

# 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](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](#)<sup>1</sup> 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.

---

<sup>1</sup><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<sup>2</sup>](#), 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<sup>3</sup>](#).

<sup>2</sup><http://support.google.com/docs/bin/answer.py?hl=en&answer=155182>

<sup>3</sup><https://docs.google.com/spreadsheets/ccc?key=0ApTo6f5Yj1jJdDBSb0FPQm9jUjYzdcyNWIUjVYMFE>

## Tables and lists?

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

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

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

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

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

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

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

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

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

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

### **Scraping tip #1: Finding out about functions**

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

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

Likewise, as you explore more powerful scrapers such as those hosted on Scrapperwiki 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](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*<sup>1</sup>

## 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:

---

<sup>1</sup><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><sup>2</sup> (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/><sup>3</sup>)

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.

---

<sup>2</sup><http://openlylocal.com/councils.xml>

<sup>3</sup><http://openlylocal.com/councils/>

This XML file does not appear to have any style information associated with it. The document

```

▼<councils type="array">
  ▼<council>
    <address>Marischal College Broad Street Aberdeen, AB10 1AB</address>
    <annual-audit-letter nil="true"/>
    <authority-type>Unitary</authority-type>
    <cipfa-code>S0001</cipfa-code>
    <country>Scotland</country>
    <created-at type="datetime">2009-06-17T12:29:39+01:00</created-at>
    <data-source-name/>
    <data-source-url/>
    <defunkt type="boolean">false</defunkt>
    <egr-id type="integer">1</egr-id>
    <feed-url>http://www.aberdeencity.gov.uk/acnews.xml</feed-url>
    <gss-code/>
    <id type="integer">37</id>
    <lat type="float">57.1481</lat>
    <ldg-id type="integer" nil="true"/>
    <lng type="float">-2.096</lng>
    <member-count type="">45</member-count>
    <name>Aberdeen City Council</name>
    <mess-id/>
    <normalised-title>aberdeen</normalised-title>
    <notes/>
  ▼<ons-url>
    http://neighbourhood.statistics.gov.uk/dissemination/LeadAreaSearch.
    a=3&i=1&m=0&enc=1&areaSearchText=AB10+1AR+&areaSearchType=13&extende
  </ons-url>

```

## 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.

## 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

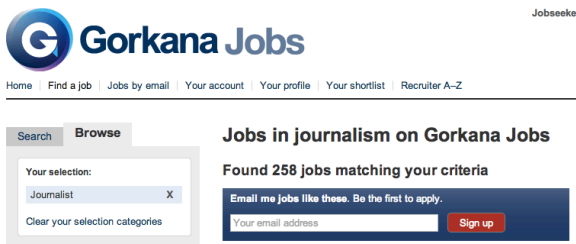
pulling data from an xml file this is where you might see what that file is. The link will probably be *relative* (i.e. it begins with / rather than `http://`). In this case add it to the site's *base* URL - for example if the flash page is `http://bobswebsite.com/flashmovie` and the xml file address 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)<sup>4</sup> 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.

---

<sup>4</sup><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](#)<sup>1</sup>. 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>
```

---

<sup>1</sup><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 `</a>` - which is an easy way to identify it.

Here’s similar code for an image:

```

```

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



`<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:

```

```

## 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/><sup>2</sup>, 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"
```

---

<sup>2</sup><http://www.gorkanajobs.co.uk/jobs/journalist/>



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](http://www.distilled.net/blog/distilled/guide-to-google-docs-importxml/)<sup>3</sup>.

---

<sup>3</sup><http://www.distilled.net/blog/distilled/guide-to-google-docs-importxml/>