with forms, see Julian Todd's cheat sheet[2] and *Emulating a Browser in Python with mechanize*[3].

Recap

This scraper was largely about combining previous experience with two libraries. The key points to take away are:

- When you switch from Mechanize to lxml, you need to **convert the results to an lxml object** by:
- adding .read() to the variable to read it, and putting the results into a new variable like so: html = response.read()
- using .fromstring to convert that into an lxml object like so: lxmlobject = lxml.html.fromstring(html)
- You can create a unique ID number for each record in your data by initialising a new variable at 0, then

---

[2]https://scraperwiki.com/views/python_mechanize_cheat_sheet/
[3]http://stockrt.github.com/p/emulating-a-browser-in-python-with-mechanize/

adding 1 to it every time a new record is created, and then storing it.

There's also a tip to be made here...

Scraper tip: using print to monitor progress

The `print` statement can be used to see what the scraper is seeing at any particular point - and help you understand why things might not be working how you want them to.

For example, printing the variable `html` allows you to see what's in that variable - is it a string of text? Or an lxml 'object' (which shows itself as something like `<'Element at XYZ123'>`) which you need to unpack or translate somehow?

Likewise, the line:

```
print "All forms:", [ form.name for form in
br.forms() ]
```

Allows you to see the names of all forms in the object `br`. (It also adds "All forms:" at the start just to help you - but that's not necessary).

Tests

Only two tests for this chapter. The first follows on from that tip:

- Add `print` statements to your script to follow what's happening in the Console. Go back to some of your previous Scraperwiki scrapers and do the same.
- Can you simplify the line `idno = idno + 1` so that it isn't as long? Tip: look for 'assignment operators' in documentation and tutorials for Python, and play around with the plus and equals sign together like so: `+=`.

23 Scraper #22: Scraping PDFs part 1

NATIONAL ASSEMBLY

OFFICIAL REPORT

Thursday, 8th July, 2010

The House met at 2.30 p.m.

[Mr. Deputy Speaker in the Chair]

PRAYERS

ORAL ANSWERS TO QUESTIONS

Question No.250

Now we come to one of the biggest uses of scrapers: tackling PDFs.

PDFs present their own challenges, so it will take us a few chapters to cover those. One significant challenge is that PDFs are encoded in a way which is not immediately obvious. Unlike webpages, where you can look at the source HTML to see the underlying structure and target that,

scraping information out of PDFs takes a lot more trial and error.

> With many PDFs you can copy text from the document and paste them into spread-sheet software, with the structure retained. If you're only working with a couple of PDFs, this manual process is likely to be quicker than writing a scraper. But if you're dealing with dozens or hundreds of PDFs, or expect a large number to appear in the future, then scraping is an option to seriously consider.

But the challenges of PDFs are also the main reason why scraping is so useful: it can bring structure to the data underlying PDFs, in a way which would take so much longer if you were doing it manually by reading hundreds or thousands of documents.

Once again, a good place to start with this is Scraper-wiki's own tutorial scraper for PDFs, which can be found on the live tutorials page[1]. It doesn't actually save any data, but it does give us a starting point.

Here's the code:

```
##########################
# We use a ScraperWiki library called pdftoxml
to scrape PDFs.
# This is an example of scraping a simple PDF.
##########################
```

[1] https://scraperwiki.com/docs/python/tutorials/

```
import scraperwiki
import urllib2
import lxml.etree
url = "http://www.madingley.org/uploaded/Hansard_-
08.07.2010.pdf"
pdfdata = urllib2.urlopen(url).read()
print "The pdf file has %d bytes" % len(pdfdata)
xmldata = scraperwiki.pdftoxml(pdfdata)
print "After converting to xml it has %d bytes"
% len(xmldata)
print  "The  first  2000  characters  are:  ",
xmldata[:2000]
root = lxml.etree.fromstring(xmldata)
pages = list(root)
print "The pages are numbered:", [ page.attrib.get("number")
for page in pages ]
# this function has to work recursively because
we might have "<b>Part1 <i>part 2</i></b>"
def gettext_with_bi_tags(el):

    res = [ ]

    if el.text:

        res.append(el.text)

    for lel in el:

        res.append("<%s>" % lel.tag)
```

```
         res.append(gettext_with_bi_-
      tags(lel))

       res.append("</%s>" % lel.tag)

       if el.tail:

           res.append(el.tail)

  return "".join(res)

# print the first hundred text elements from the
first page
page0 = pages[0]
for el in list(page)[:100]:

   if el.tag == "text":

       print el.attrib, gettext_-
      with_bi_tags(el)

# If you have many PDF documents to extract data
from, the trick is to find what's similar
# in the way that the information is presented
in them in terms of the top left bottom right
# pixel locations. It's real work, but you can
use the position visualizer here:
# http://scraperwikiviews.com/run/pdf-to-html-preview-1/
```

Once again, a first thing to do with this code is trace what happens to the thing being scraped by looking for a URL.

That line is here:

```
url = "http://www.madingley.org/uploaded/Hansard_-
08.07.2010.pdf"
```

That's the PDF being put into a variable called `url`.

Finding and following `url` in turn we can see it being 'opened' (search for the documentation[2]) and 'read' to be put into a variable called `pdfdata` here:

```
pdfdata = urllib2.urlopen(url).read()
```

It's then used as a parameter in the **pdftoxml function**, and the results put into a variable called `xmldata`:

```
xmldata = scraperwiki.pdftoxml(pdfdata)
```

This is the first time we've encountered this function. As you might guess, it's very useful in scraping PDFs.

With any new function, it's a good idea to check out what documentation exists around it. Because it comes after scraperwiki., we know that it's part of that library, and a search for 'scraperwiki documentation' will take you to https://scraperwiki.com/docs/python/python_help_documen-tation/[3], which you can search within to find pdftoxml. That doesn't tell us a lot, but it does tell us that it converts a PDF file to an XML file "containing the coordinates and font of each text string" - something which will be useful if we want to grab those text strings.

We can also search the web more generally for uses of 'scraperwiki pdftoxml'.

Back to the scraper, our PDF data is now contained in the variable `xmldata`, and that's used as the parameter in

---

[2]http://docs.python.org/library/urllib2
[3]https://scraperwiki.com/docs/python/python_help_documentation/

a line soon after (notice that we're using the .**fromstring** function from the **etree** part of the **lxml** library. As always, if you want to know more look at the documentation for lxml.etree to find out about that and other possible functions you could use):

```
root = lxml.etree.fromstring(xmldata)
```

From previous experience, or from searching the documentation, we might remember that this converts our variable from a string into an lxml object, which we've called 'root'. The main difference is that where **lxml**.**html** - which we used before - is used for html webpages, **lxml**.**etree** - which we're using now - is used for XML documents. And we've converted that PDF into an XML document.

In the next line we get another new function:

```
pages = list(root)
```

Searching for documentation on it finds[4] that it "takes sequence types and convert them to lists."

And indeed, if we wrote

```
print pages
```

...in the next line, and then ran it, we'd get something like this:

```
[<Element page at 0x1ae3fa0>, <Element page at
0x1b70050>, ...]
```

...a list of lxml objects.

So now our data is in a list.

Ignore the print commands which tell us about the variables, for now, and keep your eye on that pages variable:

---

[4]http://www.tutorialspoint.com/python/list_list.htm

```
   # print the first hundred text elements from the
first page
   page0 = pages[0]
   for el in list(page)[:100]:

      if el.tag == "text":

          print  el.attrib,  gettext_-
        with_bi_tags(el)
```

In the first line of this section - after the comment - 'pages' is used to create a new variable, page0. Specifically, it grabs the first item in the pages list by using [0] - an index of zero (meaning the first item).

What's curious is that page0 is then not used again. It's a dead end - and I think this is an error in the code (*In fact, I checked - and it is*), because the comment indicates that this section should be dealing with the first page, i.e. pages[0]

So the next line should read:

```
   for el in list(page0)[:100]:

      if el.tag == "text":

          print  el.attrib,  gettext_-
        with_bi_tags(el)
```

Now we no longer have a dead end.

The **for loop** we've seen before, and we know what the **list** function does too. But [:100] is new. Some background reading on Python will help here, but to save you having to do that, here's a quick detour:

Detour: indexes and slicing shortcuts

We've already seen that indexes use a number in square brackets to specify which item in a list, or which character in a string, to grab. But you can also specify a range in the same way. For example:

```
page0 = pages[0]
```

Grabs the first page in a list. But you can use a colon to specify a range:

```
page0 = pages[0:10]
```

This will grab the first to tenth elements (*yes - the 10th (index 9), not the 11th (index 10). There's probably an insanely logical reason for this*).

Because Python *assumes you start from the first item* unless otherwise specified, you can also use **shortcuts** to *simplify* the code:

```
page0 = pages[:10]
```

This does exactly the same as the last line of code.

Python also assumes that you end with the *last* item unless otherwise specified, so the following code:

```
page0 = pages[10:]
```

...will grab everything from the 11th item (index 10) to the last.

Finally, remember you can use negative indexes as well. The following means *last*:

```
page0 = pages[-1]
```

And you can specify a range in the same way:

```
page0 = pages[-8:-1]
```

So now we know that the code in the scraper `for el in list(page0)[:100]:` uses the `list` function to turn **the first 100 items into a list**.

Back to the scraper

```
for el in list(page0)[:100]:

    if el.tag == "text":

        print el.attrib, gettext_-
    with_bi_tags(el)
```

Within this *for loop*, then, we're grabbing the tag property of each `el`. Remember that `el` is still an lxml object (you can see this by adding `print el`), and `.tag` is part of lxml.

A read of the documentation at http://lxml.de/tutorial.html[5] tells us that "The XML tag name of elements is accessed through the tag property".

The == operator means 'equals to' and now we need another detour:

Detour: operators

Operators are used in Python and other programming languages to test or change values. You'll be familiar with simple operators like +, -, * and / which are used to add, subtract, multiply and divide values.

---

[5]http://lxml.de/tutorial.html

The *equals sign* in Python, however, is used to *create* a variable like so:

```
page0 = pages[-1]
```

So a *different* operator is needed to test if something is equal to something else. And that is the **double equals sign**:

```
    if el.tag == "text":
```

There are a number of other operators - just search for 'Python operators' to find a list[6]. But the ones that compare two values and give you a 'true' or 'false' answer are particularly useful:

```
!=
```

...for example, means *not* equals to. And this:

```
>=
```

means greater than *or* equal to. Likewise this:

```
<=
```

means less than or equal to.

Back to the scraper (again)

```
for el in list(page0)[:100]:

    if el.tag == "text":

        print el.attrib, gettext_-
        with_bi_tags(el)
```

---

[6]http://www.tutorialspoint.com/python/python_basic_operators.htm

So, if that condition is met - i.e. that the tag of the lxml object el is 'text', then we move on to the next command, which is to print two things: the attribute of el, and the result of a new function, which uses el as its ingredient:

```
gettext_with_bi_tags(el)
```

Where's that function? Higher up the code (it had to be created first in order to work here).

```
# this function has to work recursively because
we might have "<b>Part1 <i>part 2</i></b>"
def gettext_with_bi_tags(el):

    res = [ ]

    if el.text:

        res.append(el.text)

    for lel in el:

        res.append("<%s>" % lel.tag)

        res.append(gettext_with_bi_-
        tags(lel))

        res.append("</%s>" % lel.tag)

        if el.tail:

            res.append(el.tail)

    return "".join(res)
```

Once it is defined, this function first creates an empty list (indicated by the square brackets) called `res`. This empty list is then going to be filled...

...**If** the `el` variable which is used as the main ingredient in this function ('passed' in the first line's parentheses) has *any* text property, then...

The .**append** function (actually a **method**, but let's not split hairs) is used to append - add - that text to the list contained in the variable `res`. A quick search about that function tells you more[7].

Now we have another **for loop** which creates a variable `lel` as it runs.

This variable is then used to append *more* items to the same list variable `res`. And here we encounter more new code - the **percentage symbol**.

Detour: the % sign explained

As always, when you encounter a new symbol or function it's a good idea to look for any documentation about it. A quick search for "percentage symbol in Python" would bring up a number of resources explaining (or trying to) how it works.

Some of this may be contradictory, because the `%` sign can act as a mathematical **operator** in calculations (like the plus, minus, and multiplication signs), *and* it can be used in **string formatting** too.

Here's the sign used in a calculation:

---

[7]http://www.tutorialspoint.com/python/list_append.htm

```
9 % 3
```

The result of this calculation is 0 - because % basically means 'divide by and give me any remainder'. 9 divided by 3 is 3 - with no remainder (put another way: 9 divides by 3 perfectly, so there is no ugly remainder left over). Here's another example:

```
9 % 2
```

The result here is **1**. 9 divided by 2 is 4, with a remainder of 1.

Now that's interesting, but unlikely to come in useful in a scraper. The other use of the percentage sign is much more interesting...

There are two things to look for here: firstly, that the % sign appears within a string; and secondly that the letter 's' appears immediately after it: `%s`

Here's some code you can write within Scraperwiki to try it out (create a new blank scraper first):

```
missingword = 'characters'
fullsentence = 'Here is a string of %s with a
missing word' % missingword
print fullsentence
```

The first line creates a variable.

That variable is then **inserted** into the string in the second line at the point at which the %s sign appears.

The % sign between the string and the variable acts as an instruction of sorts: 'insert into'. *Reading from right to left helps again here.*

You can write it another way without using an initial variable:

```
fullsentence = 'Here is a string of %s with a
missing word' % 'characters'
print fullsentence
```

And you can write it even simpler, using no variables at all:

```
print 'Here is a string of %s with a missing
word' % 'characters'
```

Read from right to left, you might describe it like so:

The string 'characters' gets **inserted** *into the string 'Here is a string of %s …'* at the point at which %s appears. *Print the results.*

One last thing: the 's' after % indicates that you are inserting a string. If you are inserting a number, use %d (think d for digit), and if you are inserting a float - a number with decimal places, such as 3.003 - then use %f.

In fact, you can see this in operation earlier in the code:

```
pdfdata = urllib2.urlopen(url).read()
print "The pdf file has %d bytes" % len(pdfdata)
```

In this example the code is finding a number - the length of the variable pdfdata - and inserting it into the string 'The pdf file has…'

Back to the scraper (again) (again)

So, now that section of our code makes more sense:

```
for lel in el:

    res.append("<%s>" % lel.tag)
```

```
     res.append(gettext_with_bi_-
    tags(lel))

    res.append("</%s>" % lel.tag)

    if el.tail:

        res.append(el.tail)

return "".join(res)
```

This line:

```
res.append("</%s>" % lel.tag)
```

Is inserting the tag of the variable lel (created by the **for loop**, remember) into the string "`<%s>`" instead of `%s`. It is basically adding tag brackets around the tag that's been extracted, before it's appended to the list variable `res`.

The line after that:

```
     res.append(gettext_with_bi_-
    tags(lel))
```

...uses a function called `gettext_with_bi_tags` on `lel` and appends the results of that operation.

But that's the function containing this line! This is what's called working **recursively** and it's helpfully explained in the comment above the function:

```
# this function has to work recursively because
we might have "<b>Part1 <i>part 2</i></b>"
```

So, because we might have tags within tags, the function runs within itself

The third line, after that, adds the closing tag (note the backslash).

Two more lines add a tail property if it exists (once again, check out the documentation on lxml[8] for more on this:

```
if el.tail:

    res.append(el.tail)
```

And a final line uses the **join** function to join the whole res list into one (with *nothing* between each item, indicated by the empty quotation marks), and **returns** it to us:

```
return "".join(res)
```

The **return** statement ends the function, and sometimes is used with nothing else in order to do only that.

But it can also return information to whatever used it - so once the function finishes that information - a series of tags, joined together - is returned to the line we covered before:

```
print el.attrib, gettext_with_bi_tags(el)
```

You can test this by changing your code earlier to read this:

---

[8]http://lxml.de/tutorial.html

```
    return "BLAH".join(res)
```

And then run the script to see where that text appears: indeed, it appears after a series of printed attributes.

Recap

Although we haven't scraped anything yet, this chapter has introduced quite a lot of new elements of code which it will be important to understand. Here is a roundup:

- The library **pdftoxml** is, not surprisingly, used to convert PDFs into XML. This can be useful in scraping the resulting XML file with the lxml library. In this scraper, for example, we've used the .**tag** and .**tail** properties of lxml objects to extract that information and add it to a list.
- The **list** function, also not surprisingly, converts things to lists. You can then loop through that list, or select items from it.
- When identifying **indexes** in an item or list of items you can **select a range** by using the **colon**, e.g. `[0:10]`. Python also assumes - if you don't specify otherwise - that you start from the first item, or end at the last item so `[:100]` will select from the first to the 100th; and `[10:]` will select from the 11th to the last.
- **Operators** are signs used to perform calculations, comparisons or other tests and processes. The operator `==` for example tests if the values either side

of the operator are the same. The result of this comparison is either 'True' (yes they are) or 'False' (no they're not). Operators can also be used to test if two things are not the same, or if one is greater or lesser than another. There are dozens of operators - google 'Python operators' to find a list[9].

- An **if statement** tests whether something is true or false. If it is true, then the script following it underneath (indented) runs. If not, then the indented script is skipped. An if statement - like functions and for loops - should always **end with a colon** to indicate the start of the script to run next. Sometimes an if statement merely tests whether something exists. For instance `if el.text:` tests whether the variable el has a text property. If no such property exists, then the result is 'false' and the script does not run.

- The .**append** method **adds items to a list**. The name of the list comes before .append, and the data to be appended comes after .append in parentheses. Sometimes this might be a function, which returns data when run.

- The % **sign** can be used to either calculate a remainder or **insert a value into a string**. If you're inserting a value into a string **add 's', 'd' or 'f' to the % sign** to specify whether you're inserting a string, digit or float (number with decimal places). The string is then followed by the % sign on its own, and the value you're inserting (often a variable) like

---

[9]http://www.tutorialspoint.com/python/python_basic_operators.htm

so:`print 'Here is a string of %s with a missing word' % 'characters'`

- **Recursive functions** are functions which run within themselves. So *our* function above picks out tags and adds them to a list, but in the middle of running that, it runs through the same process: picks out more tags (now already within tags) and adds them again. This can make your head hurt: like standing between two mirrors, you wonder when it can end. But it works, and is especially useful when you have problems buried within problems, like tags within tags. At the moment, the main thing is to not get freaked out when you see one.
- The .**join** function joins parts of a list into one thing, with each part separated by whatever string you put before .join.
- A **return statement** returns certain results of a function back to whatever called (used) it. It also ends the function.

Tests

- Read up on the **% operator** with strings. Try it out with some basic experiments. A helpful post is here[10].
- Look at some of the **properties** in the lxml documentation, such as .tag, .tail and .text - try to extract and print them in the code.

---

[10]http://importflying.com/2011/conversational-programming-how-to-use-the-percent-sign-operator-in-python/

- Play around with lists and the **append** method - try some simple scripts in an empty Scraperwiki scraper just to test it out.

24 Scraper 23: Scraping PDFs part 2

Mobile Routes: Week Start 03/09/2012

Monday   : 03/09/2012

005	A34 Stone Roadbetween Redhill (A513 / A34) island and Lloyds island / Eccleshall Road.
006	A34 Stafford South between A449 junction to Acton Hill Lane junction.
011	A449 (Stafford) between Lichfield Road and Gravel Lane.
012	A449 Penkridge between Lynehill Lane and half mile north of Goodstation Lane.
014	A449 Gailey between Rodbaston Drive and Station Drive.
018	A51 Weston between New Road and 500 metres past Sandy Lane (going north).
019	A51 Pasturefields between Amerton Lane and Hoomill Lane.

With the basic parts of a PDF scraper explored, we can now actually start grabbing some parts of that PDF.

To do that in the most simple way, we can just borrow some code from a previous scraper to add that to the scraper tutorial we explored in the last chapter.

Here's the key line:

```
for el in list(page0)[:100]:
 if el.tag == "text":
 print el.attrib, gettext_with_bi_tags(el)
```

Until now this has just been printing the results. Adding some lines will also save them. Here are the lines:

```
ID = 0
for el in list(page0)[:100]:
 print el.tag
 if el.tag == "text":
```

```
print el.attrib, gettext_with_bi_tags(el)
record = {}
record["text"] = gettext_with_bi_tags(el)
ID = ID+1
record["ID"] = ID
scraperwiki.sqlite.save(["ID"],record)
print record
```

The first line creates a variable called ID, and sets it at zero (0). This will be used in the **for loop** that follows, because we will need a unique reference (key) for every entry.

Skipping the first few lines of that loop which were explained in the last chapter, we come to these:

```
record = {}
record["text"] = gettext_with_bi_tags(el)
ID = ID+1
record["ID"] = ID
scraperwiki.sqlite.save(["ID"],record)
print record
```

These have been explained in previous chapters, but to recap line by line:

- Create a variable called 'record', and assign it the value of {} (a dictionary with nothing in it for now)
- Now add a 'text' field to that variable, and allocate the results generated by running the gettext_-with_bi_tags function on the variable el.
- Add 1 to the ID variable...
- ...and assign that to an 'ID' field in the variable 'record'

- Save the results to the scraperwiki database, with `'ID'` as the unique key
- Print them too

Save this and click **RUN** and you should see the **Data** part fill up with entries as it loops through every item in the list and matches it against the *if* test.

These entries don't make much sense, but at least we know it works. The next step is tweaking the bit that selects what we want to scrape, so that it grabs the right part.

Where's the 'view source' on a PDF?

A large obstacle to scraping PDFs is that you cannot right-click and 'view source' on a PDF in the same way that you can with HTML webpages.

But as with so many obstacles in programming and scraping, someone has already tackled that problem. Indeed, there's a link to it in that tutorial scraper in the comments at the end: https://views.scraperwiki.com/run/pdf-to-html-preview-1/[1]

This is the *PDF to HTML Preview* tool, written by Scraperwiki's Julian Todd[2].

Open up that URL[3] in a new browser window, and in the box paste the address of the PDF from the tutorial in

---

[1]https://views.scraperwiki.com/run/pdf-to-html-preview-1/
[2]http://blog.scraperwiki.com/2010/12/17/scraping-pdfs-now-26-less-unpleasant-with-scraperwiki/
[3]https://views.scraperwiki.com/run/pdf-to-html-preview-1/

the previous chapter[4] - then click **Go** to see the underlying code.

If you look in the box of code shown underneath you should be able to match the code to the parts being grabbed by your scraper (click in the box and use **Find** in your browser to get there quicker), namely:

```
<text top="1099" left="135" width="528" height="16"
font="0"><b> 1 Thursday, 8</b></text>
<text top="1095" left="663" width="11" height="11"
font="1"><b>th</b></text>
<text top="1099" left="673" width="114" height="16"
font="0"><b> July, 2010 (P) </b></text>
<text top="115" left="309" width="307" height="24"
font="2"><b>NATIONAL ASSEMBLY </b></text>
```

Each line uses the <text> tag, which is what the scraper was looking for. Some of those lines are empty, resulting in empty records in our dataset. You can also see how some text is separated, for example because of formatting (the 'th' in '8th' gets a separate tag because it uses a different font).

This is going to be particularly helpful in scraping another PDF...

---

[4]http://www.madingley.org/uploaded/Hansard_08.07.2010.pdf

Scraping speed camera PDFs - welcome back to XPath

This example comes from local newspaper journalist David Elks, who used some of the skills covered in this book to look at PDFs containing data on speed camera locations.

Mobile Routes: Week Start 03/09/2012

Monday : 03/09/2012

005	A34 Stone Roadbetween Redhill (A513 / A34) island and Lloyds island / Eccleshall Road.
006	A34 Stafford South between A449 junction to Acton Hill Lane junction.
011	A449 (Stafford) between Lichfield Road and Gravel Lane.
012	A449 Penkridge between Lynehill Lane and half mile north of Goodstation Lane.
014	A449 Gailey between Rodbaston Drive and Station Drive.
018	A51 Weston between New Road and 500 metres past Sandy Lane (going north).
019	A51 Pasturefields between Amerton Lane and Hoomill Lane.
022	A51 Tamworth, between Peelers Way and Ascot Drive.
024	A518 Stafford, between Riverway and Blackheath Lane.
027	A53 Leek New Road Endon between junction with Nursery Avenue and junction with Dunwood Lane.
028	A53 Longsden between junction with Dunwood Lane and junction with Wallbridge Drive.
030	B5080 Pennine Way,Tamworth between B5000 and Pennymoor Road.
031	B5404 Tamworth between Sutton Road and junction of A4091 (Coleshill Road / Fazeley Road).
036	A5 between A461(Muckley Corner) and A5127 (Wall Island).
037	A5 between A5127 (Wall Island) and A38 (Weeford island).
041	A454 Bridgenorth Road, Trescott between Brantley Lane and Shop Lane.
057	B5404 Watling Street, Tamworth between A51 and A5.
062	B5051 Stoke on Trent, between Sneyd Hill Road and Brown Edge.
068	A53 Blackshaw Moor between Thorncliffe Road and Hazel Barrow Lane.
073	A449 Stourton between Ashwood Lower Lane and Dunsley Lane.
075	A5 Weston under Lizard.
079	A51 Hopwas Hill/Lichfield Road, Hopwas.

An example PDF can be found at http://www.staffssaferroads. co.uk/media/114997/03092012_forwebsite.pdf[5]. If you put this address into the *PDF to HTML Preview* tool you will see code like this:

```
<text top="131" left="123" width="169" height="16"
font="3"><b>Monday : 03/09/2012</b></text>
<text top="128" left="292" width="6" height="20"
```

---

[5] http://www.staffssaferroads.co.uk/media/114997/03092012_forwebsite.pdf

```
font="4"><b> </b></text>
    <text top="152" left="106" width="5" height="17"
font="0"> </text>
    <text top="156" left="123" width="20" height="12"
font="5">005</text>
    <text top="152" left="143" width="5" height="17"
font="0"> </text>
    <text top="156" left="166" width="474" height="12"
font="5">A34 Stone Roadbetween Redhill (A513 / A34)
island and Lloyds island / Eccleshall Road.</text>
    <text top="153" left="640" width="4" height="15"
font="6"> </text>
    <text top="170" left="106" width="5" height="17"
font="0"> </text>
    <text top="174" left="123" width="20" height="12"
font="5">006</text>
```

Each line is, once again, contained within a <text> tag. This time, we are going to grab these in a different way, using a tool from the very early chapters: **XPath**.

Because we are converting PDFs to XML documents with the pdftoxml function, we can deal with those XML documents in much the same way that we dealt with them way back when we were using the importXML function in Google Docs (*remember? Oh, we were so young then.*), by writing XPath to specify the part of XML we wanted.

To demonstrate this, create a new blank scraper. The first few lines will be the same as our last scraper, but with a different PDF URL:

```
import scraperwiki
```

```
import urllib2
import lxml.etree
url = "http://www.staffssaferroads.co.uk/media/114997/0309201
forwebsite.pdf"
pdfdata = urllib2.urlopen(url).read()
print "The pdf file has %d bytes" % len(pdfdata)
xmldata = scraperwiki.pdftoxml(pdfdata)
print "After converting to xml it has %d bytes"
% len(xmldata)
root = lxml.etree.fromstring(xmldata)
```
Now we add some new lines:
```
# this line uses xpath to find <text tags
lines = root.findall('.//text')
print lines
for line in lines:

    print line.text
```

These should be familiar - apart from that bit of XPath. The first line uses the .findall function on *root* to find any tags matching the XPath of `.//text`

There is some [documentation on the use of XPath in lxml.etree](6) or you can refer to the [general documentation on XPath on w3schools.com](7).

In brief, the full stop at the start is needed to specify the current 'node' (any particular branch of the XML 'tree' of tags). Without it (try it!) you will get an error telling you that it "cannot use an absolute path". You didn't need that

---

[6] http://lxml.de/tutorial.html#using-xpath-to-find-text
[7] http://www.w3schools.com/xpath/xpath_syntax.asp

full stop in Google Docs, where things were simpler, but you do here.

The double slash `//` is used as it was in Google Docs, to select a tag, or as the lxml documentation says, parts "that match the selection no matter where they are" - the selection being 'text'. You can try to change that to 'page' or 'fontspec' - the other two tags here - to see the results: neither has any text content but they will run without error.

If you revisit the chapters on importXML and XPath, you'll find you can use the same techniques here to specify which part of the PDF you want (once converted to XML).

So, if you wanted to grab the contents of every text tag with a font value of "3" you could adapt your first line like so:

```
lines = root.findall('.//text[@font="3"]')
```

If you try this, you'll find it works without any errors, but you get a series of 'None' results. Closer inspection of the XML shows you that this is because a ‹b› tag comes straight after it (any new tag effectively ends the search). So you can adapt your code further to grab the contents of that tag like so:

```
lines = root.findall('.//text[@font="3"]//b')
```

Try that. You should see each date contained within those tags printed as it runs.

Now, to save the results into a dataset, we again borrow from previous experience, and loop through the list of results from that .findall function, storing them in turn with this code:

```
 record = {}
```

```
for line in lines:

  record["date"] = line.text

  scraperwiki.sqlite.save(['date'], record)
```

Explained line-by-line this does the following:

- Create a variable, 'record', and make it an empty dictionary.
- We use a **for loop** to go through each item in the list lines. As it loops, that item will be called 'line'.
- Create a field 'date' in that dictionary, and assign the text value of the item to that field.
- Save the results, with 'date' the unique key. At the moment this is fine, because they're all different and there are no empty entries - but we'll need to find another unique key if we're to scrape all entries.

You now have a working PDF scraper with data which makes sense. But if we're going to grab all the information on this PDF, we'll need to add some fussier lines.

Ifs and buts: measuring and matching data

To demonstrate our problem, try amending this line:
```
lines = root.findall('.//text[@font="3"]//b')
```
...to this:
```
lines = root.findall('.//text[@font="5"]')
```

This line should catch the lines in the PDF that look like this (note the font="5" part):

```
<text top="156" left="166" width="474" height="12"
font="5">A34 Stone Roadbetween Redhill (A513 / A34)
island and Lloyds island / Eccleshall Road.</text>
```

But click **RUN** and you'll see a problem: it's also grabbing the data in these lines:

```
<text top="174" left="123" width="20" height="12"
font="5">006</text>
```

As usual, we could choose to clean this up somehow outside of the scraper. But we won't always have that option. If we want to grab both pieces of data separately, and get some structure from this, then, we will have to look for other identifying information.

There are a number of possibilities here. Each piece of data has the following attributes: *top; left; width; height*; and *font*.

Top and *left* are different every time, as they are used to position the text on the page. *Font*, we know, is the same for both pieces of data. And *height* is too. *Width* is relatively consistent at 12 for the codes - although it will be longer for the later ones which have an extra character.

In addition, there is the data itself. What patterns are there in it? Here are two:

- The data seems to *alternate* between codes and locations. We could find out if there is some code we could use to grab alternates.
- The codes are relatively consistent: each appears to be only three characters. Looking at the original

PDF, however, shows that the last few are four characters. So we can amend that to say 'three or four characters' or 'four or fewer characters' - which is enough to distinguish it from the location information.

How do we test that? In an earlier chapter we covered **if/else statements**: parts of script that say 'if this is so, do this; otherwise (else), do that.

Here, then, is how we might use an **if statement** to store data if it is a particular length:

```
lines = root.findall('.//text[@font="5"]')
 record = {}
for line in lines:

    if len(line.text)>4:

    record["date"] = line.text

 scraperwiki.sqlite.save(['date'], record)
```

The only line to be added is that **if statement** which uses the **len function** to find out what the *length* of the text (the number of characters) is. As we only want to grab parts of data longer than four characters, we use this line to say '*if* the length of the text in the variable 'line' is *more than 4*, then execute the indented lines following the colon at the end.

The next two lines are the same as before - with the key difference that they are now *indented* by four spaces as part of that if statement. Those lines will only be followed if the if statement is met.

So, as the **for loop** goes through each item in the list variable 'lines', it checks each item against that if statement. If the condition is met, the data is saved. If it is not met (i.e. the text is less than five characters), then it is not.

(*By the way, the field is still called 'date' even though we are now scraping locations, so we'll change that at some point, but functionally it doesn't really matter what it's called*)

But we *also* want to save those pieces of data with four or less characters - the codes.

We can create another if statement to look for those, saying '*if* the length of the text in the variable 'line' is *less than 5*, then execute the indented lines following the colon at the end.

But if we added another if statement after the end of the first if statement, it wouldn't work - at least not properly - because the two if statements would be saving the data separately, rather than connecting them together.

So we need to find a way to run *both* if statements and save the data *once.*

This will involve some trial and error - and I'm going to intentionally explore some of the errors here so we can learn from them.

Here's a first run, then, which attempts to grab both bits of data:

```
if len(line.text)>4:

    record["location"] = line.text

    print record
```

```
if len(line.text)<5:

  record["code"] = line.text

print record
scraperwiki.sqlite.save(['code'], record)
```

Clicking **RUN** with this will save just the codes. Although the locations will be *printed*, they are not being saved to the database, because that happens in the *second* **if statement**.

How do we carry the data across to that statement then? Well, we can try storing it in a variable like so:

```
if len(line.text)>4:

  location = line.text

  print record

if len(line.text)<5:

  record["code"] = line.text

  record["location"] = location

 print record
 scraperwiki.sqlite.save(['code'], record)
```

Clicking **RUN** on this, however, generates the following error:

```
NameError: name 'location' is not defined
```

This is the same problem as we encountered in the previous chapter. Why is it not defined? As before, going

through the script step by step will help show us the problem.

On the first line it asks 'is this item greater than 4 characters?' Because the codes come first and then the location, the answer is *no* on the first run, and so *the following lines are not executed*. This means that the location variable is not created ('instantiated' in the jargon - or 'defined' in the error message). So when it gets to the next if statement there is no 'location' for it to store.

As before, then, one way to solve this would be to instantiate the variable *before* the for loop runs, like so:

```
location = ""
if len(line.text)>4:
```

This will get rid of the error message but when you **RUN** this the data will be out of sync by one line: as the scraper runs, the first line in the *Data* view will show the code 005 and an empty location. The code 006 will be associated with the location "A34 Stone…", which (checking the original PDF) should be associated with the code 005.

A better solution, then, may be to change the order in which we're grabbing the data. We need our *first* if statement to grab the *first* piece of data, otherwise the second if statement will run first!

Change your code to the following then:

```
if len(line.text)<5:

  code = line.text

  print record

if len(line.text)>4:
```

```
    record["location"] = line.text

    record["code"] = code

  print record
  scraperwiki.sqlite.save(['code'], record)
```

Click **RUN** and the scraper not only runs OK but the data matches up with what we see on the PDF.

(*Although it has worked, instantiating that location variable before the for loop might not be a bad idea if there are entries where there is a code but no location.*)

There's also another thing we might do: add a **continue statement** at the end of the first if statement to ensure that it does run on to the second. You can find more about this in the [documentation around if statements](http://docs.python.org/tutorial/controlflow.html)[8]. This would simply look like this in the code:

```
  if len(line.text)<5:

    code = line.text

    print record

    continue

  if len(line.text)>4:
```

Now you have the two most important pieces of data. How to grab the dates too?

Firstly, we're going to have to be less fussy in that line of code which creates a list of data to loop through:

---

[8]http://docs.python.org/tutorial/controlflow.html

```
lines = root.findall('.//text[@font="5"]')
```

As the date information uses font="3" we'll need to simplify to catch that and the other two bits of data by changing it like so:

```
lines = root.findall('.//text')
```

This means we'll also need to identify the font value (3 or 5) later on. We can do this with the following line of code within the for loop:

```
for line in lines:

    fontvalue = line.get("font")
```

This uses the **.get** method to get the value of the font attribute of each 'line', and puts that value in a new variable we've called 'fontvalue'.

That variable can then be used in further code like so:

```
 if fontvalue == "3":

    date = line.find('.//b').text
```

These two lines say:

- If the variable fontvalue is "3" (== being the operator that basically means 'equals to')
- Then create a new variable called 'date', and put in it: the results of using the **.find** method on 'line' to find anything within a <b> tag (we're using XPath again here - remember that dates have that added formatting), and then grabbing any text in that.

If this second part is confusing you by cramming too much into a single line, you can always break it up into more like so:

```
boldtext = line.find('.//b')

date = boldtext.text
```

Carrying this logic through to the other parts of the data we would get a final *for loop* which looks like this - I've added comments before each line to explain what they do:

```
# this line uses xpath, which is supported by
lxml.etree (which has created root) to grab the
contents of any <text> tags and put them all in a
list variable called 'lines'
lines = root.findall('.//text')
  #create an empty dictionary variable called
'record'
 record = {}
# loop through each item in the list, and assign
it to a variable called 'line'
for line in lines:

   #create a variable called 'fontvalue',
   and use the get method to give it the
   value of the <font> attribute of 'line'

   fontvalue = line.get("font")

   #print that variable
```

```
print fontvalue

#if that variable has a value of '3'

if fontvalue == "3":

     #create a variable called
    'date', use the find method to
    grab the contents of any <b>
    tag (identified with XPath) in
    'line', grab the text within
    that, and store it in the
    variable

     date = line.find('.//b').text

    #continue on to next if statement

    continue

  #if 'fontvalue' is 5 AND the length
of the text in 'line' is less than
5 characters (counted with the len
function)

 if fontvalue == "5" and len(line.text)<5:

     #grab the text in 'line' and
    put it in a new variable called
    'code'

     code = line.text
```

```
        continue

   #if 'fontvalue' is 5 AND the length
of the text in 'line' is more than 4
characters

 if fontvalue == "5" and len(line.text)>4:

     #grab the text in 'line' and
    store it in a field called
    'location' in the dictionary
    variable 'record'

     record["location"] = line.text

     #store the value of the 'date'
    variable in a field called
    'date' in the dictionary variable
    'record'

     record["date"] = date

     #store the value of the 'code'
    variable in a field called
    'code' in the dictionary variable
    'record'

     record["code"] = code

     #print the value of 'record'

     print record
```

```
 #save those values in scraperwiki's
sqlite database, with 'code' as the
unique key

 scraperwiki.sqlite.save(['code'], record)
```

If we wanted to create our own unique key in case we couldn't rely on the data to be unique, we could add the following line just before the for loop:

```
uniquekey = 0
```

...and change the last line of the loop to these three:

```
 uniquekey += 1

 record["uniquekey"] = uniquekey

 scraperwiki.sqlite.save(['uniquekey'],
record)
```

This adds 1 to the value of 'uniquekey' (*+= 1 is just a quicker way to write = uniquekey+1*), then stores the value of 'uniquekey' in a new field in the dictionary variable 'record', called 'uniquekey'. In the final line we have changed the key of the sqlite database to that new field.

To see the final code in full with comments go to the working scraper[9] and click **View source** or **Edit**.

---

[9]https://scraperwiki.com/scrapers/pdftoxml_livetutorial_pt3/

Recap

This chapter has taken in quite a lot of key elements of PDF scraping. Here are the highlights:

- You can **view the XML code underlying a PDF** by using the PDF to HTML Preview tool[10] - or you can use the `print` command in your script to print the XML generated by the `pdftoxml` function and search through that.
- Once you know the XML code you can **specify which parts you want to grab with your scraper**, by using functions such as `.attrib` to grab the attributes, or `.findall` and `.xpath` to grab contents of particular tags, **using XPath**. These can then all be placed in a list which you then loop through.
- **Look for what the data you're grabbing has in common**: e.g. its position from the top or left of the page (`top=` or `left=`), its height or width, its font or other properties, the number of characters (length) or position in the running order (which you can specify with an index).
- **Store these properties in a variable** that you can then test.
- Use **if statements** to test against these qualities, and store if there's a match.
- Write if statements **in the same order as the data appears**, to make sure that the first if statement runs first.

---

[10]https://views.scraperwiki.com/run/pdf-to-html-preview-1/

- To avoid non-matches tripping up your scraper (where an if statement is not met) **try initialising your variable with a zero value before the if statement runs**.
- **Create a separate variable that generates a unique ID number for every entry** by initialising it outside the loop and adding 1 to it every time the loop runs.

Tests

- Adapt the scraper so that it grabs codes with less than 4 characters, or 3. What happens?
- Adapt the scraper to grab the value of 'width' rather than 'font'. Can you grab the same data using this approach?
- See if you can work out how to grab alternate items in a list (clue: look up the documentation on the range() function[11] - this can be used to create a range with only odd or even numbers, e.g. range(1,100,2) would go 1, 3, 5 etc.)

---

[11]http://docs.python.org/release/1.5.1p1/tut/range.html

25 Scraper 24: Scraping multiple PDFs

Now we know how to scrape one PDF, we should be able to combine that with knowledge gained on previous scrapers to go through a series and scrape them all.

Our story in this chapter concerns knife crime in London. Data on this is published on the London Metropolitan Police's publication scheme page[1] - right near the bottom at the time of writing.

View the source HTML, search for 'knife crime' and identify the part to grab PDF URLs from. This is the key

---

[1] http://www.met.police.uk/foi/c_priorities_and_how_we_are_doing.htm

line:

```
    <table summary="Corporate level publications:
What our priorities are and how we are doing: Knife
Crime Summaries" class="foidocuments">
```

As in so many previous scrapers, what we need to do is find any lines that look like that, grab the links within them, and loop through each link, grabbing and storing the data inside.

Breaking that process down helps us identify what previous code we can re-use:

1. Find any pieces of HTML that have `<table summary="Corporate level publications: What our priorities are and how we are doing: Knife Crime Summaries"` or indeed `table class="foidocuments">`
2. Grab the `<a href= ...>` link within those tags which should be the URL where each PDF is
3. Loop through each PDF
4. Grab particular pieces of data from it

Task 1 and 2 are processes we've covered previously in Scraper #18 so we can adapt the same code.

Task 3 - well, we've looped through webpages so again we can adapt the same process

And task 4 we covered in the previous chapter.

In short, we understand the component parts of this scraper, and this process is about combining and adapting those.

The code

Here's the code[2] I came up with to do that:

```
#import the libraries we'll need
import scraperwiki
import urllib2
import lxml.etree
import lxml.html
#<table summary="Corporate level publications:
What our priorities are and how we are doing: Knife
Crime Summaries" class="foidocuments">
#This creates a new function to find the part
of the page we want, scrape bits, and follow links
in it.
def scrapetable(root):

   #create an empty variable 'record',
   which is a dictionary

    record = {}

   #create another variable, 'uniqueid',
   set to zero, which will be added to later
   on.

    uniqueid = 0

   #Grab any bits of 'root' (passed into
   'scrapetable' as a parameter above)
```

[2]https://scraperwiki.com/scrapers/pdftoxml_pt4_geocoding/

that have the tag <table> containing
summary=" and 'Knife' somewhere in that,
then the contents of <tr>. Put the
results in variable called 'rows'

```
rows = root.xpath(".//table[contains(@summary,
'Knife')]//tr")
```

#That will be a list, so we start a for
loop to go through each item, calling it
'row'

```
for row in rows:
```

    #show us the text content of
    that item

```
        print row.text_content()
```

#now grab the contents of all <td><a ... > tags
within that 'row' object, and put it in the variable
'report'

```
        report = row.cssselect("td
        a")
```

    #if that exists...

```
        if report:
```

#get the value of the first (index 0) 'href='
attribute and put in 'pdfurl' variable

```
pdfurl = report[0].attrib.get('href')

#store  the  value  of
the first 'href=' and
'title=' attributes with
the  labels  'URL'  and
'Date' in the variable
'record'

record["Date"] = report[0].attrib.get('title')

record["URL"] = report[0].attrib.get('href')

# if the 'pdfurl' variable
was indeed created...

 if pdfurl:
```

```
    #Start running a PDF scraper on that, firstly
'opening' the PDF URL with the urllib2 library's
urlopen function, and 'reading' the PDF into a
variable called 'pdfdata'...
```

```
                    pdfdata = urllib2.urlopen
                    (baseurl+pdfurl).read()
```

```
   #...Then using pdftoxml to convert that into a
variable called 'xmldata'
```

```
                    xmldata = scraperwiki.pdftoxml(pdfdata)
```

```
   #...Then using .fromstring to convert that into
a variable called 'pdfxml'

                  pdfxml = lxml.etree.fromstring(xmldata)

                  print xmldata

   #Use .xpath again to find <text ... top="191">
tags, and <b> tags within those

                   boldtags1  =
                  pdfxml.xpath ('.//text[contains(@top,
                  "191")]//b')

   #Then  store  the  first  [0]  result's  text  in
'Date2'

                  record ["Date2"]
                  = boldtags1[0].text

                  boldtags = pdfxml.xpath
                  ('.//text[contains(@top,
                  "386")]//b')

   #This is the code that the line above is looking
for

                  #<text top="386"
                  left="464" width="75"
                  height="21" font="0"><b>04/09/2012</b></text>
```

```
    #Then store the second [1] result's text in
'Review Date'

                    record ["Review
                   Date"] = boldtags[1].text

                   print record

    #Now we grab all the <text ...> tags, and in the
next line loop through them

                   texttags = pdfxml.xpath
                   ('.//text')

                    for  text  in
                   texttags:

                     left = text.attrib.get('left')
```

```
1          >>>>`#convert the attribute from a strin\
2 g into an integer:`
```

```
                    leftinteger
                    = int(left)

    #If leftinteger is between 96 and 99...
    #see other options at http://stackoverflow.com/questions/
618093/how-to-find-whether-a-number-belongs-to-a-particular-rang
    #Literally: If 96 is smaller than leftinteger,
AND leftinteger is smaller than 99:
```

```
                          if 96 <
                      leftinteger
                      < 99:

                      #Record the
                      text of 'text'
                      (sorry)

                         record
                         ["BOCUname"]
                         = text.text

                         print
                         record
```

   #All the 'if' tests from hereon do similar
things: store data in a particular place based on
its properties

```
                         if 324 <
                      leftinteger
                      < 327:

                         record
                         ["Offences"]
                         = text.text

                         print
                         record

                        if 405 <
                      leftinteger
                      < 408:
```

```
 record
 ["Sanction_-
 detentions"]
 = text.text

 print
 record

 if 481 <
leftinteger
< 484:

 record
 ["Sanction_-
 detention_-
 rate"]
 = text.text

 print
 record

 if 587 <
leftinteger
< 590:

 record
 ["Offences
 FYTD_-
 2011_-
 12"] =
 text.text
```

```
   print
   record

  if 661 <
leftinteger
< 664:

    record
    ["Offences
    FYTD_-
    2012_-
    13"] =
    text.text

    print
    record

  if 713 <
leftinteger
< 716:

    record
    ["Offences
    percentage
    change"]
    = text.text

    print
    record

  if 812 <
leftinteger
< 815:
```

```
  record
  ["Sanction
  detections
  FYTD 2011_-
  12"] =
  text.text

 if 887 <
leftinteger
< 890:

   record
   ["Sanction
   Detections
   FYTD_-
   2012_-
   13"] =
   text.text

 if 943 <
leftinteger
< 946:

   record
   ["Sanction
   Detection
   rate FYTD
   2011_-
   12"] =
   text.text
```

```
                     if 1021 <
                    leftinteger
                    < 1024:

                      record
                      ["Sanction
                      Detections
                      rate FYTD
                      2012_-
                      13"] =
                      text.text

                     uniqueid
                   = uniqueid+1

                    record ["uniqueid"]
                   = uniqueid

                    print record

                    scraperwiki.sqlite.save(["uniqueid","Date"],
                   record)

   #This creates a new function to scrape the
initial page so we can grab report titles and the
links
   def scrape_and_look_for_next_link(url):

     #scrapes the page and puts it in 'html'

      html = scraperwiki.scrape(url)
```

```
 print html

#turns html from a string into an lxml
object called 'root'

 root = lxml.html.fromstring(html)

#runs another function - created earlier
- on 'root'

 scrapetable(root)

#This will be used for relative links in later
pages
baseurl = "http://www.met.police.uk/foi/"
#When added to the baseurl, this is our starting
page
startingurl = "c_priorities_and_how_we_are_doing.htm"
#Run the function created earlier above on that
URL
scrape_and_look_for_next_link(baseurl+startingurl)
```

Tasks 1 and 2: Find a pattern in the HTML and grab the links within

The code towards the end of the scraper (where, with the groundwork done, the scraper actually starts working) is adapted from that earlier code:

```
#This will be used for relative links in later
pages
```

```
baseurl = "http://www.met.police.uk/foi/"
#When added to the baseurl, this is our starting
page
startingurl = "c_priorities_and_how_we_are_doing.htm"
#Run the function created earlier above on that
URL
scrape_and_look_for_next_link(baseurl+startingurl)
```

Now look for that function being defined just above this section of code:

```
#This creates a new function to scrape the
initial page so we can grab report titles and the
links
def scrape_and_look_for_next_link(url):

   #scrapes the page and puts it in 'html'

    html = scraperwiki.scrape(url)

    print html

   #turns html from a string into an lxml
   object called 'root'

    root = lxml.html.fromstring(html)

   #runs another function - created earlier
   - on 'root'

    scrapetable(root)
```

This is pretty much unchanged - we haven't even changed the name of the function, despite the fact that it no longer looks for a 'next' link (everything we want is on this one page).

In fact, as it's not looking for 'next' links we probably don't need a whole new function and could incorporate this into the start of the scrapetable function instead.

But that's for later. For now, it'll do, so we move on to first part of the scrapetable function created earlier in the code:

```
def scrapetable(root):

  #create  an  empty  variable  'record',
  which is a dictionary

   record = {}

  #create  another  variable,  'uniqueid',
  set to zero, which will be added to later
  on.

   uniqueid = 0

  #Grab  any  bits  of  'root'  (passed  into
  'scrapetable'  as  a  parameter  above)
  that  have  the  tag  <table>  containing
  summary=" and 'Knife' somewhere in that,
  then  the  contents  of  <tr>.  Put  the
  results in variable called 'rows'

   rows = root.xpath(".//table[contains(@summary,
  'Knife')]//tr")
```

```
    #That will be a list, so we start a for
    loop to go through each item, calling it
    'row'

     for row in rows:

         #show us the text content of
         that item

          print row.text_content()
```

```
  #now grab the contents of all <td><a ... > tags
within that 'row' object, and put it in the variable
'report'
```

```
            report = row.cssselect("td
        a")

        #if that exists...

         if report:
```

```
  #get the value of the first (index 0) 'href='
attribute and put in 'pdfurl' variable
```

```
            pdfurl = report[0].attrib.get('href')
```

```
  #store  the  value  of  the  first  'href='  and
'title=' attributes with the labels 'URL' and 'Date'
in the variable 'record'
```

```
            record["Date"] = report[0].attrib.get('title')
```

```
        record["URL"] = report[0].attrib.get('href')

    # if the 'pdfurl' variable
    was indeed created...

     if pdfurl:
```

Again, almost all of this is familiar from previous scrapers, and the comments (some of which I've not indented, so that reading is easier) explain it line by line: we are looking for patterns of HTML, grabbing anything that fits those patterns, putting them in a list, and then looking for more patterns within items in that list.

This covers tasks 1 and 2 we identified at the start.

At the end of the section, if it finds any PDF URLs, it then moves into the PDF scraping part of this function - task 3.

But before we look at that, you may have encountered a small piece of code we need to explain:

```
    rows = root.xpath(".//table[contains(@summary,
    'Knife')]//tr")
```

XPath contains...

This line uses **XPath** again, which we reintroduced in the last chapter after first tackling it in scraper #4 with =importXML in Google Docs.

In the previous chapter we used XPath with an XML document created with pdftoxml. Now we are also using it

with an lxml object created with `lxml.html.fromstring`. That object is `root`.

Once again, we're using the `.xpath` method from the `lxml` library[3].

As before, the double slashes indicate a tag, and each new double slash drills down within the first, so that in our XPath we are first looking for `<table>` and then, within that, `<tr>`.

Anything in square brackets specifies something about that particular tag, so `//table[contains(@summary, 'Knife')]` says we want a `<table ...>` tag that contains an attribute of `summary=` (i.e. `<table summary=" ... ">`) for which the value *contains* the string `Knife`.

Here we benefit from earlier experience with XPath, because the `contains` function is not mentioned in the lxml documentation - it was mentioned in the XPath tutorials we referenced alongside Scraper #4 and Scraper #23.

We need this because the HTML isn't very helpful, and the only way to identify our links is to look for summaries that mention 'knife'. This bit of XPath allows us to do that.

The code: scraping more than one PDF

Back to our code, then. This is where we left off:

```
if pdfurl:
```

---

[3]http://lxml.de/tutorial.html#using-xpath-to-find-text

And this is what comes next, if we do indeed find a pdfurl:

```
    #Start running a PDF scraper on that, firstly
'opening' the PDF URL with the urllib2 library's
urlopen function, and 'reading' the PDF into a
variable called 'pdfdata'...
```

```
                    pdfdata = urllib2.urlopen
                    (baseurl+pdfurl).read()
```

```
    #...Then using pdftoxml to convert that into a
variable called 'xmldata'
```

```
                    xmldata = scraperwiki.pdftoxml(pdfdata)
```

```
    #...Then using .fromstring to convert that into
a variable called 'pdfxml'
```

```
                    pdfxml = lxml.etree.fromstring(xmldata)
```

```
                    print xmldata
```

```
    #Use .xpath again to find <text ... top="191">
tags, and <b> tags within those
```

```
                    boldtags1 =
                    pdfxml.xpath ('.//text[contains(@top,
                    "191")]//b')
```

```
    #Then store the first [0] result's text in
'Date2'
```

```
                    record ["Date2"]
                 = boldtags1[0].text

                  boldtags = pdfxml.xpath
                 ('.//text[contains(@top,
                 "386")]//b')
```

```
   #This is the code that the line above is looking
for
```

```
                    #<text top="386"
                 left="464" width="75"
                 height="21" font="0"><b>04/09/2012</b></text>
```

```
   #Then store the second [1] result's text in
'Review Date'
```

```
                  record ["Review
                 Date"] = boldtags[1].text

                  print record
```

```
   #Now we grab all the <text ...> tags, and in the
next line loop through them
```

```
                  texttags = pdfxml.xpath
                 ('.//text')

                   for text in
                 texttags:

                    left = text.attrib.get('left')
```

```
#convert the attribute
from  a  string
into an integer:

  leftinteger
 = int(left)
```

...hold on: what's that?

The wrong kind of data: calculations with strings

As in the previous chapter, most of this code uses the formatting of our data to identify it and then grab it. Any bold text with a top="191" property goes into our `boldtags1` variable, and the first one stored in our `record` variable. Any bold text with a top="386" property goes into the `record` with the label 'Review Date'...

But the rest of the data presents a problem.

On one of the PDFs, the BOCU name (*Borough Operational Command Unit - impress friends and family with your knowledge of British policing jargon!*) has a property of `left="97"`; but in the other it's `left="98"`.

(*This was only discovered by running the scraper once, and then checking the results against the PDFs, and seeing what's missing. Then checking the code against the missing data on the PDF it didn't work on. Trial and error, again.*)

We need our scraper to be able to grab either.

There are a number of ways to do this. We could ask if it's either of two numbers, or in a range, or if it has two digits, and so on.

I've chosen to test if the numbers are within a range, largely because we can change the range more easily than changing the alternatives.

But there's another problem. To demonstrate, try adding the following line after `left = text.attrib.get('left')`:

```
print left+10
```

...and **RUN** the script.

This will generate the following error:

`TypeError: cannot concatenate 'str' and 'int' objects`

If you can't guess what 'str' and 'int' objects are, do a quick search for "str int object python" - but you might be able to guess.

Str is short for string; and int for integer. In other words, we are trying to combine two different types of data: a string of text, and a number.

That's because the '98' in `left="98"` is not a number: it's in quotation marks. As far as this scraper is concerned, it's a string.

So how do we get the scraper to recognise it as a number? We turn to a search engine again, and search for "converting string to integer python".

The answer is a basic python function: `int()` where the parameter (in the parentheses) is what you want to convert, like so:

```
int("98")
```
Try adding
```
print int("98")
```

before your troublesome line of code, **RUN** it, and you should see the results before the error: 98.

And that's exactly what we do in the code, using int with the variable left, and putting the results into a new variable, leftinteger:

```
     leftinteger
   = int(left)
```

(*By the way, you can do the same to convert integers to strings by using* str()*)*

Now that we have a proper integer, and not a string that *looks* like an integer, we can perform some calculations.

Putting square pegs in square holes: saving data based on properties

The next section of code, then, saves the data based on its `<text left="...">` value.

Once again, a quick search helps us find out the code. In this case, a page on programming Q&A site Stack Overflow[4].

---

[4]http://stackoverflow.com/questions/618093/how-to-find-whether-a-number-belongs-to-a-particular-range-in-python

We know that our BOCU code is always left=97 or left=98, the 'Offences' column is left=325 or 326, and so on. With the left value converted into an integer, here's the code that fits the text into the right field depending on the number. Once again, comments explain what's happening:

```
#If leftinteger is between 96 and 99...
#see other options at http://stackoverflow.com/questions/61809
#Literally: If 96 is smaller than leftinteger,
AND leftinteger is smaller than 99:

                 if 96 <
                leftinteger
                < 99:

                #Record the
                text of 'text'
                (sorry)

                  record
                 ["BOCUname"]
                 = text.text

                  print
                 record

   #All the 'if' tests from hereon do similar
things: store data in a particular place based on
its properties

                 if 324 <
                leftinteger
                < 327:
```

```
 record
 ["Offences"]
 = text.text

 print
 record

 if 405 <
leftinteger
< 408:

 record
 ["Sanction_-
 detentions"]
 = text.text

 print
 record

 if 481 <
leftinteger
< 484:

 record
 ["Sanction_-
 detention_-
 rate"]
 = text.text

 print
 record
```

```
  if 587 <
leftinteger
< 590:

   record
   ["Offences
   FYTD_-
   2011_-
   12"] =
   text.text

   print
   record

 if 661 <
leftinteger
< 664:

   record
   ["Offences
   FYTD_-
   2012_-
   13"] =
   text.text

   print
   record

 if 713 <
leftinteger
< 716:
```

```
 record
 ["Offences
 percentage
 change"]
 = text.text

 print
 record

 if 812 <
leftinteger
< 815:

 record
 ["Sanction
 detections
 FYTD 2011_-
 12"] =
 text.text

 if 887 <
leftinteger
< 890:

 record
 ["Sanction
 Detections
 FYTD_-
 2012_-
 13"] =
 text.text
```

```
 if 943 <
leftinteger
< 946:

  record
  ["Sanction
  Detection
  rate FYTD
  2011_-
  12"] =
  text.text

 if 1021 <
leftinteger
< 1024:

  record
  ["Sanction
  Detections
  rate FYTD
  2012_-
  13"] =
  text.text
```

The final four lines, once all those `if` statements have been run, adds 1 to the `uniqueid` variable, then saves it, prints the whole record, and saves the lot - with `uniqueid` and `Date` as the unique keys

```
    uniqueid
= uniqueid+1
```

```
 record ["uniqueid"]
= uniqueid

 print record

 scraperwiki.sqlite.save(["uniqueid","Date"],
record)
```

Once it's saved this record, it loops again, and again, and again, for each row.

Note that many lines earlier it has also saved the URL of the PDF, the date and the review date. Although these are grabbed only once, they are saved over and over again with each new row. This is important because it allows us to separate these records from the records in other PDFs with different URLs for different months.

Finally, the *whole thing runs again* for any further PDFs. We only have two, which means you can see the results quickly. But the same script would still work if another 100 PDF links were added to the webpage (assuming formatting of those links remained consistent). The key is to check that you're getting data for all the PDFs you're expecting, and that data matches up.

Recap

- **Break down what you need your scraper to do, into separate tasks** so you can more easily identify what code is needed for each. Some people write this as comments at the start of the scraper.

- **Re-use code from previous scrapers** - it isn't cheating. It's being quick and efficient.
- **There's more than one way to skin an onion**. You can find patterns using cssselect, or XPath, or regex, or something else entirely. You can use various calculations. Use whatever you're comfortable with.
- **Record different bits of data at different points of the scraping process**. Our scraper, for example, first stores the URL and date of the PDF (from the webpage), then two more dates (from the first lines of the PDF), a unique ID (initialised at a point so that it won't ever be reset by a loop), and the contents of each row of data.
- **Use the `int` function to convert something into an integer**, so `"98"` for example becomes simply `98`. You can also use `str` to do the reverse, and convert an integer into a string.
- **Use `if` statements to store data based on its position in the PDF**. A header row might all share the same position from the top, for example; one column will tend to be the same position from the left.
- When scraping more than one document, **widen your tests so that it can accept some slight variation in position between documents** - without catching the wrong columns of data.

Tests

- See if you can rewrite the code so that it tests the

`leftinteger` or `left` variables in *different* ways with the same result.

- Search for *other ways to convert variables between different types* (as well as integers and strings there are floats, lists, and dictionaries, for example), or test them.

- Add some code which adds the string "NO DATA!" if data isn't found so you can easily spot it when the scrape is finished. You might want to create a new `if` statement, or an `elif` or `else` statement, or change some existing `if` statements to `elif`, and/or a new field in your `record`. Be prepared to use trial and error (for example, you don't want it to record 'NO DATA' if the data should be saved in another column - think about where in your script you might want to put your new line(s)).

- Try to create some variables up front in your code to store the positions of the various data fields, and use those instead of numbers in the lines that test for text position (i.e. the lines that read `if 1021 < leftinteger < 1024:`). This might sometimes be preferable if you're going to need to change those values. For example you could do the following: `firstcol = 98` and `secondcol = firstcol+30` and so on. If your positions change or you need to adapt your code during testing, you then only need to change one line instead of several.

- Find another set of PDFs to scrape (try an advanced

search that uses site:gov.uk[5] or site:police.uk[6] or a relevant domain type for your country or field, e.g. site:gov.it for Italian government sites[7]. Map out the tasks your scraper needs to complete - what new problems do they raise? Can you search for code or tutorials that helps address those problems? If you want a real challenge, try these PDFs of stop and search data in London[8].

---

[5] https://www.google.co.uk/search?q=site:gov.uk+reports
[6] https://www.google.co.uk/search?q=site:police.uk+reports
[7] https://www.google.co.uk/search?q=site:gov.it+rapporti
[8] http://www.met.police.uk/foi/units/stop_and_search.htm

26 Scraper 25: Text, not tables, in PDFs - regex

Our adventures with PDFs so far have involved grabbing data from tables - but what if we are dealing with reports and passages of text?

Scrapers can be enormously helpful here: they can scour through thousands of documents looking for mentions of particular people, organisations, phrases or terms.

In this chapter we are going to do just that: use a scraper to go through a number of reports on one police force's

use of stop and search powers[1] (the power to stop and search individuals) and see how often they mention issues around over-use of those powers to stop people from ethnic minorities. More specifically, we'll also grab the context in which those mentions were made.

In these cases, the structure is not in the position of the data, but in the text itself: *the structure of a series of characters.*

And instead of grabbing all the data in all the reports, in these cases we generally only want to grab data - text - when it occurs near to that person, organisation, or term of interest.

But the tools we have used in Scraperwiki so far are designed to find structure in code: `.cssselect` and `.xpath` both look at HTML tags and their properties.

We need to turn to something we used with text before, back in the chapters on OutWit Hub: **regex**.

Starting the code: importing a regex library

In OutWit Hub, regex is built in. In Scraperwiki, however, you need to specifically *import* it - just as you've had to import the `scraperwiki` library, `lxml.html`, and others.

Python's regex library is called `re`. Our code begins by importing that and some other more familiar libraries:

```
#import the libraries we'll need
```

---

[1]http://www.dpa.police.uk/default.aspx?page=473

```
import re
import scraperwiki
import urllib2
import lxml.etree
import lxml.html
```

Note: normally we would not start writing our code with all of these - we would add them as we hit particular problems which require those libraries.

Here are the tasks we need our scraper to perform:

1. Find all the links to PDF reports on a particular webpage
2. Look in each report for any mention of the words 'black' or 'ethnic'
3. Store details on the lines containing those mentions, as well as the URL of the report, date, etc.

Task 1 should be pretty familiar by now, as should the code:

Code continued: Find all the links to PDF reports on a particular webpage

The page we're scraping is on the Dorset Police website[2]. It contains links to reports on stop and search going back to 2006.

---

[2]http://www.dpa.police.uk/default.aspx?page=473

The links to each PDF look like this in the raw HTML:

```
<p><a onclick="window.open(this.href, '_blank');
return false;" title="Link to the October 2012 Stop
and Search update - link opens in new window"
href="/PDF/PSD_031012_04_Stop_and_Search.pdf" onkeypress="if
(event.keyCode==13) {window.open(this.href, '_blank');
return false;}">Stop/Search and Stop/Account Update
- October 2012</a> &gt;</p>
```

This is quite imposing - it will be easier to scrape if we break it down:

```
<p>
<a
onclick="window.open(this.href, '_blank'); return
false;"
title="Link to the October 2012 Stop and Search
update - link opens in new window"
href="/PDF/PSD_031012_04_Stop_and_Search.pdf"
onkeypress="if (event.keyCode==13) {window.open(this.href,
'_blank'); return false;}">
Stop/Search and Stop/Account Update - October
2012</a> &gt;</p>
```

What we have, then, is a ‹p› tag containing an ‹a› tag, which contains some text. The ‹a› tag has a number of properties, such as onclick=, title=, href=.

We could try any of these patterns to identify our links. But which is most likely to be unique to the ones we want, rather than others?

It's always best to start simple rather than trying to be over-complex. Here's a list of possibilities:

- Grab all `<a>` links
- Grab all `<a>` links within `<p>` tags
- Grab all `<a>` links within `<p>` tags with a `title=` attribute that contains 'Stop and Search'
- Grab all `<a>` links within `<p>` tags with a `href=` attribute that ends '.pdf'
- Some combination of the above

Looking at our page we can immediately rule out the first option, as there are other links on this page. But these all sit in menus, so it may be worth trying the second:

- Grab all `<a>` links within `<p>` tags

Here's the code to do it:

First, we create a new function to scrape the initial page so we can grab report titles and the links. Only once that function has been created can we write the extra code to run that function on our webpage - so follow this code to the end to see that happening.

Once again, we're adapting code from earlier, and I've left the function name the same so you can see (even though this function doesn't look for a 'next' link):

```
#define our function. It has one parameter which
it names 'url'
def scrape_and_look_for_next_link(url):

    #scrapes the page and puts it in 'html'

    html = scraperwiki.scrape(url)
```

```
 #turns html from a string into an lxml
object called 'root'

 root = lxml.html.fromstring(html)

 #grabs anything in 'root' within the
tags <p><a>, and puts them into a new
variable 'pdflinks'

 pdflinks = root.cssselect("p a")

#that will be a list of results, so we
need to loop through them with a 'for'
loop

 for link in pdflinks:

     #This just shows us the href=
     attribute of each item ('link')
     in that list

      print link.attrib.get('href')
```

This particular line of code helps us write the rest. It shows us, for example, whether the links are relative or absolute. In this case, they're relative (e.g. /files/file.pdf), so we need to add them to the base URL (e.g. http://site.com).

That also means that we need to specify what the base URL *is*. We do that *before the function runs* but *after it's written*. Here's why: we write our function first so that when we want to run it, it's ready (otherwise the computer will say 'this function does not exist').

Here's how the script runs step-by-step:

1. Import our libraries (otherwise functions from those libraries won't work)
2. Define our functions (so they're ready to be used)
3. Set any variables that will be needed by the whole script (such as the base URL of the site we're scraping). These are called **global variables**.
4. Run the functions using those variables

You can see that in order for the final step to work we must have *both the functions and the variables ready to go.*

Back to our code, then. Remember that at this point we have grabbed all links within ‹p›‹a› and put them in a list, and we're now looping through each item in that list.

We printed the link so we can see what it looks like. Now we add that link to the baseurl variable - which will be defined, as we explained, later, before the function runs. The result of that, is put into a new variable: 'next_link_absolute'

```
next_link_absolute = baseurl+link.attrib.get('href')
```

```
   #this line has 'is not None' added as future
versions won't accept 'if link'
```

```
        if link is not None:

            scrapepdf(next_link_-
            absolute)
```

The final two lines here check that the link exists (`is not None`), and if it does indeed exist, runs a new function, `scrapepdf` on the URL that adds that link to the base URL.

Now we need to make sure that *that* function is written before *this* function runs, so it's ready to go. We'll come back to that in a moment.

First, with that function created, here are the lines of code *after* it that create those **global variables**, and run the function.

```
#This could be used for relative links in later
pages
baseurl = "http://www.dpa.police.uk"
#When added to the baseurl, this is our starting
page: http://www.dpa.police.uk/default.aspx?page=473
startingurl = "/default.aspx?page=473"
#Run the function created earlier above on that
URL
scrape_and_look_for_next_link(baseurl+startingurl)
```

Detour: global variables and local variables

Before we go any further, I should explain this term '**global variable**', which hasn't been used before.

A global variable is a variable which can be used by any part of your script. The term makes most sense when contrasted against the alternative: a **local variable**.

A **local variable** can only be used within a particular function.

For example, in our scraper, the URL of the website homepage might be needed by *any* function: the function that scrapes our first page and then finds PDF links (or 'next' links in another scraper) will need to know that URL if those links are only relative ones.

That URL, then, needs to be put into a **global variable** at some point (preferably early on, before those functions run) so that it can be accessed.

However, other information might only be needed by *one* of our functions. For example, a function that is saving data from each PDF in turn does not need to store that data in a global variable because no other function will need it. So it creates a **local variable** within that function.

There are, of course, ways to pass information between functions even if it is stored in a local variable. Our first function, for example, calls a second function at the end of our code above, and 'passes' the *local* variable `next_link_-absolute` at the same time:

```
scrapepdf(next_link_-
absolute)
```

When that next function runs, it takes the contents of that local variable, and puts it in another *local* variable (local to the new function). That might happen to have the same name as the previous variable, but it is still a local variable.

We can also use `return` at the end of a function to return that value of a local variable to whatever function

called this one (as we did in scraper #22: Scraping PDFs part 1).

But don't get too hung up on all that if it doesn't make sense in the abstract: when you come across a concrete problem with local variables, just come back here to remember how it all works - and as always, explore the documentation or search for other examples of code to find possible solutions.

In the meantime, this is what you need to know: **local variables** only exist while a function is running; and **global variables** can be used by any of your functions.

The code part 3: scraping each PDF

We've now created the code for our first task:

1. Find all the links to PDF reports on a particular webpage

Now we need to create that new function which performs the next two tasks:

1. Look in each report for any mention of the words 'black' or 'ethnic'
2. Store details on the lines containing those mentions, as well as the URL of the report, date, etc.

This function needs to be created *before* `scrape_and_look_for_next_link` runs because it will be needed at the end of that. Here's the code that begins the new function:

```
    def scrapepdf(url):
   #use the urllib2 library's .urlopen function to
open the full PDF URL, and the .read() function to
read it into a new object, 'pdfdata'

     pdfdata = urllib2.urlopen(url).read()

    #use pdftoxml to convert that into an
    xml document

     pdfread = scraperwiki.pdftoxml(pdfdata)

     print pdfread

    #use lxml.etree to convert that into an
    lxml object

     pdfroot = lxml.etree.fromstring(pdfread)

    #find all <text> tags and put in list
    variable 'lines'

     lines = pdfroot.findall('.//text')

     #create variable 'linenumber', initialised
    at 0

     linenumber = 0

     record = {}

     #loop through each item in 'lines' list
```

```
for line in lines:

     #add one to 'linenumber' so
    we can track which line we're
    dealing with

     linenumber = linenumber+1

     #if 'line' has some text:

     if line.text is not None:

   #create a new variable 'mention' that is filled
with the result of using the 're' library's .match
function

                mention = re.match(r'.*black.*',line.text)
```

Most of this code is adapted from earlier scrapers: we take a URL at the start, 'read' it into a new variable, convert it into XML, and find particular things in it. This line, for example, uses XPath to find all lines with the tag ‹text›:

```
lines = pdfroot.findall('.//text')
```

We also, as we loop through those lines of text, keep track of the line *number*:

```
linenumber = linenumber+1
```

The reason for doing this is that we'll need to grab the lines before and after our target text, to put it into context. And in order to do *that*, we'll need to know which line it's in.

The really new line - the focus of this whole chapter - is the one that uses regex:

```
mention = re.match(r'.*black.*',line.text)
```

If we didn't know that, we could have found it by searching for `re` - the regex library we mentioned earlier.

Re: Python's regex library

Because `.match` is attached to `re`, that means it is a function from that library. If we wanted to find out more we could look for documentation by searching for "regex match function python" or similar - and *then* searching within one of the pages that we find[3] for the exact piece of code "re.match".

This would give you some example code to try to customise, and jargon to decode. If you prefer, however, here are the key things you need to know in plain English:

Firstly, the `.match` function requires two parameters: the regular expression, and where you are looking for it.

The regular expression needs to:

- Include `r` before it starts, e.g. `r'.*black.*'`

---

[3]http://docs.python.org/2/howto/regex.html

- Be contained in inverted commas (or straight quote marks)

In case you need reminding, our particular regex can be understood as follows:

- The full stop means 'any character'
- The asterisk means 'none or more' of that 'any character'
- The string of characters 'black' needs to be matched exactly
- And then we get another full stop and asterisk, meaning 'none or more' of 'any character'

After the end of our regex, we have a comma, and then *where you are looking for that regular expression*: in this case the text of line, or: `line.text`.

If it helps, you can also test the process by using a plain string of text rather than a variable, e.g. "Pot calling the kettle black".

Other functions from the `re` library

If you read some of the documentation on re[4] you'll find extra instructions on how you can use it.

For example, we've used the `.match` function, but it's worth noting the following passage[5]:

---

[4]http://docs.python.org/2/howto/regex.html
[5]http://docs.python.org/2/howto/regex.html#match-versus-search

"The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

"On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

"Sometimes you'll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead [because this will result in faster code]."

Oops. So we should probably be using `.search`. I'll leave that for the tests later...

Some other useful advice in that documentation includes *compiling* regular expressions and putting them in variables so you can use them more than once without having to type the expression more than once. Have a play[6].

Back to the code

We left off from the code where we were using regex to test if a line of text contained a string of characters: "black". The

---

[6]http://docs.python.org/2/howto/regex.html

next passage of code only runs if it *does*:

```
if mention:

    print line.text
```

```
#the RANGE function generates a list from the
first parameter to the second, e.g. range(5,8) would
make [5, 6, 7] - it doesn't include the 'end' of
the range. In this case we're using the line number
minus 2, and the linenumber as our start and end
points
```

```
print range(linenumber-2,linenumber+1)

 linebefore =
"EMPTY LINE"

 lineafter =
"EMPTY LINE"

 incontextlist
= []

 if pdfroot.xpath('.//text')[linenumber-2].text

  linebefore
 = pdfroot.xpath('.//text')[linenumber-2].text

  incontextlist.append(linebefore)

 incontextlist.append(pdfroot.xpath('.//text')[
```

```
                  if pdfroot.xpath('.//text')[linenumber].text
              is not None:

                 lineafter
                = pdfroot.xpath('.//text')[linenumber].text

                 incontextlist.append(lineafter)

               record["mention
              in context"] =
              ''.join(incontextlist)

               record["linenumber"]
              = linenumber

   #this  stores  the  'url'  variable  which  is
passed right at the start of this function: def
scrapepdf(url):

                 record["url"]
                = url

                print record

                scraperwiki.sqlite.save(["linenumber",
                "url"],record)
```

Now you can see why we were storing line numbers -
and what we're going to do with them.

When we find any mention of "black" in the stop and
search report, we not only want to grab the line containing
that mention, but also the lines before and after.

Initially you might write some simpler code that does just that - but you will hit a problem: sometimes there *is no line before*, or no line after. So I've added some lines which test that first before adding the extra lines. They look like this:

```
if pdfroot.xpath('.//text')[linenumber-2].text
```

This requires some breaking down because, although at this point we've been focusing on one particular line, here we break away from that and take a fresh look at the PDF report as a whole.

`pdfroot`, remember is our lxml 'version' of the *whole PDF* created with the line `pdfroot = lxml.etree.fromstring(pdfread)`.

We use the `.xpath` method on that to find anything in the tag `<text>`...

And specifically, we want the one with the index of `linenumber-2` (we know it's an index because it's in square brackets immediately after (`'.//text'`)).

At this point, `linenumber` (which goes up by 1 for every line the code looks at, remember) is the number of the line containing our 'match' for "black". We don't want that line (yet), but the line before.

So why not use `linenumber-1`? Well, our *lines* start counting at 1 (`linenumber` is initialised at 0 and 1 is added to it every time the loop runs through a line) - but an *index* starts at 0.

So our *first* line sets the `linenumber` variable at 1, but if we wanted to use an index, we would use 0.

We could solve this by changing the line that says linenumber = 0 to linenumber = -1. Or we could just remember to use an index which is one less than the linenumber we want.

This is probably the worst of the two - but I'm using it here so I can explain what's going on at the point where it makes most sense.

So, in short, we subtract 2 from the line number to get the *index* of the line before.

If that line does exist and has text (the .text at the end), then we put the text of the line in a new variable, linebefore.

And we then **append** the value of that variable in another variable called incontextlist.

Appending means to add an item to a list variable. It's done here with the .append method which works like this: listname.append(thingtobeadded). Or, specifically in this case: incontextlist.append(linebefore)

Just in case these variables don't exist (and the list won't yet), we initialise them before this if statement a few lines earlier:

```
 linebefore =
"EMPTY LINE"

  lineafter  =
"EMPTY LINE"

 incontextlist
= []
```

Note that the last variable is an empty list.

We repeat the process for the *index* of the main line containing our regex match (`linenumber-1`) - without an `if` statement because we know it exists:

```
incontextlist.append(pdfroot.xpath('.//text')[linenumber-1].t
```

And once more for the line after it.

And with those 3 lines all stored in a list, we can start saving our data in `record`:

But unless we want a list in our data (which would look like this: `["some text", "more text", "more text"]`), we need to join that list of lines back together.

We can do that with the `.join` method.

Joining lists of items into a single string

The `.join` method is very similar to functions like `=CONCATENATE` in Excel and Google Docs. It is a **string method**, which means it works on a string.

Taken in isolation, `.join` seems a bit back-to-front. Instead of saying 'I want to join all the items in this list, with spaces in between', it says 'I want to join a series of spaces, with items from a list placed between'.

Here's how we join our list, for example:

```
joinedlist = ''.join(incontextlist)
```

First comes the string: `''`

(In other words, nothing. If we wanted to put a space between each item, we would use `' '`).

Next comes the `.join` method being used on that string.

And finally comes the parameter, our list: (`incontextlist`)

By joining our items together, we create something which is no longer a *list*, but a **string**.

Instead of storing that string in a variable, here we just store it directly in `record` under the label "mention in context":

`record["mention in context"] = ''.join(incontextlist)`

But if it makes more sense to you, add an extra line to put the string in a variable first, before storing it in a further line.

The final few lines just store some other information, such as the linenumber and URL, which we will use together as the unique key:

```
record["linenumber"]
= linenumber
```

```
#this stores the 'url' variable which is
passed right at the start of this function: def
scrapepdf(url):
```

```
record["url"]
= url
```

```
print record
```

```
scraperwiki.sqlite.save(["linenumber",
"url"],record)
```

Now we've completed tasks 2 and 3:

- Look in each report for any mention of the words 'black' or 'ethnic'
- Store details on the lines containing those mentions, as well as the URL of the report, date, etc.

...and we can now review the code in full.

The code in full

```
#import the libraries we'll need
    import re
    import scraperwiki
    import urllib2
    import lxml.etree
    import lxml.html
    def scrapepdf(url):
    #use the urllib2 library's .urlopen function to
open the full PDF URL, and the .read() function to
read it into a new object, 'pdfdata'

     pdfdata = urllib2.urlopen(url).read()

    #use pdftoxml to convert that into an
    xml document

    pdfread = scraperwiki.pdftoxml(pdfdata)

    print pdfread
```

```
#use lxml.etree to convert that into an
lxml object

 pdfroot = lxml.etree.fromstring(pdfread)

#find all <text> tags and put in list
variable 'lines'

 lines = pdfroot.findall('.//text')

 #create variable 'linenumber', initialised
at 0

 linenumber = 0

 record = {}

 #loop through each item in 'lines' list

 for line in lines:

     #add one to 'linenumber' so
    we can track which line we're
    dealing with

     linenumber = linenumber+1

     #if 'line' has some text:

     if line.text is not None:
```

```
    #create a new variable 'mention' that is filled
with the result of using the 're' library's .match
function

                mention = re.match(r'.*black.*',line.text)

                if mention:

                    print line.text

    #the RANGE function generates a list from the
first parameter to the second, e.g. range(5,8) would
make [5, 6, 7] - it doesn't include the 'end' of
the range. In this case we're using the line number
minus 2, and the linenumber as our start and end
points

                    print range(linenumber-2,linenumber+1)

                    linebefore =
                "EMPTY LINE"

                    lineafter =
                "EMPTY LINE"

                    incontextlist
                = []

                    if pdfroot.xpath('.//text')[linenumber-2].text

                        linebefore
                    = pdfroot.xpath('.//text')[linenumber-2].text
```

```
                   incontextlist.append(linebefore)

                  incontextlist.append(pdfroot.xpath('.//text')[

                  if pdfroot.xpath('.//text')[linenumber].text
                is not None:

                    lineafter
                   = pdfroot.xpath('.//text')[linenumber].text

                    incontextlist.append(lineafter)

                 record["mention
                in context"] =
                ''.join(incontextlist)

                 record["linenumber"]
                = linenumber

    #this  stores  the  'url'  variable  which  is
passed right at the start of this function: def
scrapepdf(url):

                  record["url"]
                = url

                 print record

                 scraperwiki.sqlite.save(["linenumber",
                "url"],record)
```

```
    #define our function. It has one parameter which
it names 'url'
    def scrape_and_look_for_next_link(url):

      #scrapes the page and puts it in 'html'

      html = scraperwiki.scrape(url)

      #turns html from a string into an lxml
     object called 'root'

      root = lxml.html.fromstring(html)

       #grabs anything in 'root' within the
     tags <p><a>, and puts them into a new
     variable 'pdflinks'

      pdflinks = root.cssselect("p a")

     #that will be a list of results, so we
     need to loop through them with a 'for'
     loop

      for link in pdflinks:

          #This just shows us the href=
          attribute of each item ('link')
          in that list

           print link.attrib.get('href')

           next_link_absolute = baseurl+link.attrib.get('href')
```

```
    #this line has 'is not None' added as future
versions won't accept 'if link'

          if link is not None:

              scrapepdf(next_link_-
              absolute)

    #This could be used for relative links in later
pages
    baseurl = "http://www.dpa.police.uk"
    #When added to the baseurl, this is our starting
page: http://www.dpa.police.uk/default.aspx?page=473
    startingurl = "/default.aspx?page=473"
    #Run the function created earlier above on that
URL
    scrape_and_look_for_next_link(baseurl+startingurl)
```

Recap

- When scraping text-based documents, the structure is not in the position of the data, but in the text itself: you are looking for **the structure of a series of characters**.
- You can use scrapers to **identify which documents contain those characters**, **and what context they occur in**.
- If you're looking for a **regular expression** - a string of characters that follows a particular pattern - import Python's regex library, re.

- When looking for any sort of match it's always best to **start simple rather than trying to be over-complex**. Sometimes 'all links within `<p>` tags' will be enough.
- **Global variables** can be used by any function in the scraper; **local variables** only work within one function - although you can *pass* these into new variables in other functions when you call them, and *return* them back at the end.
- Try recording the number of the line where you get a match, so you can grab the lines around it by **using that number to generate an index position**.
- But be aware that **you will need to count your first line as 0, not 1**, to reference them directly with an index. Otherwise you'll have to subtract 1 from your line number to get its index.
- You can **use the `.append` method to add items to a list** as you find them.
- And you can **use the `.join` method to join items from a list** into a single string.

Tests

Lots of things to explore at the end of this chapter - in particular the possibilities of regex, and adding to and converting lists. Here are some things to try:

- Alter the code so that the `linenumber` variable is first initialised at -1 and when the same variable

is used to generate indexes, the right calculation is made (i.e. `linenumber-2` becomes `linenumber-1` and so on). Does it still work? Compare the results with the PDFs to see.

- Look at the documentation for the string method `.join` and other string methods[7], and search for other useful tutorials. List each method and write down how they might be useful in dealing with documents you might be scraping.

- Look at the documentation for list methods such as `.append`[8] - and search for other resources.

- Explore the differences between `.match` and `.search` in regex[9] - and change the code to use `.search`. Does it still work? Can you simplify it?

- Our scraper currently only looks for the word 'black'. Can you change the regex so that it also looks for the word 'ethnic'?

- If we looked for a phrase rather than a single word, we might not find it - because the phrase might be split across multiple lines. How would you try to solve that problem?

- Look at other documentation and resources around the `re` library[10] - what other methods and functions might you be able to use?

---

[7]http://docs.python.org/2/library/string.html
[8]http://docs.python.org/2/tutorial/datastructures.html
[9]http://docs.python.org/2/howto/regex.html#match-versus-search
[10]http://docs.python.org/2/howto/regex.html
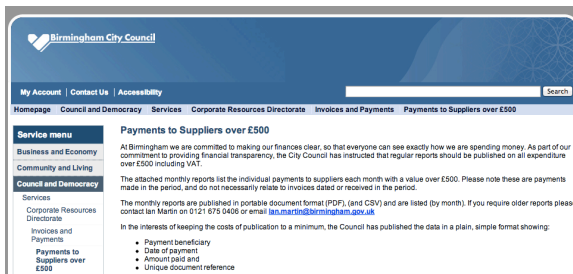
- See if you can *compile* the regular expression in a variable and use it that way.
- See if you can simplify or change the code in other ways - for example: at one point we create a **range** of line numbers but don't do anything with it. Could you loop through that range and do something with each number?

27 Scraper 26: Scraping CSV files

PDFs aren't the only documents we might want to scrape. The data we want may be published in a series of spreadsheets.

But if they're in spreadsheets already, why would we want to scrape them?

Well, if it was just one spreadsheet, we wouldn't. But if there are dozens, or thousands, we might want to save a lot of manual downloading and combining by writing a scraper to do all that for us.

In this case we're going to use a scraper to save us a lot of manual work in downloading and combining local government spending data. We'll also do some analysis on the fly to save us a little time in Excel too.

The specific data we're going to look at is from the

biggest local council in Europe: Birmingham City Council. Like almost all local authorities in England, they publish monthly data on spending over £500[1].

Because the data is in CSV format, we're going to need the right library to scrape it.

The CSV library

The csv library (or "module" according to the documentation[2]), is unsurprisingly, designed to read (and write) data in CSV format.

Scraperwiki has a useful CSV reading guide[3] which explains how to use csv in a scraper. It is encouragingly short: there are only a few lines which you need to use. However, in this chapter we're going to face some common problems which will require us to add some more code.

Based on that Scraperwiki guide, we'll need to start by seeing if we can scrape a *single* CSV from our council spending data. The key points to look out for are:

- We need to import the csv library
- We need to use the .reader function and .splitlines() method to put our scraped data into an object that csv can work with.
- We need to loop through that object and store the data in it.

---

[1] http://www.birmingham.gov.uk/payment-data
[2] http://docs.python.org/2/library/csv.html
[3] https://scraperwiki.com/docs/python/python_csv_guide/

Here is the code[4]:

```
import scraperwiki
import csv
#scrape the csv file into new variable 'data'
data = scraperwiki.scrape('http://www.birmingham.gov.uk/cs/Sat
over_%C2%A3500_October_2012.csv')
#use .reader function and .splitlines() method
to put 'data' into csv object 'reader'
reader = csv.reader(data.splitlines())
print reader
record = {}
for row in reader:

    record['Vendor'] = row[0]

    record['Vendor Name'] = row[1]

    record[' Invoice Amount '] = row[2]

    record['Payment Date'] = row[3]

    record['Doc Number'] = row[4]

    record['Invoice Ref'] = row[5]

    record['Cost Cente'] = row[6]

    record['Directorate'] = row[7]

    print record
```

[4]https://scraperwiki.com/scrapers/birminghamcouncilexpenditure/

```
    scraperwiki.sqlite.save(['Doc Number'],
    record)
```

If you click **RUN** on this code, the scraper will scrape the first row of data (the header row) and then generate the following error:

`SqliteError: Binary strings must be utf-8 encoded`

Scroll up a few lines before that error to see what *did* work, and *where* the error was generated. This provides some useful clues:

- The line `print reader` works: we get a csv *object*: `<_csv.reader object at 0x25a08a0>`
- The `print record` line works on the first loop (the first row - containing our headers) too: `{'Invoice Ref': 'Invoice Ref', 'Directorate': 'Directorate', 'Vendor': 'Vendor', ' Invoice Amount ':...more`
- And on the second loop (the second row - the first one containing our data): `{'Invoice Ref': 'J875431', 'Directorate': 'Adults & Communities', 'Vendor': '1700102486', ' Invoice ...more`
- Then the line generating the error: `Line  24  - scraperwiki.sqlite.save(['Doc Number'], record)`

So we know that the first row of data - the headings - causes no problems. But the second row - the data proper - does.

How is that row different? Well, in quite a few ways. It contains dates, for example, and numbers, and currency. Any could be causing the problem. How do we work it out?

Process of elimination 1: putting blind spots in the code

We could eliminate the suspects by changing one of your lines so that it doesn't actually cause a problem, like so:

```
record['Vendor'] = "NOT BEING SCRAPED"

record['Vendor Name'] = row[1]
```

RUN the code once more, and if it still generates the error, revert that line to the way that it was, and change the next one, like so:

```
record['Vendor'] = row[0]

 record['Vendor  Name']  =  "NOT  BEING
SCRAPED"
```

Repeat this process until the code runs without error (*if this doesn't happen it may be more than one line, in which case, try inverting the process so that your scraper is only working on one line at a time, instead of all-but-one*).

When it does run without error, you have identified your culprit.

In this case, the culprit is the column *Invoice Ref*, and it's pretty easy to guess why: the column contains amounts of money, each of which starts with a pound sign.

How do we know for sure?

Process of elimination 2: amending the source data

Well, it's not just the code that we can tweak. We can also tweak the data it's scraping. How? By copying the data into a Google spreadsheet, publishing it as a CSV, making our tweaks and trying to scrape *that*.

To do this, you'll need to publish your Google spreadsheet as a CSV.

Once you've copied some data from the CSV that you originally wanted to scrape, and pasted it into the Google spreadsheet, go to **File > Publish to the web**... and on the window that appears click **Start publishing**.

This will activate the bottom half of the window. In that section change the drop-down menu that says *Web page* to **CSV (comma-separated values)**.

The bottom box on the window should now show a URL ending in `&output=csv` - this URL is the one you can now try to scrape (just replace the URL after `data = scraperwiki.scrape('` with your new URL).

In this case, our hypothesis is that the pound sign is causing the problem. So remove the pound sign from the second row (after the heading row), save, and then try to run your scraper with the Google spreadsheet URL instead of the original council spending URL.

It should now work - until it hits the next pound sign.

Great. So we know what the problem is. What about a solution?

Encoding, decoding, extracting

There are actually at least two possible solutions, and I'm going to explain both.

The first solution is to directly *tackle* the problem our error is telling us about:

`SqliteError: Binary strings must be utf-8 encoded`

In other words, we can try to change our data so it is *not* "utf-8 encoded".

(*By the way, you could also try to find out more about UTF-8 encoding -* this documentation is one of the clearer explanations around[5] *and* this site tackles UTF-8 and a whole lot of other things to know about strings in Python[6] *- but all you need to know is that UTF-8 is just a way of encoding characters so that computers can understand them.*)

The second solution is to *remove* the problematic data: in other words, that pound sign.

After all, we don't need the pound sign: we only need the amount *after* the pound sign.

If we choose the first solution - trying to change our data so that it *is* UTF-8 encoded - we will, once again, need to do a bit of digging around online to find code we can borrow. This actually turns out to be more time-consuming than it should be.

For example, searching the documentation for csv[7] for

---

[5]http://docs.python.org/2/howto/unicode.html

[6]http://getpython3.com/diveintopython3/strings.html

[7]http://docs.python.org/2/library/csv.html

"UTF" will bring up a number of lines of code to create a `unicode_csv_reader` - but let me save you some time: it doesn't work here.

Another line of enquiry might be a search for "encoding UTF-8 python". This will lead you to the method `.encode` - specifically `.encode("utf-8")`

We can add this to our lines of code[8] to encode each line like so:

```
record[' Invoice Amount '] = row[2].encode("utf-8")
```

However, this generates a new error when it hits the pound sign:

UnicodeDecodeError: 'ascii' codec can't decode byte 0xa3 in position 1: ordinal not in range(128)

But this does help with our search for a solution.

Eventually, with a search for that error message - "UnicodeDecodeError: 'ascii' codec can't decode byte 0xa3 in position 1: ordinal not in range(128)"[9] - we stumble across this thread on Stack Overflow[10] which finally gives us some code that works.

It seems the solution is *not* to try to encode our data as UTF-8, but to *decode* it from Latin-1:

```
record[' Invoice Amount '] = row[2].decode("latin-1")
```

Well, whatever floats your boat.

Meanwhile, long before then, we might have decided to try the other, rather simpler, solution: to get rid of those

---

[8]https://scraperwiki.com/scrapers/birminghamcouncilexpenditure_2nd/
[9]https://www.google.co.uk/search?q=UnicodeDecodeError%3A+'ascii'+codec+can't+decode+byte+0xa3+in+position+1%3A+ordinal+not+in+range
[10]http://stackoverflow.com/questions/3479961/python-csv-unicodedecodeerror

pesky pound signs altogether.

What we want to do here is *store all characters after the first character* (the pound sign) in our data.

If you remember the section from Scraper #22 on indexes, ranges, and slicing shortcuts, you may have worked out the solution...

Chances are, however, that you don't - which is fine. To be honest, neither did I. I just searched for "removing the first character of a string Python".

The answer is to use an index and the colon symbol to specify a range of characters in your string like so:

```
 record[' Invoice Amount '] = row[2][1:]
```

Because row is a list of items, the first index [2] means the third item in that list. Once we've selected that, the [1:] at the end there selects all characters in *that* item, from the second character (index 1) to (the colon sign signifies a range) the last (if there is no number after the colon, Python assumes you mean 'to the end').

In fact, a bit of trial and error with this and the print command - i.e.:

```
print row[2][1:]
```

Leads us to change it slightly to:

```
 record[' Invoice Amount '] = row[2][2:]
```

For no better reason than *it works best.*

With both of these solutions outlined, you can decide yourself which you prefer to use in your own situation. You might also want to consider some safety checks. For example, if you're omitting the first character in one field, you might want to store it in another so you can check that:

```
   record['firstcharacterCHECK'] = row[2][0]
```
Or indeed:
```
 record['firstcharacterCHECK'] = row[2].decode("latin-1")[0]
```
Make sure to do this *before* you strip out any characters with the line:
```
   record[' Invoice Amount '] = row[2][2:]
```

Removing the header row

There's one other thing: so far we've grabbed the whole dataset, including headers. And we've had to specify those headers ourselves.

If you want to avoid that work, you can use the `.DictReader` function instead of the `.reader` function, like so:
```
reader = csv.DictReader(data.splitlines())
```
This will create a dictionary so you can remove the line:
```
record = {}
```
...and replace `record` with `row` (the name of the new dictionary) in all subsequent lines, not forgetting the line that saves the data which should now read:
```
scraperwiki.sqlite.save(['Doc Number'], row)
```
In fact, you can remove most of those lines[11], because, unlike `record`, `row` does not start off empty and needing to be filled. The only lines you need are those which *change* the contents of `row`, which are now not done by index, but by *key* (the headers of each field, or column, in your data), i.e.:

---

[11]https://scraperwiki.com/scrapers/birminghamcouncilexpenditure_3rd/

```
   row[' Invoice Amount '] = row[' Invoice Amount
'][2:]
```

Note, by the way, that we must select the row by the exact name that is used in the dataset - including spaces. The best way is to copy it from a sample CSV, and paste directly into your code: in this case, the header for the column "Invoice Amount" actually has a space before and after - hence: `[' Invoice Amount ']`

Ready to scrape multiple sheets

With that solved and working on one sheet, we can now adapt our code to scrape multiple CSV files.

Unlike in previous scrapers, I'm not going to adapt previous code. Instead I'll write it from scratch, all in one bundle. Here's the code:

```
import scraperwiki
import csv
import lxml.html
#page with links:
url = 'http://www.birmingham.gov.uk/payment-data'
baseurl = 'http://www.birmingham.gov.uk'
def scrape_and_find_csv(url):

   html = scraperwiki.scrape(url)

   root = lxml.html.fromstring(html)

   #this selects all HTML containing link:
   <p class="fileicon"><a>
```

```
csvs = root.cssselect('p.fileicon a')

print csvs

for link in csvs:

     #this prints the result of
    adding the base URL to the
    relative link grabbed

    print baseurl+link.attrib.get('href')
```

...This `print` command is quite important: it will show us if the URLs we have grabbed are the right ones, and as it turns out they're not, quite.

The HTML ‹p class="fileicon"›‹a› contains not just our CSV file links, but also the PDF versions of the same data.

To make sure we only try to scrape the CSV files, we *could* use regex - but actually there's a simpler way, by using the same technique we employed to avoid grabbing the pound signs in the invoice amounts.

In this case you might note that the CSV files all end in .csv, while the PDF files end in .pdf.

We can grab those final three characters, then, and use an `if` test to *only* scrape the file if those final three characters are 'csv'.

To grab and print the final three characters we use the range `[-3:]` (from the third character from the end, until the end) by continuing our code like so:

```
print link.attrib.get('href')[-3:]
```

And to test if those three characters are 'csv', we use
the == operator in the next line like so:

```
if link.attrib.get('href')[-3:]
== "csv":

    data = scraperwiki.scrape(baseurl+link.attrib.get(

    reader = csv.DictReader(data.splitlines())

    for row in reader:

       row['missingletter']
       = row[' Invoice
       Amount '][0]

       row[' Invoice
       Amount_full'] =
       row[' Invoice Amount
       '].decode("latin-1")

       row['Invoice Ref']
       = row['Invoice
       Ref'].decode("latin-1")

       row[' Invoice
       Amount '] = row['
       Invoice Amount
       '][2:]
```

```
          row['URL'] =
     baseurl+link.attrib.get('href')

      print row

      scraperwiki.sqlite.save(['Doc
     Number', 'URL'],
     row)
```

```
#STARTS WORKING HERE!
scrape_and_find_csv(url)
```

And there our full code ends.

We've arrived here by running it, partially, in steps, and addressing the problems that come up at each point: the pound signs; the different file types.

The code above could then, of course, be simplified: you might not want to use all the trouble-shooting techniques we discussed earlier (using `.decode`; grabbing all characters after the pound sign; grabbing the first character just in case) but I've left them all in so you can see them in action.

There's also another line that needs explaining. If we know that pound signs trip up our scraper in *one* column, we might want to check if they occur in any *other* columns, by searching a sample CSV file to see if they turn up elsewhere.

And indeed, they do sometimes appear in the Invoice Ref column.

So one line applies that `.decode` method to that part of our data too. If you wanted to be sure, you might apply it to *all* fields.

Finally, if you run this, after some considerable time you might hit the following error: `AttributeError: 'NoneType' object has no attribute 'decode'`. This means that the scraper is trying to apply the `.decode` method where the data is 'None'.

In that case, you might want to add an `if` statement that only uses the `.decode` method if the data *is not None*, e.g.

```
 if row['Invoice
Ref']  is  not
None:

  row['Invoice
 Ref'] = row['Invoice
 Ref'].decode("latin-1")
```

This problem can also occur with other fields - for example, whatever you choose as your unique key *cannot* be 'None', so you might want to add another `if` statement to set that field as something else if *that* `is not None`, or only store it if that's the case (rather than it falling over).

Combining CSV files on your computer

A final note before we leave CSV files: if your intention is merely to combine CSV files then there is one particularly quick way to do so without doing any scraping. This involves downloading all the CSVs to a single folder on your

computer, and then using a special command to combine them.

On a Mac, this is done using the `cat` command in the Terminal like so:

1. Find and open up Terminal (the quickest way is by using the magnifying glass in the upper right corner to search for it).

2. On the window that appears, navigate to the directory containing your CSV files by using the `cd` command, e.g. type `cd desktop` followed by `cd csvs` (if your folder was called 'csvs' and was on your desktop).

3. Now you're in that directory, combine the files inside by typing this command: `cat *.csv > combinedsheets`

4. Open the new CSV file and check that headings are consistent. (You may have a few empty rows at the end of the first CSV data so don't assume it's not worked if you can't see the next set immediately.) If, for example, different sheets had headings in different columns you'll need to clean that up.

This thread[12] explains how to do the same thing using the command-line prompt on a Windows computer. And you can search for more details on the process online by looking for "using terminal combine csv files mac" or "using command line combine csv files"

---

[12]http://www.pcreview.co.uk/forums/do-you-combine-multiple-csv-files-into-one-file-t1739041.html

Of course you will have to weigh up the time that it will take to manually download each CSV, with the time it will take to write and run a scraper.

But that's it for this chapter. Now for the recap.

Recap

The `csv` module is quite straightforward in terms of using it to scrape CSV files. The files themselves are often, however, not. Here are some of the solutions to the problems we've encountered:

- If you have dozens of CSV files, you can combine them with a scraper.
- **Test your scraper on one sheet first** to iron out any problems before trying to scrape them all.
- If you get problems, **try omitting columns from your scraper** to identify which one is - or which *ones* are - responsible.
- If you *still* get problems, **try copying the data into a Google spreadsheet**, **removing elements you suspect**, **publishing as a CSV**, **and then scraping that**.
- **The `.DictReader` function can be used to create a dictionary of your CSV data**, with the column headings as keys. This also means you don't have to name each field yourself.
- If you're using that, **make sure you identify the column headings accurately by copying them directly from one** CSV and pasting into your code.

- Pound signs cause problems for the `csv` module, so **use `.decode("latin-1")` to convert your data into a format it's happy with**.
- **Use index ranges to grab particular parts of your data** - either to omit other parts (like a pound sign), or to test it (e.g. does the link end in .csv or .pdf?). An index range like [1:] will select from the second character to the end; a range like [-3:] will select from the third-to-last character to the end.
- **Record safety check data if you're changing what you're scraping**. For example, if you're omitting the first character in one field, you might want to store it in another so you can check that.

Tests

- Try formatting your data as you scrape it. For example, you might want to format numbers as integers or floats by using `int(row['key name here'])` or `float(row['key name here'])`
- Try setting up your scraper to only look for one particular type of spending. Cost codes are quite useful here. In this expenditure data there's a column for 'Cost Cente' (sic). I used a Freedom of Information request[13] to ask for a breakdown of these, which provided three more levels of detail on expenditure. The code RBL23, for example, refers to spending

---

[13]http://www.whatdotheyknow.com/request/data_dictionary_and_cost_centres#outgoing-243259

from the "Lord Mayor's Parlour". Try adapting the scraper to look for that code only, or a particular vendor.

- After the first two sheets you may hit an error where `.decode("latin-1")` fails because there is no data to decode. How can you add an `if` statement to only run that line if there *is* data to decode? What about the other lines too - including the unique key ('Doc Number')?
- What if CSV files created some time ago (and therefore linked at the bottom of the page) had different column headings (fields) or slightly different names for the same fields? How would you adapt your scraper to cope with that?

28 Scraper 27: Scraping Excel spreadsheets part 1

Having tackled CSV files in the previous chapter, now we need to look at their muscular cousins: Excel spreadsheets.

Whereas a CSV file is a single sheet of data, Excel spreadsheets typically contain multiple sheets within a single document. And they're used more often.

As well as tackling that problem in this chapter, we're going to tackle another: sometimes you might want to write a scraper for spreadsheets that are going to be published in

the future. In other words, we want to grab each new one automatically as it appears, and combine it with previous ones.

In this case we're going to do this with 'SITREPs' - situation reports generated during Winter by the health service in England (not, in this example, for other parts of the UK). They are published on the Department of Health's Transparency section[1], which describes them further:

> "Daily SITREPs are collected from acute trusts each weekday and indicate where there are any winter pressures on the service around the country such as A&E closures, cancelled operations, bed pressures, or ambulance delays. Daily Flu highlights the number of patients with confirmed or suspected influenza in critical care beds at 8am."

As you can see, this is lovely, timely data which could potentially have some newsworthy data in it. And we have time to prepare a scraper for it.

But to do that, we need a library of functions designed for the problems we're going to face.

---

[1]http://transparency.dh.gov.uk/2012/10/26/winter-pressures-daily-situation-reports-2012-13/

A library for scraping spreadsheets

It's been a while since we visited Scraperwiki's live tutorials page[2], but you will find there a tutorial for scraping Excel files - using, as it says in the description, the xlrd library.

If you click on that live tutorial to open it up, you'll find a link in the comments to the documentation for xlrd[3]. This will come in very handy later.

For now, here's the code of that tutorial scraper. I'll interrupt it when we get to something interesting:

```
# xlrd is a library for examining and extracting
data from Excel spreadsheets
# Documentation is available here:
 # http://www.lexicon.net/sjmachin/xlrd.html
# Perhaps the best way to learn it is to read
the source (Luke!) at:
# http://python-xlrd.sourcearchive.com/documentation/0.6.1/fi
import xlrd
import re
import string
from scraperwiki import scrape
# Nice example of a spreadsheet with useful data!
url = 'http://www.hmrc.gov.uk/stats/tax_structure/incometaxra
1974to1990.xls'
# This line will open the spreasheet from an url
book = xlrd.open_workbook(file_contents=scrape(url))
```

[2]https://scraperwiki.com/docs/python/tutorials/
[3]http://www.lexicon.net/sjmachin/xlrd.html

Here's our first interesting line. As you should be quite used to libraries by now, you should be able to work out that `.open_workbook` is a function from `xlrd` (given that it's attached to it with that period). If we wanted to know more, we could look for it in the documentation for `xlrd`[4].

Indeed, it is there, with the simple description "Open a spreadsheet file for data extraction." We can also see it has quite a few parameters. But as our tutorial code only uses one parameter, we can assume that most are optional.

Let's return to the code:

```
# We can find out information about the workbook
- number of sheets
print "Workbook has %s sheet(s)" % book.nsheets
```

Now, `.nsheets` looks interesting. This is being used on `book`, which we created a couple lines earlier with an `xlrd` function.

Because it was created with an `xlrd` function we can call it an `xlrd` *object*, which means we can use `xlrd` *methods* on it. How can we find out? The documentation.

And indeed, there it is[5]: `nsheets` tells us "The number of worksheets in the workbook".

We also know from previous scrapers that the `%` sign inserts a value (whatever comes after it) into a string (whatever comes before it), wherever `%s` or `%d` or `%f` appears.

Back to the code again:

```
# Loop over each sheet, print name, and number
of rows/colums
```

---

[4]http://www.lexicon.net/sjmachin/xlrd.html
[5]http://www.lexicon.net/sjmachin/xlrd.html#xlrd.Book.nsheets-attribute

```
for sheet in book.sheets():

   print "Sheet called %s has %s rows and %s
   columns" %(sheet.name,sheet.nrows,sheet.ncols)
```

Some more methods here: `.sheets` is used on `book`
too; and then on the resulting variable the methods `.name`,
`.nrows` and `.ncols` are all used.

These are all in the documentation too[6]:

- `sheets` "Returns: a list of all sheets in the book."
- `name` "Name of sheet."
- `nrows` "Number of rows in sheet. A row index is in
  `range(thesheet.nrows)`."
- `ncols` "Number of columns in sheet. A column index
  is in `range(thesheet.ncols)`."

Useful stuff. Back to the code:

```
# You can also access each worksheet by its index
firstSheet = book.sheet_by_index(0)
```

The comment here gives us all the explanation we need
for the `.sheet_by_index` method, but it's in the documen-
tation too[7].

The final lines of code contain some of those same
methods again, as well as some regex, and a function called
`unicode` which we haven't used before:

---

[6]http://www.lexicon.net/sjmachin/xlrd.html#xlrd.Sheet.name-attribute
[7]http://www.lexicon.net/sjmachin/xlrd.html#xlrd.Book.sheet_by_index-
method

```
    # You can loop over all the cells in the
worksheet
    for row in range(0,firstSheet.nrows): # for each
row

        for column in range(0,firstSheet.ncols):
      # for each column within the row

            cell = firstSheet.cell(row,column)

            # Each cell may contain unicode

            cellValue = unicode(cell.value)

             # Test each cell to see if
            it's a year range and print
            each value

            match = re.match("(\d{4}-\d{2,4})",cellValue)

             if match:

                print "Year: " + cellValue

                 # We could then extract
                the income tax rates
                by year.
```

To find out about the unicode function search for "unicode function python". You'll find some documentation

that explains *that*[8]. In a nutshell, `unicode` converts something into a unicode string - just as in previous chapters we used `str` to convert something to a string; and `int` to convert something to an integer.

What's unicode text? You can guess I'm going to recommend you do some searches to try to find out...

After you've read all of this code, then, you go ahead and click **RUN**.

And it's at that point you discover the tutorial scraper is broken.

What can you learn from a broken scraper?

According to the error messages in the Console underneath the scraper, the error is generated by line 16:

```
Line 16 - book = xlrd.open_workbook(file_-
contents=scrape(url))
```

Reading from right to left, this line takes the `url` variable, uses the `scrape` function to scrape it, and the results of that are assigned to the `file_contents=` parameter for the `open_workbook` function to use...

Still with me?

If we look at that line in the wider context of the scraper as a whole, we can see that it is - aside from importing libraries and initialising a variable - the first line to try to *do* something with those variables and functions.

---

[8]http://docs.python.org/2/library/functions.html#unicode

So our problem is probably quite fundamental. When an error is generated this early, it's often because we have forgotten to import the library containing the function we're using, or have forgotten to initialise a variable we're using.

We can rule both of these out.

But there's another simple test we can make: looking at the actual file this script is supposed to be scraping: `http://www.hmrc.gov.uk/stats/tax_structure/incometaxrates_-1974to1990.xls`[9]

If you stick that into a browser you'll be redirected to this page: http://www.hmrc.gov.uk/statistics/[10]

You can see that this is *not* the spreadsheet we were looking for: it's a webpage, with a different URL to boot. So our scraper is breaking because it's not actually scraping a spreadsheet.

The last part of the error message in the Scraperwiki console, in its own sweet way, is actually telling us this:

```
XLRDError: Unsupported format, or corrupt file:
Expected BOF record; found '<!DOCTYP'
```

XLRD is, of course, the name of our Excel scraping library. It's expecting one thing (a spreadsheet), and finding another (a HTML webpage).

To overcome this error, we need to give the scraper a *working* link to a spreadsheet file rather than this *broken* one which is redirected.

---

[9]http://www.hmrc.gov.uk/stats/tax_structure/incometaxrates_1974to1990.xls

[10]http://www.hmrc.gov.uk/statistics/

We may as well try one of the spreadsheets from our SITREPs.

To get the direct link we need to right-click on any of the links on the SITREPs webpage[11], and select 'Copy link address' (or something similar, it will vary depending on your browser).

Here's an example of one:

https://www.wp.dh.gov.uk/transparency/files/2012/10/DailySR-Web-file-WE-02-12-12.xls

With that copied, we just need to replace the broken link in the tutorial scraper with this working one:

```
url = 'https://www.wp.dh.gov.uk/transparency/files/2012/10/Da
```

And click **RUN**.

Now the scraper works - or at least doesn't generate any errors.

But what is the scraper doing?

Run through the messages in the Console to follow the results of the print commands in the scraper, as well as other information.

Firstly we have this:

```
NOTE *** Ignoring non-worksheet data named
u'Macro1' (type 0x01 = Macro sheet)
```

If you had already opened this spreadsheet manually, you'd have had a message in Excel asking if you wanted

---

to disable macros. This line tells you that the scraper - or rather the `xlrd` library - is ignoring that macro.

The next line is our first `print` command:

```
Workbook has 13 sheet(s)
```

This is the result of this line in our code:

```
print "Workbook has %s sheet(s)" % book.nsheets
```

So we can see that `.nsheets` generates a number for how many sheets the spreadsheet has.

The following lines are similar: they are the result of a `for` loop which grabs the name, rows and columns for each sheet, then inserts them into a string which it prints. Those strings look like this in the Console:

```
Sheet called A&E closures has 176 rows and 9 columns
```

After a dozen or so of those lines, the scraper finishes.

Looking back to our code, we can see a print command which didn't run:

```
print "Year: " + cellValue
```

This is because the preceding lines look for a regular expression in our spreadsheet, and only run the `print` command if there's a match.

Of course, that regular expression was written for a different spreadsheet, so it's no surprise that there should be no match.

We could adapt this for our spreadsheet. For example, if we wanted to grab all SHA (Strategic Health Authority) codes, we could adapt the line containing regex to this:

```
 match = re.match("([A-Z][0-9][0-9])",cellValue)
#SHA codes, e.g. Q30
```

You'll remember from Scraper #9, when we explored regex, that [A-Z] means 'any upper case character from A to Z' and [0-9] means 'any digit from 0 to 9', so the line above looks for any uppercase character followed by two digits.

You might also want to alter the string two lines down so that it reads:

```
print "SHA Code: " + cellValue
```

Otherwise you'll get a string of results saying "Year: Q30". But really, it's not essential - it's only a string.

Now the code is doing everything that it was designed to do:

1. Grab a spreadsheet from a URL, and open it
2. Count how many sheets it has
3. Tell us how many rows and columns each sheet has
4. Grab just the first sheet, and look for a regular expression in all the cells

But to save the contents of a spreadsheet, we're going to have to combine all of this with the knowledge we've already gained - and solve some new problems along the way.

Before we do that, let's recap what's been covered in this chapter.

Recap

- **The `xlrd` library can be used to scrape Excel files**.

- Useful functions and methods from the library include `.nsheets`, `.nrows` and `.ncols` which tell us, respectively, **how many sheets a spreadsheet has; how many rows a sheet has; and how many columns**.
- We can also use `.open_workbook` to open a spreadsheet; `.sheets` to do something with all sheets (for example loop through them), and `.sheet_by_index` to grab a sheet by its position (**index**) in the spreadsheet.
- We can use the `re` library to look for regular expressions in the cells.

Tests

- The documentation for `xlrd` mentions various *classes*. See if you can find out more about how classes work in Python - or in programming in general. Have a play with some tutorials to gain confidence in how they work. Might they be useful for journalism?
- Try altering the regex to look for another pattern in the data - for example a date.
- Try altering the code so that it grabs a different sheet.
- The sheets have the data column headings on a particular row. Can you add a line that prints the contents of that row?
- At the moment this scraper isn't saving. Can you borrow some code from previous scrapers to save some of the data?

29 Scraper 28: Scraping Excel spreadsheets part 2: scraping one sheet

Although we've fixed the broken tutorial scraper, we still need to work out how to use the lessons from that - and from the documentation - to scrape some spreadsheets.

Scraperwiki does have two other useful resources in addition to the live tutorial: a blog post about scraping Excel files[1], and, linked from that, an Excel reading guide[2].

You could also use advanced search techniques to look for scrapers that use the `xlrd` library by searching for "'import xlrd' site:scraperwiki.com"[3].

With that to hand, let's start mapping out what we need to do with our SITREP spreadsheets.

It's useful to be quite systematic in the construction of your scraper if it's trying to do more than one thing. In this case, we need to:

1. Grab *links* to spreadsheets from one webpage

---

[1]http://blog.scraperwiki.com/2011/09/14/scraping-guides-excel-spreadsheets/

[2]https://scraperwiki.com/docs/python/python_excel_guide/

[3]https://www.google.co.uk/search?q='import+xlrd'+site%3Ascraperwiki.com

2. From each spreadsheet, grab each *sheet*
3. From each sheet, grab all the *data*

The second two will be new challenges for us. In addition, we're going to face another complication: the data is going to vary from sheet to sheet.

But we're getting ahead of ourselves.

Rather than try to write a scraper that does all of this from the start, it's better to break it down into smaller parts, and build on each. That way, we can use trial and error to spot problems at each stage, rather than trying to solve multiple problems in one overwhelming scraper.

At the most basic level, then, we need to make sure that we can *scrape just one sheet on one spreadsheet*. Because once we've solved that problem, we can then try to do it for *all* sheets.

And once we've done *all* sheets in one spreadsheet, we can adapt our scraper further to scrape all sheets in *all* spreadsheets.

Testing on one sheet of a spreadsheet

Because we're going to scrape just one sheet on one spreadsheet, we don't yet need to write the code that finds all the links.

Instead we can go straight to the code that grabs the data from one of those spreadsheets. Here it is:

```
#set a variable for the spreadsheet location
```

```
    XLS = 'https://www.wp.dh.gov.uk/transparency/files/2012/10/Da
    #use the scrape function on that spreadsheet to
create a new variable
    xlbin = scraperwiki.scrape(XLS)
    #use the open_workbook function on that new
variable to create another
    book = xlrd.open_workbook(file_contents=xlbin)
    #use the sheet_by_index method to open the first
(0) sheet in variable 'book' - and put it into new
variable 'sheet'
    sheet = book.sheet_by_index(0)
    #use the row_values method and index (1) to grab
the second row of 'sheet', and put all cells into
the list variable 'title'
    title = sheet.row_values(1)
    #print the string "Title:", followed by the
third item (column) in the variable 'title'
    print "Title:", title[2]
    #put cells from the 15th row into 'keys'
variable
    keys = sheet.row_values(14)
    record = {}
    #loop through a range - from the 16th item (15)
to a number generated by using the .nrows method on
'sheet' (to find number of rows in that sheet)
    #put each row number in 'rownumber' as you loop
    for rownumber in range(15, sheet.nrows):

        print rownumber
```

```
record['SHA'] = sheet.row_values(rownumber)[1]

record['Code'] = sheet.row_values(rownumber)[2]

record['Name'] = sheet.row_values(rownumber)[3]

record['date1'] = sheet.row_values(rownumber)[4]

record['date2'] = sheet.row_values(rownumber)[5]

record['date3'] = sheet.row_values(rownumber)[6]

record['title'] = title[2]

print "---", record

scraperwiki.sqlite.save(["Name"], record)
```

Some things to explain: the title of the sheet is in the 3rd cell of the second row, and we grab it then print it in two steps with these lines - first all values in the second row, then the third item in that list of values (i.e. the third cell):

```
title = sheet.row_values(1)
print "Title:", title[2]
```

Secondly, like many government spreadsheets, the column headings are inconveniently placed a good 15 rows down. We grab those and put them in a list variable called keys with this:

```
keys = sheet.row_values(14)
```

(*That variable isn't used for now - but we'll be using it in a later chapter*). Now we grab the data itself by looping through the values in each row from row 16 onwards. That loop deserves breaking down:

```
for rownumber in range(15, sheet.nrows):
```

Read it from right to left: `sheet.nrows` is the number of rows in `sheet`. 15 is also a number. The `range` function creates a range of numbers you define with two parameters: the start and end of that range. In this case, the range is from 15 to the number of rows in this sheet.

Looping through the numbers in that range then, each is put in the variable `rownumber`: first 15, then 16, and so on until the number of rows in `sheet`.

That number is used in the lines that come within this loop, like this one:

```
record['SHA'] = sheet.row_values(rownumber)[1]
```

Here we're creating a field in the dictionary variable `record` called 'SHA', and pairing it with a value on the other side of the = sign.

The value is (take a deep breath): the result of using the `.row_values` method on `sheet` to grab the values for the row with the number of the variable `rownumber` (remember this is 15 the first time this loops, and 16 the second time and so on). That row has a number of values, so grab `[1]` - the second one.

So the first time the `for` loop runs, it will grab and store the second item in the 16th row. The second time it runs, it will grab and store the second item in the 17th row, and so

on until it reaches the end of the range specified in the `for` loop, and ends.

It does this for a number of different fields in `record`, storing the value of a different item from the row each time.

It also grabs that title cell we printed earlier, and then stores the lot in scraperwiki's datastore.

When you **RUN** this code it should scrape 161 rows of data: from row 16 to row 176. Great.

You might have noticed that there's an empty line of data in there (row 17). But it didn't cause us problems. If it had we could have always added an `if` test to assign it another value in those cases (perhaps the value of `rownumber`).

But now we've tackled that one sheet (you can find the same code here[4], by the way), it's time to broaden our horizons - which we'll do in the next chapter. First, a recap.

Recap

- With a spreadsheet scraper, **write it in stages**: first make sure you have solved any problems presented by scraping a typical sheet. Then adapt that to scrape multiple sheets (if you want to grab all sheets), and solve the problems presented by that challenge. And finally, adapt that so that you're scraping multiple spreadsheets - and any problems that that presents. Doing it systematically like this makes it easier to **identify problems at each stage**, rather than trying to find them in a whole mess of code.

---

[4]https://scraperwiki.com/scrapers/excelscraper_sitreports2/edit/

- The scraper works by **storing all values from each row in a list variable**. You then **store each item by accessing it with an index**, e.g. `sheet.row_values(0)[1]` grabs the second item in the list identified by `sheet.row_values(0)` - or the first row in `sheet`.
- You can **grab each row in turn by looping through a range of row numbers** and using each one in turn as the parameter for `.row_values()`
- To **identify the end of that range of row numbers** (i.e. how many rows there are), use `.nrows` on your sheet

Tests

- In our code we created a variable which we didn't use again: `keys = sheet.row_values(14)`. How can you adapt the code so that these keys are used to name each key (field) in the `record` variable? So, for instance the key `'SHA'` in the line `record['SHA'] = sheet.row_values(rownumber)[1]` is replaced with some code which calls the key for that column? We'll cover this in the next chapter, so don't worry if you can't.
- In the next chapter we'll adapt this scraper to loop through multiple sheets. But why wait until then? See if you can adapt it to do that. (*Tip: you'll need to know how many sheets there are, and use that to create another loop*).

- Perhaps you don't want all sheets but only one sheet, in a number of spreadsheets. Can you adapt the scraper so it loops through a series of links to spreadsheets (the page with those links is `http://transparency.dh.gov.uk/` and grabs a particular sheet from each?

30 Scraper 28 continued: Scraping Excel spreadsheets part 3: scraping multiple sheets

In the last chapter we wrote a scraper that tackled one sheet[1] of a spreadsheet. Having ironed out the bugs from that, we're now ready to adapt that so that we're repeating the process across multiple sheets in a single spreadsheet.

A good way to approach this problem as a programmer is to think **what fixed elements can be replaced with variables**. Here are some examples you've already encountered:

- The URLs being scraped - at first we scraped a single fixed URL, but later we *stored that URL in a variable* which we could run through a scraper. This meant that we could change the URL being stored in the variable, and *repeat the same code.*
- A part of a URL: in a previous scraper the URLs ended with ID numbers. To scrape them all we *put them in a list and then looped through the list,*

---

[1]https://scraperwiki.com/scrapers/excelscraper_sitreports2/edit/

> putting one ID number into a variable so we could add that to the variable being scraped, and *repeat*.
> - A range of numbers: earlier we used the number of rows as the end of a range, which we looped through - again the variable kept changing as we looped.

In this case we need to *repeat* the scraping process performed on one sheet, so that it is done on *all sheets*.

The solution can be adapted from the techniques listed above: we can use a `for` loop to repeat the process on every sheet within a particular range (and we know a **method** we can use to count how many sheets there are in total). And we can replace the index of the sheet to be scraped with a variable representing that value in each case. Here's how.

The start of our code remains the same:

```
import scraperwiki
import xlrd
URL = 'http://transparency.dh.gov.uk/2012/10/26/winter-pressu
#set a variable for the spreadsheet location
XLS = 'https://www.wp.dh.gov.uk/transparency/files/2012/10/Da
#use the scrape function on that spreadsheet to
create a new variable
xlbin = scraperwiki.scrape(XLS)
#use the open_workbook function on that new
variable to create another
book = xlrd.open_workbook(file_contents=xlbin)
```

But where we need to change the code is on the next line:

```
sheet = book.sheet_by_index(0)
```

Instead of scraping one sheet, we need to add a for loop so it scrapes one, and then the next, and so on until the last. To do that we need to know how many sheets there are. We'll store that number in a new variable called `sheetstotal`:

```
sheetstotal = book.nsheets
```

Then we can create a range beginning with '0' and with that variable's value as the end of the range:

```
sheetsrange = range(0,sheetstotal)
```

And with those two ingredients, we can start a for loop that scrapes each sheet:

```
for sheetnum in sheetsrange:
```

The first line of that loop is very familiar:

```
    sheet = book.sheet_by_index(sheetnum)
```

It's the same as the one we removed, but with (`0`) replaced by (`sheetnum`) so it will change each time.

The rest of the code is identical, save for the fact that lines have now been indented so that they run within the for loop:

```
    title = sheet.row_values(1)

    #print the string "Title:", followed by
    the third item (column) in the variable
    'title'

    print "Title:", title[2]

    #put cells from the 15th row into 'keys'
    variable
```

```
keys = sheet.row_values(14)

record = {}

#loop through a range - from the 16th
item (15) to a number generated by using
the .nrows method on 'sheet' (to find
number of rows in that sheet)

#put each row number in 'rownumber' as
you loop

for rownumber in range(15, sheet.nrows):

    print rownumber

    Name = "no entry"

    record['SHA'] = sheet.row_-
    values(rownumber)[1]

    record['Code'] = sheet.row_-
    values(rownumber)[2]

    record['Name'] = sheet.row_-
    values(rownumber)[3]

    record['date1'] = sheet.row_-
    values(rownumber)[4]

    record['date2'] = sheet.row_-
    values(rownumber)[5]
```

```
 record['date3'] = sheet.row_-
values(rownumber)[6]

 record['title'] = title[2]

 print "---", record

 scraperwiki.sqlite.save(["Name"],
record)
```

However, there is a problem: what was unique in a single sheet - the name of the hospital trust - is not unique across multiple sheets. The trust appears once on each sheet. Our previous unique key, "Name", will not do.

Now you can have multiple keys like so:

```
 scraperwiki.sqlite.save(["Name",
"date1"], record)
```

But in this case that will still cause problems - largely because the other columns are no more unique, even in combination.

So we'll once again need to create an arbitrary 'counter' to create a unique id for each row. In this case we need to set it at 0 before the first for loop, like so:

```
id = 0
for sheetnum in sheetsrange:
```

And then within the code add one each time it loops just before the data is stored:

```
id = id+1
```

```
record['id'] = id

print "---", record

scraperwiki.sqlite.save(["id"],
record)
```

So, that's that sorted. But what other problems might
we have when moving from a single sheet to multiple ones?
Well, looking at what data we're storing, it seems we've
made two big assumptions: that the data is in the *same
place*, and that it has the *same fields*.

One of those assumptions is wrong: although columns
2, 3 and 4 are consistently the SHA, code, and name of the
trust, the columns after those represent varying fields such
as different dates, and in many spreadsheets there are more
columns than we're grabbing.

One dataset, or multiple ones

At this point, as with many problems, we face a fork in
the road. We can solve the problem in two ways: either by
storing more and more fields to accommodate the variety
within the sheets so, for example, we would have multiple
rows for one hospital trust, all of which had empty columns
where they refer to a different sheet (and so a different
row).

Or we could store more datasets. Each of these would
have only one row per hospital. It would be effectively a
mirror of the spreadsheet - but with the key difference that

running it across multiple spreadsheets would combine
them - i.e. the 'Cancelled operations' sheets from a number
of spreadsheets would be combined into one dataset, while
the 'A&E Closures' sheets would be combined into another,
separate one.

There is no 'right' answer, and both routes will have
their merits in different situations.

It's particularly worth thinking here about post-scraping
cleanup: it might be easier to scrape data in an ugly way (for
example by combining sheet titles with column headings)
which can be easily cleaned up in Excel or Google Refine
afterwards (by adding in a special character during scrap-
ing that you can split it on), if that's going to be faster than
working out a way to do it cleanly in Scraperwiki in the
first place.

In this case - as in most cases - the key question is:
"What do you want the data for?" Or, what is the question
you want to ask it? In our case it's unlikely that we're going
to want all the sheets to be combined from the start - it's
more likely that we're going to ask a question specifically
about cancelled operations (sheet 4). And if we wanted
to ask a question of two datasets - say, if there was any
relationship between cancelled operations and norovirus -
we can do that more easily in Excel or Google Docs, or
even by using SQL queries across multiple datasets using
the Scraperwiki API (see exercise 2 in this post[2] for more
on using the API and SQL queries).

---

[2]http://datamineruk.wordpress.com/2011/07/21/getting-to-grips-with-
scraperwiki-for-those-who-dont-code/

For our purposes, then, we're going to store different sheets as different datasets.

The key line here is:

```
scraperwiki.sqlite.save(["id"], record)
```

This is the line that stores the data, and names the dataset. Only, we haven't been naming the dataset, because that's optional (you can find the documentation on the Scraperwiki datastore here[3].

The name of the dataset can be added as a third parameter in the line as follows:

```
scraperwiki.sqlite.save(["id"], record, table_-
name="nameofdataset")
```

Of course the string "nameofdataset" can be replaced with a variable like so:

```
scraperwiki.sqlite.save(["id"], record, table_-
name=sheetnum)
```

Changing the code in this way and clicking **RUN** will run the scraper in the same way as before (you can see an example of that code here[4] by the way), but it will name the datasets based on the sheet number, and each sheet number will be a separate dataset.

At this stage that's no better than simply downloading the spreadsheet - but we have bigger plans: we want to combine a whole bunch.

Before we can do that, though, we still need to solve the problem of varying columns: as it stands, our scraper grabs a *fixed* number of cells from each row (positions 1

---

[3]https://scraperwiki.com/docs/python/python_datastore_guide/

[4]https://scraperwiki.com/scrapers/excelscraper_sitreps_multiplesheets_2/

through 6 in `sheet.row_values`) and the headers are fixed too, including the vague titles 'date1'. 'date2' and 'date3'.

We really want to grab the dates themselves, and all columns.

Using header row values as keys

Now, once again, we need to move from fixed references to variables. Instead of listing and limiting the fields we want to capture (for example, columns 2 to 7), we need to make our code more flexible (for example, from the second column to the last one containing data).

Along the way, we're going to do the same for the keys themselves: instead of 'SHA', 'code' and so on, we'll use 'The heading of this column', 'the heading of that column'.

As before, we solve this through **variables** and **loops**. And yes, we're going to create another **range** to loop *through*.

Here's a line of code to illustrate the problem:

```
record['date3'] = sheet.row_values(rownumber)[6]
```

In this line the name of the key - `'date3'` - and the index of the column - `[6]` are both fixed.

Instead of fixing the index of the column we could loop through them with a couple of lines like this:

```
for column in range(1,sheet.ncols):

    record['date3'] = sheet.row_values(rownumber)[column]
```

Note that the `[6]` in the original line is now replaced by a variable - `column`, which is `1` the first time it runs (because

our range starts at 1: the second column; the first column is empty so we're not including it), and might end at 6 on one sheet, or 9 on another, depending how many columns (`.ncols`) it has.

This is a much shorter piece of code than the previous six lines which individually stored each item. But it has a problem: the key for *all* six items as they are stored is now 'date3', where we really need separate keys for each.

We solve that by replacing *that* fixed part of code. How? By fetching the right *key* for the right *column.*

Thankfully we can identify that in exactly the same way that we identify each item in the row: with its index.

Remember in the last chapter we created a variable that we didn't use? It was this line:

```
keys = sheet.row_values(14)
```

This grabbed the contents of the 15th row in `sheet` and put it in the variable `keys` (*the name was a rather obvious hint*). Because there are a number of values, that is a list variable, and we can access the first, second or third item (etc.) in that list with an index like so: `keys[1]`

You could find out what each key was by using the `print` command like so:

```
print keys[1]
```

Which would probably give you "SHA".

Now, before, we manually entered the string 'SHA' as the first key for our variable `record` (a dictionary). Now we've explained about this `keys` variable, we can use that to create the keys in our loop by adapting it to look like this:

```
for column in range(1,sheet.ncols):
```

```
     record[keys[column]] = sheet.row_values(rownumber)[column]
```

This time we've replaced `record["date3"]` with `record[keys[column]]`.
Whereas `"date3"` is a fixed string, `keys[column]` is (note
the absence of quotation marks) a *dynamic* reference to an
item in a list.

To illustrate - and this is a useful exercise with loops -
try writing down what happens as the loop operates:

The first time `for column in range(1,sheet.ncols):`
runs, the variable `column` is the first number in the range:
1.

That means that every mention of `column` in the next
line should be read as 1. So, we grab `keys[1]` as the
name of our key in `record`, and we grab `sheet.row_values(rownumber)[1]` to put into it.

To complicate matters, `rownumber` is also a variable
being generated by a loop containing *this* loop:

```
for rownumber in range(15, sheet.nrows):
```

So, the first time this loop runs within *that* loop,
`rownumber` is 15, meaning we can further decode our code
like so:

```
record[keys[1]] = sheet.row_values(15)[1]
```

Until now we've been focused on the most recent `for`
loop, so at this point it's worth taking a step back to gain
some perspective:

- Our first loop goes through each rows within a
  particular range...
- Within that, **for** each row, a second loop goes through
  all the *columns*, and records each one.

So, in the first run of the first loop, `rownumber` is 15, and within that it loops through each `column`. This means it runs like this:

- `rownumber` 15, `column` 1, 2, 3, 4, 5, 6
- `rownumber` 16, `column` 1, 2, 3, 4, 5, 6
- ...and so on.

Because we grabbed the `keys` list earlier as well it adds those as keys and stores them as it loops through each value.

Finally, we need to save the whole `record`, and to be safe we need to use the same convention in identifying our unique key:

```
scraperwiki.sqlite.save([keys[2]], record)
```

But if you indent that line with the last `for` loop along with the line ' record[keys[column]] = sheet.row_-values(rownumber)[column]' then you will generate this error:

```
SqliteError: unique_keys must be a subset of
data; bad_key:Code
```

That is because the first time that the `for` loop runs, it is only saving the value for *one* key. At that point `keys[2]` hasn't been used at all.

To solve that problem make sure that the line saving `record` to the datastore comes *after* that final `for` loop but *within* the one before it, like so:

```
for rownumber in range(20, sheet.nrows):
```

```
for column in range(1,sheet.ncols):

    record[keys[column]]
  = sheet.row_values(rownumber)[column]

scraperwiki.sqlite.save([keys[2]], record)
```

You can see this code at this example scraper[5] (*look towards the end for the relevant section of code*).

Unfortunately, this scraper generates a new error.

I say unfortunately. Actually, it just gives me an excuse to explain another problem-solving technique...

But that technique opens a whole Pandora's Box of new problems and solutions - so I'll save that for the next chapter.

For now, we desperately need a recap.

Recap

- **Think what fixed elements can be replaced with variables** in your scraper: that might be a URL (if you want to do the same thing to more than one URL, such as scrape it); the position of a header (if you want to grab each one), or the end of a range (if it will change for each sheet).
- Use methods like .nsheets, .ncols and .nrows to **identify how many sheets, columns and rows**

---

[5]https://scraperwiki.com/scrapers/excel_sitreps_multiplesheets_varyingcols/

there are in a spreadsheet or sheet, and replace previously fixed elements.

- **Store a list of the headings in the header row** and access each one to create keys for the dictionary variable containing your data.
- **Loop through a range** of sheets, rows or columns with a `for` loop to grab or scrape each one.
- **If your key is no longer unique**, **try using multiple unique keys**
- **Ask yourself if your questions are likely to only need multiple datasets which you can question separately**, or if you really need one whopping single dataset at the end of this.
- **Use the `table_name=` parameter and a variable which changes from sheet to sheet** (such as `sheetnum`) to name and save more than one dataset separately.
- Make sure you **save your data during the loop for each row** - *not* the row for each column.

Tests

- The keys cause a problem that we solve in the next chapter. But why wait until then? See if you can identify the cause, search for possible solutions, and try them out.
- Write down what happens to the spreadsheet as it passes through the scraper. See if you can trace what happens within the `for` loops.

- Use the skills developed so far to scrape spreadsheets from the UK's Higher Education Statistics Agency (HESA)[6]. Each year has its own spreadsheet - a scraper would save you having to combine them manually.

---

[6]http://www.hesa.ac.uk/index.php/content/view/1973/239/

31 Scraper 28 continued: Scraping Excel spreadsheets part 4: Dealing with dates in spreadsheets

At the end of the last chapter I left you on a cliffhanger. I know: you were desperate to know what would happen next.

The cliffhanger was this error:

```
SqliteError: key must be string type; bad_-
key:41219.0
```

That's actually quite easy to understand. Keys must be strings of characters. One of our keys is not a string. It is a *bad* key. And its value is 41219.0.

Now that's odd. When you look at the column headings (our keys) in the spreadsheet, there *is no number 41219.0.*

Well, actually, there is. It's just that Excel didn't show it to us that way.

41219.0 is actually a date: the 6th November 2012. Excel *stores* this date as a number[1], normally representing the number of days since the start of the year 1900 (although some count from 1904. Don't ask.)

---

[1]http://www.cpearson.com/excel/datetime.htm

This means it can calculate the differences between dates, and present dates in different ways (US style, UK style, as a combination of words and digits, and so on).

Because Excel normally does all this work for us, we don't normally need to know how to convert 41219.0 into a date. But now we do. And we need to make it a string too.

As you'd expect, you're not going to be the first or the thousandth person to have encountered this problem, and it has been solved many times by many of those people already. And as we've learned, programming is all about re-using code, including other people's.

So to find their solutions, we just need to search for "converting excel date number to date python" or something similar. The results[2] should contain some useful links, including, as it happens, the Scraperwiki documentation on Excel and xlrd[3], which says:

> "Dates will come out as floating point numbers by default (e.g. 36649.0), unless you specially convert them. This cellval function does the hard work for you."

Oh, if only we'd been paying attention.
Here's the function below that:

```
import datetime
def cellval(cell, datemode):

    if cell.ctype == xlrd.XL_CELL_DATE:
```

---

[2]https://www.google.co.uk/search?q=converting+excel+date+number+to+date+python

[3]https://scraperwiki.com/docs/python/python_excel_guide/

```
    datetuple = xlrd.xldate_as_-
tuple(cell.value, datemode)

 if datetuple[3:] == (0, 0, 0):

     return datetime.date(datetuple[0],
     datetuple[1], datetuple[2])

 return datetime.date(datetuple[0],
datetuple[1], datetuple[2], datetuple[3],
datetuple[4], datetuple[5])

 if cell.ctype == xlrd.XL_CELL_EMPTY:
return None

 if cell.ctype == xlrd.XL_CELL_BOOLEAN:
return cell.value == 1

 return cell.value
```

And then another two lines of code which will come in useful:

> "*Read a whole row, with dates properly converted.*"

```
print [ cellval(c, book.datemode) for c in
sheet.row(4) ]
```

> "*Read individual cells like this.*"

```
print cellval(sheet.cell(0,0))
```

We'll come back to those in a minute. First, some things to explain about the function `cellval`:

- `datetime` is a Python library for interpreting dates. (*We need to* `import` *it because this new function uses a function from that library:* `datetime.date`)

- The new function `cellval` takes two parameters: `cell`, and `datemode`. These could be given any names, but they're pretty literal, and give us a clue as to what they should be. For example, we can guess that the `cell` parameter should be the spreadsheet cell we want to convert. But `datemode` is more cryptic.

- A little searching[4] will tell you that `datemode` refers to whether the date number is calculated from a base of 1900 or 1904. If it's 1900 then datemode is 0, and for dates calculated from 1904 the datemode is 1. This is also explained in the xlrd documentation[5], which also mentions (blink and you'll miss it) that you can use `book.datemode` to find out what the datemode *is* of the object `book`. (Here's an example of someone using that code[6]).

The rest of the function largely consists of a number of `if` tests conducted on the cell it's been given, the results

[4] http://www.onnraves.com/2008/02/23/datemode-meaning-in-xldate_as_tuplexldate-datemode/

[5] http://www.lexicon.net/sjmachin/xlrd.html

[6] http://thoughtsbyclayg.blogspot.co.uk/2008/12/get-dates-from-excel-with-python-xlrd.html

of which determine what date is returned at the end.
Most of those tests involve more functions from the xlrd
library (you can tell because they start with xlrd., such
as xlrd.XL_CELL_DATE and xlrd.xldate_as_tuple). If you
want to know more about those you can look in the xlrd
documentation[7] - again.

The key thing, however, is to test if this code works. To
do this, we're going to have to define our function quite
early on in the scraper - straight after you have imported
your libraries, for example.

Copy the code from the Scraperwiki documentation on
Excel and xlrd[8] into that part.

Now you just need to add a line which *runs* that
function, using the cell and the datemode as parameters.
This line is best added *before* the for loop begins: for
rownumber in range(15, sheet.nrows): (*in fact, it may
be worth commenting out that entire section, which you can
do by adding three quotation marks before it and after it,
like so: """*).

At this point it's worth playing with the two lines of
code which the Scraperwiki documentation suggested for
using the function. The first…

```
print [ cellval(c, book.datemode) for c in
sheet.row(4) ]
```

…we can adapt like so:

```
print [ cellval(c, book.datemode) for c in
sheet.row(14) ]
```

---

[7]http://www.lexicon.net/sjmachin/xlrd.html
[8]https://scraperwiki.com/docs/python/python_excel_guide/

All we have changed is the number of the row we're looping through.

That line of code could also be rewritten like this:

```
for c in sheet.row(14):

    print cellval(c, book.datemode)
```

The only difference is that because the original line is contained within square brackets, the results are put into a list:

```
[None, u'SHA', u'Code', u'Name', datetime.date(2012,
11, 6), datetime.date(2012, 11, 7), datetime.date(2012,
11, 8), u'9 Nov 12 to 11 Nov 12']
```

In our rewrite each result is printed one by one, with dates looking like this:

```
2012-11-06
```

The second sample code which the Scraperwiki documentation suggests is this:

```
print cellval(sheet.cell(0,0))
```

Here we are using the function on just one cell, rather than a whole row. The cell is identified by its coordinates (row and column, or: `0,0`) but Scraperwiki doesn't tell us which is which. No matter, we can work that out quickly through a test:

```
print cellval(sheet.cell(14,4))
```

Actually this generates an error: `TypeError: cellval()` `takes exactly 2 arguments (1 given)`

What this is telling us is that we've only given `cellval` one argument (parameter). That's right: we only gave it

the location of the cell, not the datemode. It seems the documentation was a little bit wrong.

No worries, we can easily amend it like so:

```
print cellval(sheet.cell(14,4), book.datemode)
```

And now it should work. It's grabbed the date, which we know is 15 rows down and 5 columns across, so that's solved that.

One final mistake to highlight: an easy mistake here is to use the *value* of your cell rather than the cell *location*. Here's an example:

```
 date = cellval(keys[4], book.datemode)
```

Or:

```
 date = cellval(41219.0, book.datemode)
```

(*As the 5th item in the list* keys *is 41219.0, these two lines of code mean the same thing*)

This will generate the error `AttributeError: 'float' object has no attribute 'ctype'`.

If you get this error, it's because the code `'ctype'` is an attribute of a *cell*, not a *value in a cell*. It's a subtle distinction, but an important one. Likewise, if you try this code:

```
print [ cellval(c, book.datemode) for c in
sheet.row_values(14) ]
```

Note that you are using `.row_values` and *not* `.row` as we did earlier, which will result in the same problem. If you want to read more about 'ctype' and the cell class, look in this part of the xlrd documentation[9].

---

[9]http://www.lexicon.net/sjmachin/xlrd.html#xlrd.Cell-class

Now that we know all this we can adapt the code creating our keys list like so:

```
keys = []
for cell in sheet.row(14):

  keys.append(cellval(cell,book.datemode))
```

This will work in solving the first problem: converting a number into a date, but it is still not a string - so you'll get this error:

```
SqliteError: key must be string type; bad_-
key:2012-11-08
```

We just need a function that converts dates to strings, and the simplest one is this: `str()`

The `str()` function turns any parameter into a string, just as `int()` and `float()` convert strings into integers and floats.

We can add this to our code like so:

```
  keys.append(str(cellval(cell,book.datemode)))
```

Note that `str()` is applied to the result of the `cellval` function before that is appended to the `keys` list (once again, the code works from right to left, and code in parentheses runs first).

Now, when we run it, our records will look like this:

```
{'2012-11-07':  0.0,  '2012-11-06':  0.0,  'Name':
u'BRADFORD TEACHING HOSPITALS NHS FOUNDATION TRUST',
'title': u'Daily Hospital Situation Report - A&E
Closures', '2012-11-08': 0.0, 'SHA': u'Q32', 'Code':
u'RAE', '9 Nov 12 to 11 Nov 12': 0.0}
```

Note that the dates are now in inverted commas: they are strings.

But after a few hundred records we'll hit another problem:

```
SqliteError: key must be simple text; bad_-
key:Urg ops canc'd in last 24hrs
```

More string formatting: replacing bad characters

This time it's an awkward heading that's causing us problems. The heading comes on sheet 4 - *Cancelled operations*. We can probably guess what the problem is with "Urg ops canc'd in last 24hrs" - it is abbreviated, which means it uses an inverted comma. We want to avoid characters like that (which might be interpreted as marking the end of a string).

Now, bearing in mind that we'll need to run the scraper again, we might want to avoid it scraping hundreds of rows on the first three sheets before we can find out if our next tweak has worked or not.

To do that - and save time testing - you could change the line:

```
sheetsrange = range(0,sheetstotal)
```

to:

```
sheetsrange = range(3,sheetstotal)
```

This means it will *start* on the fourth sheet (index 3) - the one with our problematic headers.

Now for our problematic headings. To remove the offending apostrophe, we can use one of a number of **string**

formatting methods. There are lots of these, which you can find listed in the documentation (or by searching for 'string methods python'), but here are just a few. In all cases the string or the variable name for the string comes before the period:

- `.upper()` converts a string to uppercase
- `.lower()` converts a string to lowercase
- `.split()` splits a string into more than one item, based on a specified **delimiter** (if the delimiter is a space, then this method will be saying "*split this string every time there is a space*"). Note that the result (if the delimiter is there) is a list, not a string.
- `.islower()` tests if a string is all lower case. This will return `True` if it is, and `False` if not - useful for `if` statements.
- `.replace()` will replace one character or series of characters with another - if it occurs in the string. You can also specify how many times it does this before it stops (e.g. replace the first two occurrences).

There are plenty of others. But it is this last method that we will use - to replace the apostrophe with, well, nothing.

To try this out in a simple way you can type the following code:

```
"Urg ops canc'd in last 24hrs".replace("'","")
```

The `.replace` method takes two arguments (parameters): first, what we want to replace (in quotation marks or inverted commas), then after a comma: what we want to replace that with. In the code above we want to replace:

```
"'"
```

(an inverted comma) with:

```
""
```

(nothing).

Note that we used double quotes to indicate the string. If it was a quotation mark we wanted to replace, we'd use single quotes like so:

```
'"'
```

It's a flexible feature of strings that's worth remembering: your string of characters can use either single or double quotation marks - whichever it is, use the alternative character to mark the start and end of the string itself.

To use this method in our original code means making an already complex line even more complex. Here it is in the second line of code below:

```
for cell in sheet.row(14):

    keys.append(str(cellval(cell,book.datemode)).replace("'","
```

Wow. This is one of those lines which looks maddeningly obscure if you haven't written it, but is really quite impressive if you have.

Thankfully, you have been writing along with this code as it's mutated from its original form. But let's reverse-engineer it anyway.

The core of this line is `cellval(cell,book.datemode)`. That runs the `cellval` function defined earlier in the code on `cell` (one of the cells from row 15, created in the `for` loop above) based on the `datemode` of the spreadsheet.

If `cell` was a date it returns a formatted version of that. Otherwise it returns the original contents of the cell. So let's say what we have, then, is `"Urg ops canc'd in last 24hrs"`

Now then, if you can imagine that phrase `"Urg ops canc'd in last 24hrs"`, wrapped around it is the function `str()` which converts it to a string. Well, it's already a string, so we still have `"Urg ops canc'd in last 24hrs"` after that function has done its work.

Next, attached to that result, is `.replace("'","")` which replaces any apostrophes in the string with nothing. So now we have `"Urg ops cancd in last 24hrs"`.

All of that is contained within a parenthesis for `keys.append()`, which appends it (`"Urg ops cancd in last 24hrs"`) to the list variable `keys`.

This repeats for each cell in row 15, populating a list of headings, each of which has been converted from a date (if it was one), turned into a string (if it wasn't one), and had any apostrophes removed (if they had one).

Now the scraper works. You can see the full code here[10].

Oh, don't forget to change the `sheetsrange` variable so it starts again at 0, and not 3:

```
sheetsrange = range(0,sheetstotal)
```

There is one remaining problem with this particular sheet - and indeed some others: the headings are actually spread over two rows: in row 15 are the SHA, Code, and text headings (e.g. "Urg ops cancd in last 24hrs"). But in

---

[10]https://scraperwiki.com/scrapers/excel_sitreps_multiplesheets_varyingcolsdates/

the cells above those text headings are the dates.

To further complicate things, some headings are in
merged cells (there is a cell E14 but no F14 or G14) and
others are not (H14 contains the next date heading but I14
and J14 are empty).

The solution to this problem is convoluted, dull, and
only goes over ground we've already covered in the most
mind-numbing way possible. But some key points if you
want to explore this:

- If this was the sheet you were interested in, then I
  would scrape it separately, allowing you to tackle
  its idiosyncrasies directly. If it is not, then don't get
  pulled into hours trying to solve a problem which
  doesn't matter that much.
- Identify the pattern in your problem. For example:
  the first four headings are only in one row. In all
  columns from 5 onwards you're looking for a cell
  containing text above - or, you're looking for the first
  cell containing text within a three-cell range. You can
  get more specific, but you get the picture.
- This pattern can be identified with `if` statements and
  `for` loops.
- Alternatively, you might look to see if that data is
  elsewhere: for example, each sheet has a 'Period'
  heading, giving the first and last dates covered. The
  title of the spreadsheet might do too. Any of these
  clues could be used to populate the columns 5 on-
  wards, if you know it will be divisible by those dates

(e.g. 5 dates and 10 columns means the first two columns belong to the first date and so on).

- Whatever you do, always check the specific part of the raw data before publishing anything based on a particular part of your scraped data.

Note: Tony Hirst wrote about this and other problems highlighted by this dataset and scraper in his post 'When Machine Readable Data Still Causes "Issues" - Wrangling Dates[11]*'. It's well worth a read if you're interested in the problems thrown up, as is his* scraper to solve those problems[12].

Scraping multiple spreadsheets

You'll be relieved to know, after all that, that the final part of the scraper is comparatively problem-free. This is where we:

- Re-use a piece of code which simply grabs a series of URLs from a list of links on one page, and scrapes each one.
- Convert most of our code so far into a *function* so that it can be used on more than one spreadsheet.
- Change our *unique keys* because they won't be unique in the context of more than one spreadsheet.

---

[11]http://blog.ouseful.info/2012/11/27/when-machine-readable-data-still-causes-issues-wrangling-dates/

[12]https://scraperwiki.com/scrapers/nhs_sit_reps/

Let's start with the grabbing-URLs code. Towards the
end of the scraper we need to add this code:

```
import lxml.html
URL = 'http://transparency.dh.gov.uk/2012/10/26/winter-pressu
def grabexcellinks(URL):

    html = scraperwiki.scrape(URL)

    print html

    root = lxml.html.fromstring(html)

    links = root.cssselect('p a')

    for link in links[1:-1]:

         #print the text_content of
        that (after the string "link
        text:")

         print "link text:", link.text_-
        content()

         #use the attrib.get method
        on 'link' to grab the href=
        attribute of the HTML, and put
        in new 'linkurl' variable

         linkurl = link.attrib.get('href')

         print linkurl
```

```
#run the function scrapesheets,
using that variable as the
parameter

scrapesheets(linkurl)
```

```
grabexcellinks(URL)
```

The first line of code above imports a library we now need to find those links - `lxml.html`. You should of course put this right at the top of the scraper where the other `import` lines are, but it will still work here.

The second line defines our URL, and the last line uses it to run a function on it. Everything in between *is* that function, which step by step scrapes it, converts it to an lxml object, identifies text in particular HTML tags, grabs links, and then runs the scraper function we wrote *oh so long ago* on those linked spreadsheets.

The only line to particularly note is the one that reads ' for link in links[1:-1]:' - this specifies a range of links to loop through, rather than all of them. It excludes the last one because that one we *don't* want to scrape, and likewise the first link is excluded from the range because this is aggregate figures.

🔑 Tip: limit ranges to test quickly

When you're first testing this scraper, you might want to limit the range of links further to check it all works properly. Alongside that, you might also want to limit the ranges that specify which sheets to scrape, and which rows. For example, you might tell it to scrape 10 rows, on 4 sheets, on 4 links first, just to check the keys and other elements work properly

There's one other, very important, thing to note: almost everything we've written before now needs to be contained within this new `scrapesheets` function. So, before the line ' xlbin = scraperwiki.scrape(XLS)' you'll now need to add this one:

```
def scrapesheets(XLS):
```

...and you'll have to **add an extra indent to every single line you want within that function** so that they are included - right up until the line that saves it to the scraperwiki datastore.

Also, because we're saving more than one spreadsheet, our previously 'unique' key - the SHA code on each sheet - is no longer unique. So we'll need to define more than one unique key by firstly adding a new key and entry for the URL within the `for rownumber` loop like so:

```
 record['URL'] = str(XLS)
```

(*Note that we've formatted the URL variable as a string with `str` too - try it without and see what happens*)

...and then we use that as one of our unique keys like so:

```
 scraperwiki.sqlite.save([keys[2],'URL'], record,
table_name=sheetnum)
```

Alternatively you could create that arbitrary `id` variable as we've done before, and add 1 to it for each row, using that as a unique key.

At the end of it all[13] you'll have a dataset where there is a different row for every trust in every spreadsheet. This will take a little cleaning up in Excel or Google Docs (the easiest thing to do is create a pivot table with trusts as your rows and all the dates in the *values* area).

The only thing left to remark is how many loops are embedded in this scraper. Let's take stock.

Loops within loops

The scraper we've just written[14] is a particularly mind-bending example of using loops within other loops, or creating a list, and for each item in that list grabbing another list, and so on. Here's what it does on that front:

- List 1: start on a webpage and make a list of all the links to spreadsheets...
- List 2: **for** each item in that list (each spreadsheet), make a list of all the sheets that one spreadsheet has...

---

[13]https://scraperwiki.com/scrapers/sitreps_linkscraper_full/
[14]https://scraperwiki.com/scrapers/sitreps_linkscraper/

- List 3: **for** each cell in the header row, convert values to dates, strings, and replace apostrophes, then use them as a list of keys to be accessed later.
- List 4: **for** each sheet in that spreadsheet, go through all the rows within a particular range...
- List 5: **for** each row, go through all the columns and record each one

To demonstrate how many operations that represents, consider the following:

- list 1 grabs a number of links (spreadsheets)
- Each of those 3 spreadsheets has around a dozen sheets
- Each of those dozen spreadsheets (times 3) has a couple hundred rows
- Each of those couple hundred rows (times a dozen, times 3) has anywhere between 8 and 18 columns

So we're looking at a number of spreadsheets times 12 sheets times hundreds of rows times around a dozen columns.

I guess it was worth it.

When does a scraper become an attack?

Some *very* heavy scraping - hitting a site with thousands of simultaneous requests for information - might cause a site to slow down so badly that its performance is noticeably impaired. This could be considered a 'cyber attack'. However, that sort of scraper is not described anywhere in this book, and the tools outlined are unlikely to be able to run such a scraper. If you do ever find yourself in a situation where such a scraper is needed, however, typical practice is to 'pace' the scraper so it proceeds slowly and/or periodically in order not to overwhelm the website being scraped (which ultimately you wouldn't want anyway, as the information would then no longer be available). This is good ethical scraping too. Stacey Higginbotham's article on the subject[15] provides a good overview.

Law is always changing in this respect, as in so many others, so follow legal developments in your own country by setting up an email alert for mentions of 'scraping law' in the news.

---

[15]http://gigaom.com/2012/04/22/as-unstructured-data-heats-up-will-you-need-a-license-to-webcrawl/

Scraper tip: creating a sandbox

If you're going to make a change to a large or complex scraper, you can save time by testing it in a 'sandbox' scraper.

After all, you don't want to wait for hundreds of data records to be scraped and saved before your scraper gets to the new code and then fall over.

Remember that Scraperwiki is a coding environment that can be used for any code - not just scrapers.

If you want to test the code that creates two separate datasets, for example, you can do just that - and only that - in a separate scraper. Here's the code for that[16]:

```
import scraperwiki
#create an empty dictionary variable called
'record'
record = {}
#create a list variable called 'testlist1' and
put 3 names in it
testlist1 = ["rod","jane","freddy"]
#loop through that list and put each item in the
variable 'name'
for name in testlist1:

  #add a new field called "name" to the
  'record' variable, and assign the value
  of the 'name' variable to that field

   record['name'] = name
```

---

[16]https://scraperwiki.com/scrapers/testingrecords/

```
    #save 'record' in the Scraperwiki datastore:
    use the "name" field as the unique key,
    and name the dataset 'testdataset1

     scraperwiki.sqlite.save(["name"], record,
     'testdataset1')

    #print  the  contents  of  the  variable
    'record'

     print record

  #create  a  second  empty  dictionary  variable
called 'secondrecord'
  secondrecord = {}
  #create a list variable called 'testlist2' and
put 4 names in it
  testlist2 = ["paul","george","ringo","john"]
  #loop through that list and put each item in the
variable 'thing'
  for thing in testlist2:

    #add a new field called "name2" to the
    'record' variable, and assign the value
    of the 'thing' variable to that field

     secondrecord['name2'] = thing

    #save 'secondrecord' in the Scraperwiki
    datastore:  use  the  "name2"  field  as
    the  unique  key,  and  name  the  dataset
    'testdataset2
```

```
scraperwiki.sqlite.save(["name2"], secondrecord,
'testdataset2')

#print the contents of the variable
'secondrecord'

print secondrecord
```

Recap

- **Dates are stored in Excel as a number**, like 41219.0, which represents the number of days since 1900 or 1904.

- To convert those numbers to a date we can recognise, **find and re-use code that solves that problem**, in this case on the Scraperwiki documentation. Make sure you 'call' it on the cell or series of cells rather than the value or values in the cell(s), and specify the `datemode` (0 for 1900 or 1 for 1904).

- As you cycle through a row of cells you can add them to a list by using the `.append` method like so: `listname.append(NameOfVariableGeneratedByForLoop)`

- `datetime` is a Python library which has a whole lot of useful methods and functions for interpreting dates.

- You can comment out a section longer than a line by putting three quotes or inverted commas before and after it like so: `"""` or `'''`.

- **Use the `str()` function to convert a date into a string**.

- If you hit an error after a few hundred rows of data and want to test a solution, **try temporarily changing the range of rows, columns or sheets that you're looping through** so that you hit the problem (and hopefully solve it) early on.
- There are lots of *string formatting methods* that can be used to clean up text or remove problematic elements that trip up your scraper (such as inverted commas or slashes), or even test or measure string properties, such as the case, length, and so on.
- **The `.replace` method can be used to replace a problematic character with something else** (including 'nothing') - use it to replace slashes, quotation marks and inverted commas in keys.
- **Either single or double quotes can be used to indicate the start and end of a string**. This enables you to include inverted commas in the string itself (if you use double quotes to indicate the start and end), or include quotation marks in a string (by using single quotes to indicate that string's beginning and end). The string "My head's hurting", for example, has an inverted comma, so the *string* is identified with double quotes; but the string 'Jamie said "programming rules"' includes quotes, so that uses single quotes to identify the start and end of the string.

This thread on Stack Overflow[17] suggests that there is no difference between either choice of single or double

quotes, but some people have adopted particular conventions. One particular response also shows examples of how to 'escape' quotes with slashes or use triple quotes where you have more than one type[18] of quote in a string.

- Always **check your raw data before publishing** specific 'facts' about it gleaned from your scraped data.
- When you switch from scraping a single spreadsheet to scraping multiple linked spreadsheets, remember to **indent all your previous spreadsheet-specific code in a new function** that can be run on *every* spreadsheet link.
- When you switch from scraping a single spreadsheet to scraping multiple linked spreadsheets, remember to *also* **change your unique key so that it will remain unique when more than one spreadsheet is scraped**. You might do this by having more than one key (such as URL and code), or creating an arbitrary key, like an index counter.
- When you have rows within sheets within spreadsheets within pages within a site you'll end up with confusing **loops within loops**. Don't be confused: you'll get used to it.
- If you need to try a new technique on a big scraper, **try creating a separate 'sandbox' scraper to test the technique out in a simple way first**.

---

[18]http://stackoverflow.com/a/4776742

- You can also speed things up by first **limiting the ranges of things being grabbed** - links, sheets, and rows - to check it all works properly before letting it run for hours grabbing everything.

Tests

- Try to sketch on paper the loops that this or another scraper goes through in its code.
- Try to scrape one of the sheets with headers across two rows. How would you combine the values using `for` loops and `if` tests? Try to describe the pattern in that particular data and use that as the basis for your code.
- Find the documentation for `datetime`: what other methods and functions might be worth knowing about?
- Try different unique keys and dataset names - why do some cause problems?

32 Scraper 29: Scraping ASPX pages

If a page you're trying to scrape ends in .aspx, you should allow yourself a loud groan: ASPX pages are some of the most challenging of all to scrape, as we'll see.

Take, for example, this page of search results from a database of clinicians[1], shown above. Go to that page and look carefully at the URL.

Now, scroll to the bottom of the page and click the 'Next' link. What happens to the URL? Nothing. But the page changes. So how do we scrape that?

With all the experience gained in the previous chapters you should be well equipped to tackle many of the prob-

---

[1] http://recognition.ncqa.org/PSearchResults.aspx?state=NY&rp=

lems involved - and know how to seek out solutions to new ones.

In this final scraper we're going to go 'undercover' again, as our scraper pretends to be a browser in order to fool a website into giving it some information. To do that we'll be revisiting the last of the Scraperwiki live tutorials[2], which shows you some code for scraping ASPX pages. I've copied that code here[3] too. Let's get stuck in with the opening section of code...

The code

```
import scraperwiki
    import mechanize
    import re
    # ASPX pages are some of the hardest challenges
because they use javascript and forms to navigate
    # Almost always the links go through the function
function __doPostBack(eventTarget, eventArgument)
    # which you have to simulate in the mechanize
form handling library
    # This example shows how to follow the Next page
link
    url = 'http://recognition.ncqa.org/PSearchResults.aspx?state=N
    br = mechanize.Browser()
     # sometimes the server is sensitive to this
information
```

```
    br.addheaders = [('User-agent',  'Mozilla/5.0
(X11; U; Linux i686; en-US; rv:1.9.0.1) Gecko/2008071615
Fedora/3.0.1-1.fc9 Firefox/3.0.1')]
    response = br.open(url)
```

These first 16 lines should be familiar to you from the chapters on mechanize and the regex library re.

Those libraries are used here to mimic a browser and open a URL, which is then put in the variable response.

Next we have a for loop which creates a number - pagenum - that starts at 0 and ends at 10 (a range generated by the range function), so it loops 10 times:

```
    for pagenum in range(10):

      html = response.read()

       print "Page %d page length %d" %
    (pagenum, len(html))

     print "Clinicians found:", re.findall("PDetails.aspx\?Provi
    html)

     mnextlink = re.search("javascript:__-
    doPostBack\('ProviderSearchResultsTable1\$NextLinkButton','
    Page", html)

      if not mnextlink:

          break

      br.select_form(name='ctl00')
```

```
     br.form.set_all_readonly(False)

     br['__EVENTTARGET'] = 'ProviderSearchResultsTable1$NextLink

     br['__EVENTARGUMENT'] = ''

     response = br.submit()
```

What needs unpicking? Well, the regex following `print`
`"Clinicians found:"`, might be intimidating at first:

```
re.findall("PDetails.aspx\?ProviderId.*?>(.*?)</a>",
html)
```

This uses the `.findall` function in `re` to find all matches
for the regular expression `"PDetails.aspx\?ProviderId.*?>(.*?)</a>"`
in the object `html` (created two lines earlier by 'reading'
`response`)

Most of that regular expression is literal, but look for
the question marks, asterisks and other 'special' characters:

```
"PDetails.aspx\?ProviderId.*?>(.*?)</a>"
```

This means:

`PDetails.aspx` followed by `\` which *escapes* the next
character, `?` (normally this would mean 'zero or one of a
character', but because it's escaped it literally means "?")

Then `ProviderId` followed by `.` (any character) and `*`
(zero or more of that character) followed by `?` (zero or one
of that character), followed by `>`.

The next line uses regex's `.search` function to look in
`html` for some javascript (the 'next' link):

```
    mnextlink = re.search("javascript:__-
  doPostBack\('ProviderSearchResultsTable1\$NextLinkButton','
  Page", html)
```

Using CTRL+F to *find* part of that in the HTML, here's
what it looks like in full there:

```
<td class="TableFooter" width="150"><a id="ProviderSearchResul
NextLinkButton" OnMouseOver="window.status='Next Page';
return true" OnMouseOut="window.status=''; return
true" href="javascript:__doPostBack('ProviderSearchResultsTable1$
Page >></a> </td>
```

Note that the link *text* is 'Next Page' but the link
URL itself - what comes after href=" is some javascript:
`javascript:__doPostBack('ProviderSearchResultsTable1$NextLinkBut`

This javascript is typical of the problem aspx scrapers
must tackle.

The remainder of the code essentially 'clicks' that 'Next
Page' link as if it was your browser, after an if test to check
that the link is there:

```
        if not mnextlink:

            break

        br.select_form(name='ctl00')

        br.form.set_all_readonly(False)

        br['__EVENTTARGET'] = 'ProviderSearchResultsTable1$NextLink

        br['__EVENTARGUMENT'] = ''

        response = br.submit()
```

How does it do this?

It will help us if we decode some of the terms used in this passage of code. In doing so we're going to pick up a little javascript - but remember, you don't need to learn a whole language if all you need to do is ask one question.

Submitting links in javascript

The link that's causing us a problem starts with a statement that what follows is javascript:

```
javascript:__doPostBack('ProviderSearchResultsTable1$NextLink
```

A quick search for details on __doPostBack brings up pages like this one[4], which says:

> "this __doPostBack function … takes two arguments:
>
> "1) eventTarget
>
> "2) eventArgument"

Well we know about functions, and we know about arguments (also called parameters - the ingredients that a function works with). This helps us understand that what's in parentheses after __doPostBack is those arguments. The first is 'ProviderSearchResultsTable1$NextLinkButton', and the second (after the comma) is empty: ''

These arguments/parameters are also used/set in our own scraper:

---

[4]http://wiki.asp.net/page.aspx/1082/dopostback-function/

```
br['__EVENTTARGET'] = 'ProviderSearchResultsTable1$NextLink

br['__EVENTARGUMENT'] = ''
```

So if we were re-creating this scraper for a different page, we would look at what arguments the original HTML page contained, and use those in our code.

You'll notice that the line finding that 'Next Page' link and storing it actually does nothing other than test whether such a link exists (and to break off if there is no 'next' link). The values of that link are not used - we've set them ourselves.

The code also does two other things: it selects the form with the name 'ctl00' (mechanize "must have a form selected" to work, according to this handy cheat sheet[5] - look in the source HTML of the page to find 'ct100'), and sets the form's 'all_readonly' status to False (which allows things to be written to br).

Finally, it 'clicks' the link:

```
response = br.submit()
```

This represents the end of the for loop, so it will now repeat for the second number in range(10) (i.e. 1), run through all the commands again, printing the length of the html object:

```
Page 1 page length 89851
```
...and all matches of the regex:

---

[5]https://scraperwiki.com/views/python_mechanize_cheat_sheet/edit/

```
Clinicians found: ['139 Medical Facility, PC',
'9317 Medical Office PC', 'A Plus Medical Care,
PC', ...
```

Before submitting that javascript link to the next page again, and so on until it hits the end of the loop's specified range (this could be changed if you expected more result pages).

Saving the data

The tutorial scraper, of course, doesn't save the data it grabs, so we'll need to add some more lines to do that.

Firstly, we need to create an empty record - *before* the loop runs (otherwise a new empty record will be created each time):

```
record = {}
```

We're going to need a unique key for this data, but because we're only storing one piece of data (a name) and there may be entries with the same name, we need another field.

So, after that new empty record, we'll set an index number at zero which, as before, we'll increase by one for every new line and store alongside our data

```
id = 0
```

Now, when this scraper first ran, for each loop it printed a **list** (note the square brackets) of names like so:

```
Clinicians found: ['139 Medical Facility, PC',
'9317 Medical Office PC', 'A Plus Medical Care,
PC', ...
```

We need to **loop** through that list and store each item. This means we need *another* `for` loop within the loop we already have, *after* the line

```
html = response.read()
```

Here's the extra code:

```
 for name in re.findall("PDetails.aspx\?ProviderId.*?>(.*?)<
html):

        id = id+1

        record['ID'] = id

        record['Name'] = name

        print record

        scraperwiki.sqlite.save(['ID'],record)
```

This should be familiar by now, but just for clarity, here is what those lines are doing:

- Looping through the list that results from using `re.findall` and putting each result into a variable called `name`
- Adding 1 to the variable `id`,
- Storing that value in the field 'ID'
- Storing the value of `name` in the field 'Name'
- Printing our full record (the ID and Name)

- Saving the record in Scraperwiki's datastore, with 'ID' as the unique key.

You can see the adapted scraper with the new lines commented here[6].

And with that final test met, it is time to draw a line under this book with our last chapter - but not before a final recap...

Recap

ASPX pages can present all manner of problems, some of which defeat even the most able programmers. This chapter has just covered some basic principles of using mechanize to scrape ASPX pages. The key points are:

- ASP.net pages end in .aspx and often **do not have individual URLs**.
- Your scraper will need to 'click' **Next links** and store the pages as they go. It can do this with the **mechanize library**.
- You can collect the data into a list if it follows a particular pattern by using **findall from the re library**.
- **Loop through the list** of results, saving each one.
- You can also identify the 'Next' links with **regex using the re library**. Loop through pages collecting the data *until there are no more Next links.*

---

[6]https://scraperwiki.com/scrapers/aspx_livetutorial_savingrecords/

- **`__doPostBack` is a javascript function** which needs two ingredients (arguments, or parameters). These are set in your scraper with `br['__EVENTTARGET'] =` and `br['__EVENTARGUMENT'] =` - **look in the source code of the page for the values for each**.
- Learning a little bit of javascript - particularly around creating online forms - will help you work with scrapers that replicate the same processes.

Tests

- Try to use `lxml.html` instead of regex to find the parts you want to store.
- Find some tutorials on creating forms with javascript - what can you learn from these in scraping aspx pages?
- Look at some of the alternative methods listed in this Stack Overflow thread[7] - try some of them out.
- You'll find all sorts of problems and solutions on threads tagged with 'asp.net' and 'mechanize'[8] - see if any apply to your problems.

---

[7]http://stackoverflow.com/questions/6116023/screenscaping-aspx-with-python-mechanize-javascript-form-submission

[8]http://stackoverflow.com/questions/tagged/asp.net+mechanize

33 The final chapter: where do you go from here?

Remember starting from a single line in a Google spreadsheet? Ah, those were the days. Now you're a hard-bitten hack(er) who can pull the needles from a haystack of documents, or tell a script how to 'go undercover' and pretend to be Internet Explorer (someone has to do it).

You've learned how regular expressions can find patterns in text, and how free and cheap tools like Outwit Hub and Google Refine can sometimes get a simple job done more quickly when you don't have time to phrase lines of code.

In order to tackle the tougher jobs, you learned quite a bit of programming: **functions** and **variables**; **indexes** and **ranges**; **loops** and **if/else** tests.

You've covered the difference between a **string** and an **integer**; a **list** and a **dictionary**; and just some of the libraries that can help us tackle different problems.

That's the 'how to program' bit - but perhaps more importantly, you've also learned how to be a *programmer*. Specifically:

- How to look for **structure** in the documents you are trying to scrape;

- How to look for solutions that have already been found by **other people**;
- And how to break down your own solution into distinct problems and stages, and test those through **trial and error**

Those three elements - an eye for structure, the ability to stand on the shoulders of giants, and trial and error - are what will take you through your own journey from here. Because, even after 476 pages and 77,200 words of this book, there will always be new problems, and new solutions to find, and new experiments to make. No two websites are the same (if they are, they only need one scraper!).

You know what I'm going to say, don't you? *"This is not the end, but just the beginning..."*?

The map is not the territory

When I began this book I wanted to map what could be an intimidating and foreign territory; a place full of strange languages and unfamiliar markings. Now that it no longer seems that way you will have particular parts you want to explore further: the dark alleyways of PDFs; the rolling hills of freeform text; the hidden gardens of databases hidden behind forms.

As I have said throughout the book, it is better to be guided by your most pressing problems than to do things just because you 'think you should'. Those problems will be frustrating at times - but their solutions will be

revelatory and empowering at others. A problem solved is more satisfying than a task completed.

You will not solve them all, but be proud of those you do. This is a rare skillset, and every success should be celebrated. Failure is *part* of that success (let me tell you, having failed at many things many times, and seen others do the same, I now think of failure as really just a *deferred success*; the memory of the attempt often returns when you stumble across a solution to your problem later on).

Having used programming to tackle scraping problems, you may want to read more widely about programming in general. You'll find that concepts which may have seemed abstract before now strike you as useful in the situations you've experienced: the `range` function (to generate a range of numbers or characters), for example, might have previously seemed great for maths but not much use for scraping. However, it's a great solution for appending page numbers to search results.

You may stick with Python or look at a different language: many organisations use Ruby on Rails; some coders are more comfortable with PHP (which will also allow you to fiddle with Wordpress).

Having learned some Python in this book it should be much easier to pick up any other 'object oriented language' like Ruby or PHP. Concepts are shared (variables, functions, libraries), techniques are the same (loops, if/else tests, identifying patterns), only the language differs - and not so much anyway.

Whichever language you choose, there are many more

libraries to try out: check out `Beautiful Soup` for an alternative to `lxml`, or `Requests` as an alternative to `Mechanize`. Or look at the libraries that solve different problems.

If you're API and you know it

One area that I would recommend looking into is **API**s (Application Programming Interfaces). APIs allow you to ask questions of data in a more straightforward way (every time you use a Twitter, Facebook or Google Maps app your app is using their API to ask questions of their data). Often this is done by forming URLs containing the query and the value - a process you may remember from previous scrapers.

The numbers of APIs are growing, not just in social media (Twitter's API even has its own YouTube channel[1]), but in government, healthcare, transport, arts and commerce: the UK police have a crime API[2]; Data.gov.uk does[3] too. The British Museum[4]; railfare prices[5]; the Russian Central Bank[6]!

Civic organisations like TheyWorkForYou.com[7] and Sunlight Labs[8] both have APIs which provide structured

---

[1] http://www.youtube.com/user/twitterapi
[2] http://www.guardian.co.uk/technology/blog/2010/mar/04/crime-mapping-api-uk
[3] http://data.gov.uk/data/api
[4] http://www.programmableweb.com/api/british-museum
[5] http://librailfare.sourceforge.net/
[6] http://www.cbr.ru/
[7] http://www.theyworkforyou.com/api/
[8] http://services.sunlightlabs.com/

information about politicians and politics in the UK and US respectively. OpenSpending[9] has one for government spending around the world.

Directories of APIs can be found at ProgrammableWeb[10] and Mashape[11] but the simplest way to find one is by using a search engine to look for 'API' and your problem, e.g. 'text' or 'postcodes'.

In journalism, APIs are becoming increasingly important: not only in news organisations from The Guardian to the New York Times (whose APIs allow you to link their content to any other data you can think of, including maps), but also less-known names like n0tice[12] and the planned collaboration-supporting API from NPR[13].

You can use APIs *within* your scraper, either to fetch data or to complement other data your scraper is collecting.

In Chapter 23, for example, I looked at a scraper by journalist David Elks which gathered data from PDFs showing speed camera locations. David added to that scraper[14] so that it also queried the Yahoo API to get geolocation information on every location listed in that data.

In other words, it 'scraped' the API for a second set of data, which was combined with the first, saving time having to attempt that in a spreadsheet later on.

---

[9] http://openspending.org/help/api.html

[10] http://www.programmableweb.com/apis

[11] https://www.mashape.com/explore/All?page=1

[12] http://n0tice.org/developers/api-documentation/

[13] http://www.poynter.org/how-tos/digital-strategies/e-media-tidbits/105504/public-media-api-could-be-engine-of-innovation-for-journalism/

[14] https://scraperwiki.com/scrapers/pdftoxmltest/

You can apply the same principles to use a sentiment analysis API[15] in your scraper, to convert URLs into IP addresses[16], or even employ facial recognition or email address analysis[17], or just to shorten URLs[18]. The Google Visualisation API can also be used to create a chart[19]. *I've not done most of these, by the way, but they sound like great ideas to try.*

You can even create your own API with tools like PopIt[20] - or Scraperwiki itself (ProgrammableWeb suggests using Scraperwiki as a way to create an API[21] for content which doesn't have one).

Scraperwiki's API allows you to query your own scraper's data and publish it as an RSS feed, HTML table or CSV file - or even use the results of that query as part of another scraper (for example a list of links to scrape).

Of course, scraping is not always the best solution to an information-gathering problem. The NICAR-L mailing list for computer-assisted reporters regularly features examples of difficult websites where the data turns out to be available at an easily-missed 'download' link, or via a Freedom of Information request, or - yes - a simple

---

[15]http://stackoverflow.com/questions/7820126/list-of-sentiment-analysis-tools-and-apis

[16]https://www.mashape.com/jack/dns-tools#!documentation

[17]http://www.rapleaf.com/

[18]https://www.mashape.com/mashaper/bitly#!documentation

[19]http://blog.ouseful.info/2012/04/03/exporting-and-displaying-scraperwiki-datasets-using-the-google-visualisation-api/

[20]http://popit.mysociety.org

[21]http://blog.programmableweb.com/2010/08/23/no-api-use-scraperwiki-to-add-one/

phonecall.

(The phone really is an under-rated social media technology...)

Recommended reading and viewing

For more on Python there are a number of free books online, including How to Think Like A Computer Scientist[22], Dive Into Python[23] and my own favourite Learn Python The Hard Way[24], part of the wider Learn Code The Hard Way series[25] which includes books on SQL, regex, C and Ruby.

Dan Nguyen has written a number of books which are all great reading, including:

- Coding for Journalists: A Four Part Series[26]
- The Bastards Book of Ruby[27]
- The Bastards Book of Regular Expressions[28]

Nguyen also recommends Exploring Everyday Things with R and Ruby[29] (this post from the author[30] provides a

---

[22]http://www.greenteapress.com/thinkpython/thinkpython.html
[23]http://www.diveintopython.net/
[24]http://learnpythonthehardway.org/
[25]http://learncodethehardway.org/
[26]http://danwin.com/coding-for-journalists-a-four-part-series/
[27]http://ruby.bastardsbook.com/
[28]http://regex.bastardsbook.com/
[29]http://www.amazon.com/Exploring-Everyday-Things-Ruby-Learning/dp/1449315151
[30]http://blog.airbrake.io/guest-post/exploring-everything/

taster). And he recommends Interactive Data Visualization for the Web[31] as a good introduction to Javascript.

More than one contributor to the NICAR-L mailing list has recommended the Head First series of books[32] from O'Reilly, particularly Head First Programming[33].

You can find screencasts about Ruby on Rails at Railscasts[34] and podcasts about Ruby, Javascript and other tools at PeepCode[35]. Videos about Javascript at the new Boston[36].

Michelle Minkoff, the source of all those recommendations, has also written a useful post on self-teaching programming which includes more resources[37]

Thinkstats[38] is a useful (and free) book for learning the statistical/visualisation language R, using Python from your computer, and picking up some statistics along the way. The Fourth Paradigm[39] covers broader issues around statistics.

To learn more about **APIs** in journalism check out the Sunlight Foundation's Sunlight Academy video on unlocking APIs[40], and their Codecademy course on using

---

[31]http://shop.oreilly.com/product/0636920026938.do

[32]http://shop.oreilly.com/category/series/head-first.do

[33]http://headfirstlabs.com/books/hfprog/

[34]http://railscasts.com/

[35]https://peepcode.com/

[36]http://thenewboston.org/list.php?cat=10

[37]http://michelleminkoff.com/2010/03/25/self-teaching-data-and-programming-skills/

[38]http://greenteapress.com/thinkstats/

[39]http://research.microsoft.com/en-us/collaboration/fourthparadigm/4th_paradigm_book_complete_lr.pdf

[40]http://training.sunlightfoundation.com/module/unlocking-api/

their own Capitol Words API[41]. Poynter also has a guide for journalists to understanding API documentation[42].

You can find documentation on the Scraperwiki API[43], and a video on using the API by Nicola Hughes here[44].

ProPublica's post about their 'Dollars for Docs' investigation[45] provides useful insights into how a team of developers tackled one of the most horrible scraping challenges. If nothing else it should make you feel better about the site you're scraping.

Useful blogs to follow include:

- Michelle Minkoff's own blog[46]
- Mindy McAdams's guides to Python and Javascript at Baby Steps in Data Journalism[47]
- Tony Hirst, who often writes about Scraperwiki, network analysis and R on his blog OUseful.Info[48];
- and OpenNews Guardian fellow Nicola Hughes blogs at Data Miner UK[49].

Useful communities to join include Scraperwiki's own

---

[41]http://www.codecademy.com/courses/python-intermediate-en-D56TP?curriculum_id=50ecbb00306689057a000188

[42]http://www.poynter.org/how-tos/digital-strategies/138211/beginners-guide-for-journalists-who-want-to-understand-api-documentation/

[43]https://scraperwiki.com/docs/api

[44]http://www.youtube.com/watch?v=kjZgx00KBu0

[45]http://www.propublica.org/nerds/item/heart-of-nerd-darkness-why-dollars-for-docs-was-so-difficult

[46]http://michelleminkoff.com/

[47]http://babydatajournalism.tumblr.com/

[48]http://blog.ouseful.info/

[49]http://datamineruk.com/

mailing list[50], and NICAR-L[51], and you can follow threads on StackOverflow relating to specific languages and/or tools - for example those tagged 'Python' *and* 'scraper-wiki'[52].

It's also a good idea to blog about your own experiences, problems, and solutions: blogging is a good way for others to find you (including potential employers), and to record things for yourself.

That's plenty to be going on with - but if I find more (or you tell me about them), I'll add them in future updates.

End != End

This project has grown from a planned book chapter, to a mini ebook, to one that was rather larger than I anticipated. As people used it, I learned from what they did. I've enjoyed that - and I hope you continue to let me know what you do with it, and what it needs to do for you.

The beauty of this form of publishing is that the book is never 'finished': I will continue to update it, perhaps add to it, and tweak it. Make sure you're signed up to receive update emails for when new versions are pushed out, or check the Facebook page[53].

Meanwhile, I have started work on two other related books - a quick and simple, more rounded introduction to

---

[50]https://groups.google.com/forum/?fromgroups=#!forum/scraperwiki
[51]http://www.ire.org/resource-center/listservs/subscribe-nicar-l/
[52]http://stackoverflow.com/questions/tagged/python+scraperwiki
[53]https://www.facebook.com/ScrapingForJournalists

basic data journalism skills based on my half day training sessions, called The Data Journalism Heist[54]; and a book tackling the surprisingly varied problems in cleaning data[55] once you have it (I think I'll call it *Making Data Play Nice* but I'm open to ideas on this one).

If you are interested in either please sign up on the Leanpub page for that book.

Thanks for buying this book. I hope I'll see you in the next one...

---

[54]http://leanpub.com/datajournalismheist
[55]https://leanpub.com/cleaningdirtydata

34 Bonus scraper #1: Scraping XLSX spreadsheets with Openpyxl

Not all spreadsheets are created equally. Just as we needed different libraries to scrape CSV files and XLS files, you'll need another library again if you come across XLSX files.

Our case study this time is politicians' expense claims

- specifically, members of the House of Lords[1] in the UK.

These are published on a series of pages on the UK Parliament website[2], in both PDF and Excel spreadsheet format: one for every month.

Trying to use the xlrd library on the spreadsheets, however, doesn't work because these are not XLS spreadsheets but the newer XLSX type (you can read more about the format on File.org[3]).

As always, there are a number of possible solutions here. We could scrape the *PDF files* instead, or even find out a way to *automate conversion* of the XLSX files to XLS format, and then use xlrd as before (a quick search for "python automate conversion xlsx to xls" yields a tutorial on the blog *Gee... Oh! Spatial*[4].

But this is a good excuse to see if there is a Python library devoted to our problem, and learn how to use it.

A search on Google for Python library xlsx will bring up a few options. A search for python library "xlsx" scraperwiki (the "xlsx" in quotes ensures that the actual pages contain that word, rather than other pages linking to it) will find some that are more specific to Scraperwiki.

This includes a blog post about a Scraperwiki data camp[5] by Paul Winkler where he describes exactly our

---

[1] http://www.parliament.uk/business/lords/

[2] http://www.parliament.uk/business/lords/whos-in-the-house-of-lords/house-of-lords-expenses/

[3] http://file.org/extension/xlsx

[4] http://geeohspatial.blogspot.co.uk/2013/04/batch-xls-or-xlsx-to-csv-with-python.html

[5] http://openplans.org/2012/02/liberating-data-at-journalism-data-camp-ny/

problem:

> "I needed a Python library that could open Excel 2007 XLSX files; a quick search found that one existed[6], but ScraperWiki didn't have it in their cloud environment. I mentioned this to [CEO] Francis [Irving] and five minutes later it was available to all ScraperWiki users."

So we know the library to solve our problem - and as it happens, one of the other results of our search is a mention of openpyxl on Scraperwiki's documentation on 3rd party libraries[7]. The next step is to click on that link to the documentation for openpyxl[8] to find out more.

Standing on the shoulders of Zarino Zappia

Extra help could be found by searching for another scraper which already uses the same library for a similar purpose. This time the search would be openpyxl site:scraperwiki.com.

One of the results (at the time of writing) is a scraper by Scraperwiki programmer Zarino Zappia called "Corresponding Local Authorities"[9]. As long as it works (and it does) we can try to borrow something from it and work back from that to fill in holes in our knowledge.

---

[6] http://pythonhosted.org/openpyxl/index.html
[7] https://classic.scraperwiki.com/docs/python/python_libraries/
[8] http://pythonhosted.org/openpyxl/index.html
[9] https://classic.scraperwiki.com/scrapers/corresponding_local_authorities/

Using CTRL+F to find any mentions of `openpyxl` we can spot the following lines to begin with:

`import openpyxl`
and:

```
wb = openpyxl.load_workbook(f.name)
```

The first line obviously imports the library so it can be used. The second line uses something called `.load_workbook` from that library (we can guess because it's attached by that period).

Again, we could search Google for `openpyxl load_workbook`, or look in the documentation we've already found.

Not surprisingly, that tells us[10] that `.load_workbook` is used to open an existing (spreadsheet) workbook. The results are put into the variable `wb`.

But rather than spend time deconstructing *that* scraper, let's start with the scraper that grabs the Lords' expenses data, and explain how that works.

The scraper in full

Here's the code in full we're going to be looking at, including explanatory comments:

```
#This line tells Scraperwiki that the scraper
is written in Python
#!/usr/bin/env python
```

---

[10]http://pythonhosted.org/openpyxl/tutorial.html#loading-from-a-file

```
    #Scraper forked from Zarino Zappia's scraper at
https://classic.scraperwiki.com/scrapers/corresponding_-
local_authorities/
    #With help from Paul Tremberth on the Scraperwiki
mailing list who improved some code: https://classic.scraperwiki.c
expenses_1/
    import scraperwiki
    import requests
    import openpyxl
    import tempfile
    import lxml.html
    import pprint
    def getworkbook(url):

      #use 'get' from the 'requests' library
     with the url and '.content' - put into
     'raw'.

      #see http://docs.python-requests.org/en/latest/user/
     quickstart/#make-a-request

      raw = requests.get(url, verify=False).content

      #uses the 'tempfile' library and 'NamedTemporaryFile'
     to store temporary information in a new
     variable 'f'

      #see http://docs.python.org/2/library/tempfile.html
     for more on that library

      f = tempfile.NamedTemporaryFile('wb')
```

```
#uses '.write' to add 'raw' to that

f.write(raw)

#uses '.seek' to start from '0' - see
http://www.tutorialspoint.com/python/file_-
seek.htm

f.seek(0)

#now use the library openpyxl's .load_-
workbook function to grab f.name and put
into 'wb'

wb = openpyxl.load_workbook(f.name)

#close the temporary file 'f'

f.close()

#return 'wb' to whatever called this
function (now go back to the function
def import_data below)

return wb

def getdata(wb, sheet='Sheet1'):

#Turn a sheet on a workbook into an
array

data = wb.get_sheet_by_name(sheet)
```

```python
#create an empty list 'stripped_table'

stripped_table = []

#loop through each '.rows' item in
'data', call it 'row'

for row in data.rows:

    #create an empty list 'stripped_-
    row'

    stripped_row = []

    #loop through each item in
    'row', calling it 'cell'

    for cell in row:

        #grab the .value of
        'cell', use 'unicode'
        and '.strip', put into
        'stripped_cell'

        stripped_cell = unicode(cell.value).strip()

        #append the 'stripped_-
        cell' to 'stripped_-
        row'

        stripped_row.append(stripped_-
        cell)
```

```
        #after that loop, append 'stripped_-
      row' to 'stripped_table'

       stripped_table.append(stripped_-
      row)

   #return 'stripped_table' to whatever
  called this whole function

   return stripped_table

def import_data(linkurl):

   #create variable 'wb' containing the
  results of running 'getworkbook' on
  'linkurl' (the spreadsheet) - see def
  getworkbook above

   wb = getworkbook(linkurl)

   #use .get_sheet_names on 'wb' and put
  results into 'sheet_names'

   sheet_names = wb.get_sheet_names()

   #if 'sheet_names' does not exist (i.e.
  there are no sheets to put into it)

   if not sheet_names:

       #return (nothing) to whatever
      called this whole function
```

```
        return

 #otherwise, assume there's only 1 sheet,
and get  first  one  (item  0  in  'sheet_-
names')

 rows = getdata(wb, sheet_names[0])

 #run .pprint from the 'pprint' library
on 'rows' - see http://docs.python.org/2/library/pprint.html

 pprint.pprint(rows)

 #create empty dictionary variable 'expenses'

 expenses = {}

 #create empty list variable 'headings'

 headings = []

 #loop through range 0-18 and assign that
number to variable 'num', then for each:

 for num in range(0,18):

      #The spreadsheets either have
      headings on row 6 or 7

       #so  these  lines  test  which
      one  contains  'Name'  in  the
      first [0] item (cell), and put
      results into 'target_row'
```

```
 if rows[5][0] == "Name":

     target_row = rows[5]

 elif rows[6][0] == "Name":

     target_row = rows[6]

 else:

     print "NEITHER ROW?",
    linkurl

     continue

 #Now print each item (cell)
for that row - note we're still
looping from 0-18 (each cell)

 print "rows[5/6][num]", target_-
row[num]

  #various  slashes,  brackets
and full stops caused problems
as keys, so this line replaces
them all

 cleanedheading = target_row[num].replace("/","_-
").replace("No.","Number").replace("(","_-
").replace(")","_").replace("&","and")

 print "cleaned heading:", cleanedheading
```

```
        headings.append(cleanedheading)

        print "headings:", headings

 #There are around 805 rows in the sheet,
so this goes through each, from 8 to row
805, putting each number in 'i'

 for i in range(7,805):

        #print that row, and the first
        [0] item, this should be the
        name

         print rows[i][0]

         #assign the linkurl for the
        whole spreadsheet (passed at
        the start of this function)
        into the 'linkurl' field of the
        'expenses' dictionary

        expenses['linkurl'] = linkurl

 #loop through range 0-18 and assign that
number to variable 'num', then for each:

        for num in range(0,18):

            #grab that number item
            from the 'headings' list
            and use as a field
            name,
```

```
            #and  assign  it  the
         row with the 'i' value
         (note  that's  a  loop
         we're  in  too,  above,
         so  that  will  repeat
         too)

          #and the number within
         that  row,  from  this
         range

          expenses[headings[num]]
         = rows[i][num]

       #the line below must use the
      heading 'Name' with a capital
      'N', as in the sheets

       scraperwiki.sqlite.save(['Name',
      'linkurl'], expenses, 'expenses')

  def grabexcellinks(URL):

     #Use Scraperwiki's scrape function on
    'URL', put results in new variable 'html'

     html = scraperwiki.scrape(URL)

     #and show it us

     print html
```

```
   #Use lxml.html's fromstring function
on 'html', put results in new variable
'root'

 root = lxml.html.fromstring(html)

 #use cssselect method on 'root' to grab
contents  of  <div  class="rte"><li><a>
tags

 #where the last characters within the
href= property are 'xlsx'

 #and put in a new list variable 'links'

  links = root.cssselect('div.rte  li
a[href$="xlsx"]')

 #for each item in that list variable,
from the second to the last [1:], put it
in the variable 'link'

 for link in links:

     #and print the text_content
     of that (after the string "link
     text:")

     print "link text:", link.text_-
     content()
```

```
           #use  the  attrib.get  method
        on  'link'  to  grab  the  href=
        attribute of the HTML, and put
        in new 'linkurl' variable

        linkurl = link.attrib.get('href')

        #print it

        print "link type:", linkurl[-4:]

        print "SCRAPE IT!"

        #see 'def import_data' to see
        what happens to that url

        import_data('http://www.parliament.uk'+linkurl)

    #Create a list of 2 pages containing links to
expenses spreadsheets
    expenseslinks = ['http://www.parliament.uk/business/
lords/whos-in-the-house-of-lords/house-of-lords-expenses/',
'http://www.parliament.uk/mps-lords-and-offices/members-
allowances/house-of-lords/holallowances/hol-expenses04/
201112/']
    #Loop through that list and run the function
'grabexcellinks' on each item:
    for expenseslink in expenseslinks:

        #see 'def grabexcellinks' above to see
        what happens to that link

        grabexcellinks(expenseslink)
```

Step by step: importing new libraries

As you can see from the comments, the first line tells Scraperwiki that the scraper is written in Python. There are a couple of lines of comments crediting others whose work contributed to this scraper.

There are then a series of lines *importing* the libraries we need for this scraper. There are three notable ones here: `openpyxl`, `tempfile` and `pprint` are all libraries we haven't mentioned earlier in this book. `Openpyxl` we've already mentioned: it's for scraping xlsx files. `Tempfile` is another library for creating temporary files; and `pprint` is short for 'pretty printer' and helps with printing information. As always, search for the documentation to find out more.

4 new functions

Most of the rest of the code creates (using `def`) four new functions called

- `getworkbook`
- `getdata`
- `import_data`
- `grabexcellinks`

Right at the bottom of the scraper, after those functions have been defined, are the lines that start it running, and that's a good place to trace how this works:

The scraper uses a list of two pages which link to those expenses spreadsheets. The list is given a name of expenseslinks and created with this line:

```
expenseslinks = ['http://www.parliament.uk/business/
lords/whos-in-the-house-of-lords/house-of-lords-expenses/',
'http://www.parliament.uk/mps-lords-and-offices/members-
allowances/house-of-lords/holallowances/hol-expenses04/
201112/']
```

The next line then loops through that list. For each item it calls that expenseslink and runs the function 'grabexcellinks' on it:

```
for expenseslink in expenseslinks:

   #see 'def grabexcellinks' above to see
  what happens to that link

  grabexcellinks(expenseslink)
```

The short version of what happens next is this:

- grabexcellinks runs first.
- At the end of that function, import_data runs
- At the *start* of that import_data function, *another* function - getworkbook - runs, which *returns* some data back to the function import_data before it resumes...
- In the *middle* of import_data, the fourth function getdata runs, which *returns* some data back to the function import_data before it resumes and, finally, ends, saving the data collected along the way.

We'll break apart each function in turn.

Function 1: grabbing only xlsx links

The function `grabexcellinks` is relatively straightforward: it does what we've done in previous scrapers: scrape the contents of a URL using `scraperwiki.scrape`, then grab the contents of particular HTML tags (in this case `<div ... class="rte"><li><a>` - which are the tags that always contain the links to the xlsx files).

There's one thing new here: we're using some code to specify that we want links where the final characters are "xlsx".

```
links = root.cssselect('div.rte li a[href$="xlsx"]')
```

In previous scrapers we might have used a line like this to grab the whole link...

```
links = root.cssselect('div.rte li a')
```

...and then used an `if` test to only scrape those than ended in "xlsx".

In *this* scraper we've saved ourselves that test by adding an extra requirement:

`[href$="xlsx"]`

The key character is the `$` sign - you may remember from the **regex** chapter that this means 'select from the end'. As a whole this says 'where the value of the href= attribute is, *from the end*, "xlsx"'.

So, it is only grabbing the contents of the HTML within the tags `<div ... class="rte"><li><a>` *and* where the

last four characters of the `href` attribute of those contents are 'xlsx'.

For more on selectors like this see the W3C documentation on selectors[11].

The results of that line are put in a list called `links`. A loop takes each of these, prints the text of the link, grabs the `href=` attribute and puts it in a new variable called `linkurl`, prints the final four characters - `[-4:]` - of that, and then runs the new function `import_data` on it.

Because these links are *relative* - not complete - we also have to add the rest of the URL when we do that: hence that function actually **passed** the value:

```
'http://www.parliament.uk'+linkurl
```
Now we move to that `import_data` function.

Functions 2 and 3: Grab the workbook from the spreadsheet

The `import_data` function not only gets the data - it uses a further two newly defined functions to do so. In fact, it uses one of them in the first line. Here's where that function is defined, followed by the line that calls a further function:

```
def import_data(linkurl):

    wb = getworkbook(linkurl)
```

The clue that this is our own function rather than one from a library is the fact that this function is not attached to

---

anything with a period, like for example `scraperwiki.scrape` or `lxml.html.fromstring`.

To trace what happens to `linkurl`, then, we need to find `def getworkbook`. Here's the code for that, which names `linkurl` (or any **parameter passed** to it) as `url`:

```
def getworkbook(url):

   #use 'get' from the 'requests' library
  with the url and '.content' - put into
  'raw'.

   raw = requests.get(url, verify=False).content
```

This first part uses the `requests` library to grab (`.get`) the contents of the webpage at that URL, and put the results into a new variable `raw`. As always, you can search for [documentation on the library](http://docs.python-requests.org/en/latest/user/quickstart/#make-a-request)[12] and read more about its methods and functions. In this case I've also included a link in the code comments so it can be easily found later if I need to understand my own (or forked) code.

But that isn't the only new library in this function:

```
   #uses the 'tempfile' library and 'NamedTemporaryFile'
  to store temporary information in a new
  variable 'f'

   f = tempfile.NamedTemporaryFile('wb')
```

---

[12]http://docs.python-requests.org/en/latest/user/quickstart/#make-a-request

As you might guess, `tempfile` is a library for creating temporary files. Again, you can find more in the [documentation for `tempfile`](http://docs.python.org/2/library/tempfile.html)[13].

In the following lines we begin to fill that temporary file. Because `f` is a `tempfile` *object* the methods used on it (attached by a period) are going to be `tempfile` methods:

```
#uses '.write' to add 'raw' to that

f.write(raw)

#uses '.seek' to start from '0' - see
http://www.tutorialspoint.com/python/file_-
seek.htm

f.seek(0)

#now use the library openpyxl's .load_-
workbook function to grab f.name and put
into 'wb'

wb = openpyxl.load_workbook(f.name)

#close the temporary file 'f'

f.close()

#return 'wb' to whatever called this
function (now go back to the function
def import_data below)
```

---

[13]http://docs.python.org/2/library/tempfile.html

```
    return wb
```

Because this function is **called** within another function, it really needs to **return** something: in this case, wb. And now we return to the function that called this, too.

Now, note that the line which set off this process *also* names the results wb:

```
wb = getworkbook(linkurl)
```

That's important, because it could have named it anything. So although the last line of the getworkbook function returned wb it is returning the *value*, which is put into the *different* variable wb in this function (which happens to have the same name, for the coder's convenience but also, for a newbie, potential confusion).

This is the distinction between **local variables** and **global variables**. Within each of these functions, wb is a local variable that only exists as long as that function is running (this is why we need to **pass** or **return** it if we want to use it in another function).

With that value returned and placed into wb, then, we can use it in the rest of this function.

Functions 2 and 4: Grab the data from the spreadsheet

Still in our second function, we now do some things with the workbook we have grabbed, using the .get_sheet_-

names function to do just that, and eventually running our last new function on the first sheet:

```
 #use .get_sheet_names on 'wb' and put
results into 'sheet_names'

 sheet_names = wb.get_sheet_names()

 #if 'sheet_names' does not exist (i.e.
there are no sheets to put into it)

 if not sheet_names:

      #return (nothing) to whatever
     called this whole function

      return

 #otherwise, assume there's only 1 sheet,
and get first one (item 0 in 'sheet_-
names')

 rows = getdata(wb, sheet_names[0])
```

At this point we need to find getdata, which is the fourth function created within this code rather than taken from any of the libraries.

That function is going to take two parameters: wb and the first item ([0]) in the list sheet_names. And it is defined higher in the scraper with the following line:

```
def getdata(wb, sheet='Sheet1'):
```

Note that the code puts that first sheet into the variable
sheet. Next, it converts the contents into a type of object
it can work with: an array. To do this it uses another
method, .get_sheet_by_name - again, we can check the
documentation on openpyxl to find out more about this[14].

```
 #Turn a sheet on a workbook into an
array

data = wb.get_sheet_by_name(sheet)

#create an empty list 'stripped_table'

stripped_table = []

 #loop through each '.rows' item in
'data', call it 'row'

for row in data.rows:

    #create an empty list 'stripped_-
    row'

    stripped_row = []

    #loop through each item in
    'row', calling it 'cell'

    for cell in row:
```

[14]http://pythonhosted.org/openpyxl/_modules/openpyxl/workbook.html#
Workbook.get_sheet_by_name

```
           #grab the .value of
        'cell', use 'unicode'
        and '.strip', put into
        'stripped_cell'

         stripped_cell = unicode(cell.value).strip()

         #append the 'stripped_-
        cell'  to  'stripped_-
        row'

         stripped_row.append(stripped_-
        cell)

     #after that loop, append 'stripped_-
    row' to 'stripped_table'

     stripped_table.append(stripped_-
    row)

   #return  'stripped_table'  to  whatever
  called this whole function

  return stripped_table
```

All of this is basically going through each cell, in each row, cleaning it up and putting the results into a variable called stripped_table. The value of that is then returned to the part that called it, which I'll repeat below:

```
rows = getdata(wb, sheet_names[0])
```

Note that this time the value of stripped_table is actually put into a variable in our second function called rows. Next we look at what is done with that.

Function 2 resumes to completion

Another new library, pprint, is used to 'pretty print' (with a function also called pprint) the contents of rows. To find out more about pprint see [the documentation](http://docs.python.org/2/library/pprint.html)[15].

And we create an empty dictionary which will be used to store our data as it comes in.

```
pprint.pprint(rows)

#create empty dictionary variable 'expenses'

expenses = {}
```

Now, our spreadsheets contain data in 18 columns. We could save the data in each one separately, which would require 18 lines of code to name each field (column heading). But it will be simpler to grab those headings from the spreadsheet itself.

It will make sure our data is stored against the correct heading. This is because columns might be in different positions in different spreadsheets: if you decided that column 1 is always 'name' and in one spreadsheet column 1 actually contained addresses, that would still be stored in the 'name' field.

The following lines address that by grabbing headings from the spreadsheet and using them as field names for the same columns.

---

[15]http://docs.python.org/2/library/pprint.html

Because those headings are sometimes in row six, and sometimes in row seven, it also conducts a test to make sure it's grabbing the right row.

```
#create empty list variable 'headings'

headings = []

#loop through range 0-18 and assign that
number to variable 'num', then for each:

for num in range(0,18):

    #The spreadsheets either have
    headings on row 6 or 7

    #so these lines test which
    one contains 'Name' in the
    first [0] item (cell), and put
    results into 'target_row'

    if rows[5][0] == "Name":

        target_row = rows[5]

    elif rows[6][0] == "Name":

        target_row = rows[6]

    else:
```

```
    print "NEITHER ROW?",
   linkurl

    continue

 #Now print each item (cell)
for that row - note we're still
looping from 0-18 (each cell)

 print "rows[5/6][num]", target_-
row[num]
```


Many of these lines were not written first-time-round, but after earlier versions of the scraper were run, and hit problems. So, the lines above solve the problem of headings on different rows.

The next line solves another problem: characters in the headings which caused the scraper to fall over. These include slashes, brackets and full stops, and they are replaced using the `.replace` method - many, many times:

```
cleanedheading = target_row[num].replace("/","_-
").replace("No.","Number").replace("(","_-
").replace(")","_").replace("&","and")
```

This warrants some more detailed explanation.

Firstly, remember that this line is within the `for` loop that says `for num in range(0,18):` so `target_row[num]` is

`target_row[0]` - the first item in `target_row` - the first time this loop runs, and so on. `target_row` is the row containing or headings.

When this line runs it takes that item and uses `.replace` to replace any instances of a forward slash (`"/"`) with an underscore (`"_"`). The results are then used as the basis of another use of the `.replace` method to replace 'No.' with 'Number', and uses it again, and again, and again - to replace opening and closing brackets, and finally any ampersands.

Of course if a particular heading doesn't *have* a forward slash, or ampersand, or the word 'No.', then that particular `.replace` changes nothing, and for many headings that will be the case. We could of course only run this on particular items but that would complicate our code, take longer, and potentially miss problematic headings we hadn't anticipated.

Once it has completed all of those replacements, it puts the result into the variable `cleanedheading`. Then it prints that - and uses `.append` to add it to the previously empty list `headings`

```
print "cleaned heading:", cleanedheading

headings.append(cleanedheading)

print "headings:", headings
```

Once the `for loop` finishes looping through its range of numbers, grabbing all headings, we move onto the rows containing the data proper.

There are around 805 rows in each sheet, beginning at row 8. So this loop needs to cover the range 8 to row 805. For our own convenience, we print the name in each row - the first item - while testing. But we could easily delete this once we're happy it works.

```
for i in range(7,805):

    print rows[i][0]

    expenses['linkurl'] = linkurl
```

Now we begin saving the data into our expenses dictionary. The line above saves the first piece of data: the URL of the spreadsheet, under the field 'linkurl'.

The subsequent lines, however, save us a lot of time naming the other 18 fields. Once again it involves a for loop going through 18 numbers. This time it uses those numbers as indexes to both the heading and the item in the row.

```
for num in range(0,18):

    expenses[headings[num]]
  = rows[i][num]

 #the line below must use the
heading 'Name' with a capital
'N', as in the sheets - overlooked
originally

 scraperwiki.sqlite.save(['Name',
'linkurl'], expenses, 'expenses')
```

Remember that this whole `for` loop *also* runs within a `for` loop which is running through the range 7-804. So, the first time this loop runs the variable `i` is 7 and the variable `num` is 0. Put another way, this line:

```
expenses[headings[num]] = rows[i][num]
```

Means the following:

```
expenses[headings[0]] = rows[7][0]
```

Or, reading right-to-left: grab the *first* item within the *eigth* list in `rows`, and put it into `expenses`, in a field given the name of the *first* item in the list `headings`.

And that's only the first time this loop runs. The 0 indexes then become 1, 2, 3, up to 17 (remember the `range` function runs up to but does not include the last number), at which point *this* `for` loop finishes its first run, and the row is saved using `scraperwiki.sqlite.save`. We use two unique keys - 'Name' and 'linkurl' - because names will appear on each sheet, but each sheet will have its own URL, so the combination of both should be unique.

The capital 'N' on name is important, too: because we are grabbing heading fields from the spreadsheets we must name it correctly here - even using a small 'n' would get it wrong, and cause the scraper to generate an error.

Once that row is saved, the `for` loop *containing* the other begins its second loop: the 7 index becomes 8, and our more recent `for` loop starts again from 0.

In the process the scraper goes through each cell, in each row, and stores its value against a field that correlates with the headings with the column it sits in. Once it has finished that, it returns to the first function and resumes

all over again with the spreadsheet at the next link in the list, and repeats until the list ends.

Recap

- Openpyxl is a useful library for scraping XLSX files - i.e. spreadsheets generated by the most recent versions of Excel.
- tempfile is a library for creating temporary files, which can useful while you scrape.
- pprint is another new library for 'pretty printing' objects.
- The $ sign can be used to select items where the last few characters meet a particular pattern (in this case 'xlsx'). See the W3C documentation on selectors[16] for more.
- You can store the values of a particular spreadsheet row in a list variable and then use those as your field names. Use .replace to clean out any problematic characters.
- However, make sure that your **key** matches the value of one or more cells *exactly*, including upper or lower case.

Tests

- Change the names of the new functions defined in the scraper so they adhere to the Python convention

---

[16]http://www.w3.org/TR/css3-selectors/#attribute-substrings

of using underscores, e.g. change `def getdata` to `def get_data`.

- See if you can change the line that grabs links ending in 'xlsx' (`links = root.cssselect('div.rte li a[href$="xlsx"]')`), so that it uses an `if` test instead.
- This scraper assumes the spreadsheet only has one sheet. Can you amend it so it grabs a different sheet, or even more than one sheet?
- You can simplify these lines...

```
stripped_table = []

for row in data.rows:

    stripped_row = []

    for cell in row:

        stripped_cell = unicode(cell.value).strip()

        stripped_row.append(stripped_-
        cell)

    stripped_table.append(stripped_-
    row)

return stripped_table
```

...into one single line:

```
return [[unicode(cell.value).strip() for cell
in row] for row in data.rows]
```

But can you work out what it's doing? (*Thanks to Zarino Zappia for this suggestion*)

35 Acknowledgements

Happy birthday to Thomas Seymat (who bought this for himself as his present).

This list will grow as the book does.

36 List of websites scraped

The following is a list of the websites scraped so far in this book:

Scraper #1: List of prisons (Wikipedia)[1]

Scraper #2: XML file of UK councils[2]

Scraper #3 and #4: Journalism job ads[3]

Scraper #5: Durham Mining Museum[4]

Scraper #6: Free school meals data in Scottish schools, example page[5]

Scraper #7: Job ads from the organisation G4S[6]

Scraper #8: NHS pathfinder consortia (CCGs)[7]

Scraper #9: List of Labour Party leader's meetings and dinners with donors and trade union general secretaries[8]

Scraper #10: Office of the Independent Adjudicator's

---

[1] http://en.wikipedia.org/wiki/List_of_prisons

[2] http://openlylocal.com/councils.xml

[3] http://www.gorkanajobs.co.uk/jobs/journalist/

[4] http://www.dmm.org.uk/mindex.htm

[5] http://www.ltscotland.org.uk/scottishschoolsonline/schools/freemealentitlement.asp?iSchoolID=5237521

[6] http://careers.g4s.com/search-job.php?showall=y&showDistance=N&location_range=&SortBy=3&pageNum=14

[7] http://healthandcare.dh.gov.uk/context/consortia/

[8] http://www.labour.org.uk/list-of-meetings-with-donors-and-general-secretaries,2012-03-30

page of Older Case Studies[9]

Scraper #11: Inclusive London's guide to accessibility at Olympic venues[10].

Scraper #12: Twitter[11]

Scraper #13: Scraperwiki's 'Hello World' page[12]

Scraper #14: Horse Race Death Watch[13]

Scraper #15: Top-selling albums of all time page 1[14]

Scraper #16: Horse Race Death Watch[15]

Scraper #17 and #18: Free school meals data in Scottish schools, example page[16]

Scraper #19: Brent[17] CCG, Ealing[18] CCG, and five others.

Scraper #20: Lottery grants[19]

Scraper #21: Independent Schools Inspectorate reports[20]

Scraper #22: National Assembly Official Report Thursday 8th July 2010[21]

---

[9]http://www.oiahe.org.uk/decisions-and-publications/recent-decisions-of-the-oia/older-case-studies.aspx

[10]http://www.inclusivelondon.com/search/what/0/37523/921/0/London/Olympic%20And%20Paralympic%20Venues/0/0/0/0/0/search.aspx?tbclr=1

[11]http://search.twitter.com

[12]https://scraperwiki.com/hello_world.html

[13]http://www.horsedeathwatch.com/

[14]http://www.madingley.org/uploaded/example_table_1.html

[15]http://www.horsedeathwatch.com/

[16]http://www.ltscotland.org.uk/scottishschoolsonline/schools/freemealentitlement.asp?iSchoolID=5237521

[17]http://www.brentccg.nhs.uk/

[18]http://www.ealingccg.nhs.uk

[19]http://www.lottery.culture.gov.uk/AdvancedSearch.aspx

[20]http://www.isi.net/reports/

[21]http://www.madingley.org/uploaded/Hansard_08.07.2010.pdf

Scraper #23: Speed camera locations, week starting 03/09/2012[22]

Scraper #24: London Metropolitan Police publication scheme page on knife crime[23]

Scraper 25: Dorset Police force's use of stop and search powers[24]

Scraper 26: Birmingham City Council spending over £500[25].

Scraper 27: HMRC income tax rates - broken link[26]

Scraper 28: Winter SITREPS - situation reports from the Department of Health [27]

Scraper 29: National Committee for Quality Assurance Recognition Directory clinician search results[28]

---

[22]http://www.staffssaferroads.co.uk/media/114997/03092012_forwebsite.pdf

[23]http://www.met.police.uk/foi/c_priorities_and_how_we_are_doing.htm

[24]http://www.dpa.police.uk/default.aspx?page=473

[25]http://www.birmingham.gov.uk/payment-data

[26]http://www.hmrc.gov.uk/stats/tax_structure/incometaxrates_1974to1990.xls

[27]http://transparency.dh.gov.uk/2012/10/26/winter-pressures-daily-situation-reports-2012-13/

[28]http://recognition.ncqa.org/PSearchResults.aspx?state=NY&rp=

37 Glossary

This will be updated as new chapters are added

API   Application Programming Interface. A technology that allows you to ask questions of data (every time you use a Twitter, Facebook or Google Maps app you are using their API to ask questions of their data), and that allows some services to talk to each other.

Append

to add an item to a list variable. In this book it's done with the `.append` method like so: `listname.append(thingtobeadded)`

Argument

the term in Python for the ingredient(s) used by a function. For example, in the line `functionname(thing)`, the *argument* is `thing`. An argument is called a *parameter* when the function is defined, e.g. `def functionname(thing):`.

aspx pages

a webpage typically generated by a search where the results are not contained at a specific URL but rather temporarily held by the browser. This is problematic for scrapers which may have to mimic a browser in order to temporarily 'store' this information in a similar way, for example by using the *mechanize* library.

Attributes

an element of HTML which can be helpful in specifying the particular data we want to scrape, e.g. `<div class="jobInfo">` has the *tag* `<div>` but also the *attribute* class, which has the *value* of jobInfo.

Call, **or calling**

calling is the process of running a function. For example if you have a function named `scrapethepage` this will be defined somewhere in your code with `def scrapethepage():` but it will also be called somewhere with, simply: `scrapethepage()`. Often you will also *pass* an *argument* when you *call* a function, e.g. the line `scrapethepage(url)` *calls* `scrapethepage` and passes it the *argument* `url`.

Class

A class is a particular type of object. These are normally created to make it easier to create and find out their properties. For example, instead of having to create a series of variables for different cars, and individually setting their colour, registration, and so on, you could create a `car` class (a "blueprint"[1], if you like) for all cars, and give it those properties, along with easy ways of accessing those. Having a class then saves you time both in creating any objects in that class, as well as in finding out a property of any object. The `xlrd` library, for example, has a class for `sheet` (a sheet in a spreadsheet). That has properties like the number of rows and columns, for

example, which can be easily accessed with *methods* like `nrows` and `ncols`.

Cloning

the process of copying a scraper (or other piece of code) into a new version that you can add to. Also called *forking*.

Cookies

for the purpose of scraping, cookies are small scripts that store information about your search *locally*, on your computer, which means that the search results do not have their own URL that can be shared. This can cause problems for scraping, which can be addressed with the **mechanize** library.

Dictionary

an object in programming which can store more than one *pair of items*. In Python dictionaries are indicated by curly brackets, and each pair is separated by a colon, like so: {"Father" : 'Paul', "Brother" : 'John', "Mother" : 'Mary', "Son" : 'Ringo'}.

Documentation

reference material on a particular function or other piece of code. These can vary in quality and amount of jargon. For instance, the Help files for Google Docs functions are basically documentation.

Field-value pair

part of a **query string** that specifies both a *field*

being specified, and the value of that field requested. For example in the field-value pair name=bradshaw the field is 'name' and the value 'bradshaw'. Some databases allow for a series of pairs with empty values, separated by ampersands like so: `name=&title=&year=2012`

Float or floating point

a number with a decimal point such as '10.1', rather than a whole number, e.g. '10' (known as an *integer*).

Index

the position of something, e.g. first, second, third. Often indicated by a number in square brackets, e.g. [1]. Most programming uses a zero-based index, which means it starts counting at 0. Confusingly for those less logical among us, that means that the second item is indicated by the index 1, and the third by 2, and so on. You'll get used to it. Some tools designed to be used beyond the programming community, such as Google Docs, use a one-based index, where counting begins at 1.

Forking

the process of copying, or *cloning*, a piece of code in order to make changes to it. Quite often this is done from a basic template which can be forked into multiple versions, e.g. various 'forks' of a Twitter scraper which scrape for tweets mentioning different terms.

For loop

> a loop which repeats for every item in a list, e.g. for each item in that list the loop might grab the contents, or its attributes, or both, or perform another action.

Forking

> the process of copying a scraper (or other piece of code) into a new version that you can add to. Also called *cloning*. One scraper may have a number of 'forks' as different users add different sections to it.

Formula

> in spreadsheets, a calculation, function, or combination of the two that generates data. A formula starts with the = sign.

Function

> a piece of programming that does something. For example, the function importHTML imports from a HTML webpage. Functions are normally followed by parentheses containing the ingredients that the function needs to work
>
> its *parameters*.

Global variable

> a *variable* which can be accessed by any function in your code (unlike a *local variable*).

HTML

> Hypertext Markup Language, the language used in

most webpages. This has a structure which is useful if you're scraping.

Integer

a whole number, e.g. '10', rather than a number with a decimal point such as '10.1' (known as a *float* or *floating point*).

Key A column in your data which is used to organise it. This should be something that is unique for every record, such as an ID number or code (you may have to invent one) You should also avoid using values that may have empty entries, as this can trip up the scraper.

LIFO Rule

stands for Last In First Out. In other words, in HTML code, the last tag used should be the first one closed: if a `<div>` tag is used and then `<p>` to open a paragraph, then the last should be closed with `</p>` before the first is closed with `</div>`

List an object in programming which can store more than one item. In Python lists are indicated by square brackets like so: ['Paul', 'John', 'Mary', 'Ringo']. You can retrieve an item from a list by using an index like so: listname[0].

Local variable

a *variable* which can only be accessed within *one* function in your code, but does not exist outside of it (unlike a *global variable*).

Mechanize

a **library** used for scraping databases that require forms to be filled in. It also tackles the problem of storing **cookies**, and generally anything where pretending to be a **browser** comes in useful.

Method

A method is a type of *function* that is used on objects of a particular *class*. For example, in the spreadsheet scraping library `xlrd` there are a number of methods which can be used on the `sheet` class, such as `nrows` and `ncols`. There are other methods for the `cell` class, and so on.

Object

Broadly, anything being manipulated by your code (Python is an *Object Oriented Language*, which means it works by creating types of objects and manipulating them). Specifically, however, it is normally used with the name of a library, such as an "lxml object" or "xlrd object", to refer to something created using that library, and which as a result is best manipulated by *functions* and *methods* from that library, or converted into a different type of object if you want to use it with another library. The function `lxml.html.fromstring` for example, converts a page scraped with `scraperwiki.scrape` into an lxml.html object.

Parameters

the ingredients used by a *function* to work. For

example, the function importHTML needs three parameters: a URL; a *query* indicating what on that page you want scraped; and an *index* for which one of those you want. When these parameters are supplied (typically when a function is *called*), they are called *arguments*.

Pass, or passing

when *calling* a function you will often also *pass* it an *argument* - the ingredient(s) it needs to work with. For instance, the line `scrapethepage(url)` *calls* `scrapethepage` and passes it the *argument* `url`. When the argument is passed to the function it will be given a new name inside that function (a *local variable*). This may be the same as the name it previously had, but it can be any name.

Predicates

in *Xpath*, a way of indicating what specific piece of data you're looking for, for example by its position, by the characters it begins with or contains, a value of a particular property it has, or something else. Predicates are always contained within square brackets[2].

Query

instructions on what you're looking for. E.g. in the formula =importHTML("URL", "table", 1) the query is "table".

---

[1]http://www.sthurlow.com/python/lesson08/

Regex - regular expressions: a way of specifying structure that recurs within a webpage or a series of pages. For example, it might specify that you do something (such as grab data) 'where there are any 3 digits followed by 4 lowercase characters' or 'where the word Date: occurs'. Regex uses a series of codes that can look hugely intimidating to the beginner, but they're just that: codes. If you can crack the code, you can crack Regex. There's lots of documentation on Regex online, as well as tools to help you write or decode it if you prefer.

Query string

in a URL, the part following a question mark which specifies the search criteria, often using one or more **field-value pairs**. For example in http://www.eib.org/projects/loans/l union&country=united+kingdom&sector=[3] the query string is start=2007&end=2012&region=european-union&country=un

Range

a range of numbers between a start and end number, stored as a list. This is generally generated by the range function, e.g. range(1,10) would generate a list of numbers from 1 to 9. The range stops just before the last number.

Regex

Short for *regular expression*, regex allows you to specify a string or pattern of characters (including invisible ones such as spaces) that you want to match

---

[2]http://www.w3schools.com/xpath/xpath_syntax.asp

against. This allows you to scrape information as long as it matches that pattern. For example, you could use regex to find telephone numbers in data by specifying 'a series of seven digits' (or however many you expect). You could use it to grab dates by saying 'one or more digits followed by a slash, followed by one or more digits, another slash, and four digits'. This is done with special characters, such as square brackets to specify a range of characters, and wildcards to match 'one or more' characters. See Scraper 9 in Chapter 10 for more.

Regular expressions

see *Regex*.

Return

The process of 'passing back' information to another part of the code. For example, one function in the code might *call* another function to *return* some information about an element. Typically that second function will eventually end with a line that uses `return` to give something back, which the first function can then work with, *after* the line where it called that second function.

Scraping

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.

String

      a string of characters, such as a name, a phrase, a URL, a code, etc. Strings are one of a number of types of data recognised by computers and treated differently. For example, you cannot perform a calculation with a string, but you can with numbers. Strings are normally indicated by straight quotation marks or inverted commas, e.g. "this is a string". If you want to treat numbers as strings, e.g. combine them rather than add them together, this is how you make the distinction. Numbers (typically called *integers* and *floats*) don't use quotation marks. Likewise, the quotation marks help distinguish strings from *variables*, which are names for other things.

Tag   a piece of HTML or XML, such as `<strong>` or `<div>`. Can be used to specify which part of a webpage you want to scrape.

Uncommenting

      comments in code are normally distinguished from working code by a sign of some sort. In Python comments are indicated by a hash sign at the start. 'Uncommenting' means removing that hash sign so that anything after it becomes active code, rather than inactive 'comment'.

Value

      in HTML, the property of a particular *tag*'s *attribute*, e.g. a border tag's attribute might have the value of 'red'. This can be helpful in specifying the particular

data we want to scrape, e.g. `<div class="jobInfo">` has the *tag* `<div>` but also the *attribute* class, which has the *value* of jobInfo.

Variable

pick whichever definition makes most sense to you. A variable is a) a 'box' containing other things, e.g. the age of a politician, or their voting record. Or b) a variable is a name for something else, e.g. the age of a politician, or their voting record. Variables are useful because they can be used over and over again in different situations, and because they can be changed depending on circumstances. For example, the variable BobsAge can start as 41, but go up by one every time today's date is his birthday. The variable BobsVotingRecord could contain a number of records, which are added to every time he votes. A variable is normally created (instantiated) with an equals sign, like so: BobsAge = 41. And it can be changed like so: BobsAge = BobsAge + 1.

XML

a structured language, often used to describe objects and data. Easy to convert into spreadsheet format with the importXML function in Google Docs.

Xpath

a language for finding information in an XML document, but which can also be used for documents in similar languages such as RSS and HTML.

While loop

> a loop which repeats as long as a particular condition is met, e.g. while a value is above or below a certain amount the loop might grab the contents, or its attributes, or both, or perform another action.