CHRIS HANRETTY

SCRAPING THE WEB FOR ARTS AND HUMANITIES

UNIVERSITY OF EAST ANGLIA

Copyright © 2013 Chris Hanretty

PUBLISHED BY UNIVERSITY OF EAST ANGLIA

First printing, January 2013

Contents

1 *Introducing web scraping* 9

2 *Introducing HTML* 13

3 *Introducing Python* 19

4 *Extracting some text* 25

5 *Downloading files* 33

6 *Extracting links* 39

7 *Extracting tables* 43

8 *Final notes* 49

List of code snippets

1.1 BBC robots.txt 11

2.1 Minimal HTML page 14
2.2 Basic HTML table 15

3.1 Python terminal 20

4.1 Pitchfork.com review 25
4.2 Pitchfork.com code 27
4.3 Scraper to print full text 28
4.4 Scraper output 28
4.5 Looping over paragraphs 29
4.6 Looping over divisions 30
4.7 File output 31

5.1 Saving a file 34
5.2 Saving multiple files 35
5.3 Leveson inquiry website 36
5.4 Links on multiple pages 36

6.1 AHRC code 39
6.2 AHRC link scraper 40
6.3 AHRC output 41
6.4 OpenOffice dialog 42

7.1 ATP rankings 43
7.2 ATP code 44
7.3 ATP code 44
7.4 Improved ATP code 45
7.5 ATP output 46

Introduction

THERE'S LOTS OF INFORMATION on the Web. Although much of this information is very often extremely useful, it's very rarely in a form we can use directly. As a result, many people spend hours and hours copying and pasting text or numbers into Excel spreadsheets or Word documents. That's a really inefficient use of time – and some times the sheer volume of information makes this manual gathering of information impossible.

THERE IS A BETTER WAY. It's called scraping the web. It involves writing computer programs to do automatically what we do manually when we select, copy, and paste. It's more complicated than copying and pasting, because it requires you to understand the language the Web is written in, and a programming language. But web scraping is a very, *very* useful skill, and makes impossible things possible.

In this introduction, I'm going to talk about the purpose of the booklet, some pre-requisites that you'll need, and provide an outline of the things we'll cover along the way.

What is the purpose of this booklet? This booklet is designed to accompany a one day course in Scraping the Web. It will take you through the basics of HTML and Python, will show you some practical examples of scraping the web, and will give questions and exercises that you can use to practice. You should be able to use the booklet without attending the one-day course, but attendance at the course will help you get a better feel for scraping and some tricks that make it easier (as well as some problems that make it harder).

Who is this booklet targeted at? This booklet is targeted at postgraduate students in the arts and humanities. I'm going to assume that you're reasonably intelligent, and that you have a moderate to good level of computer literacy – and no more. When I say, 'a moderate to good level of computer literacy', I mean that you know where files on your computer are stored (so you won't be confused when I say something like, 'save this file on your desktop', or 'create a new directory within your user folder or home directory'), and that you're familiar with word processing and spreadsheet software (so you'll be familiar with operations like search and replace, or using

a formula in Excel), and you can navigate the web competently (much like any person born in the West in the last thirty years). If you know what I mean when I talk about importing a spreadsheet in CSV format, you're already ahead of the curve. If you've ever programmed before, you'll find this course very easy.

What do you need before you begin? You will need

- a computer of your own to experiment with
- a modern browser (Firefox, Chrome, IE 9+, Safari)[1]
- an internet connection (duh)
- an installed copy of the Python programming language, version two-point-something
- a plain-text editor[2].

[1] Firefox has many add-ons that help with scraping the web. You might find Firefox best for web scraping, even if you normally use a different browser.

[2] Notepad for Windows will do it, but it's pretty awful. Try Notetab (www.notetab.ch) instead. TextEdit on Mac will do it, but you need to remember to save as plain-text

We'll cover installing Python later, so don't worry if you weren't able to install Python yourself.

What does this booklet cover? By the time you reach the end of this booklet, you should be able to

- extract usable text from multiple web pages
- extract links and download multiple files
- extract usable tabular information from multiple web pages
- combine information from multiple web pages

There's little in this book that you couldn't do if you had enough time and the patience to do it by hand. Indeed, many of the examples would go quicker by hand. But they cover principles that can be extended quite easily to really tough projects.

Outline of the booklet In chapter 1, I discuss web scraping, some use cases, and some alternatives to programming a web scraper. The next two chapters give you an introduction to HTML, the language used to write web pages (Ch. 2) and Python, the language you'll use to write web scrapers (Ch. 3). After that, we begin gently by extracting some text from web pages (Ch. 4) and downloading some files (Ch. 5). We go on to extract links (Ch. 6) and tables (Ch. 7), before discussing some final issues in closing.

1
Introducing web scraping

WEB SCRAPING is the process of taking unstructured information from Web pages and turning it in to structured information that can be used in a subsequent stage of analysis. Some people also talk about screen scraping, and more generally about *data wrangling* or *data munging*.[1]

[1] No, I don't know where these terms come from.

Because this process is so generic, it's hard to say when the first web scraper was written. Search engines use a specialized type of web scraper, called a web crawler (or a web spider, or a search bot), to go through web pages and identify which sites they link to and what words they use. That means that the first web scrapers were around in the early nineties.

Not everything on the internet is on the web, and not everything that can be scraped via the web should be. Let's take two examples: e-mail, and Twitter. Many people now access their e-mail through a web browser. It would be possible, in principle, to scrape your e-mail by writing a web scraper to do so. But this would be a really inefficient use of resources. Your scraper would spend a lot of time getting information it didn't need (the size of each icon on the display, formatting information), and it would be much easier to do this in the same way that your e-mail client does.[2]

[2] Specifically, by sending a request using the IMAP protocol. See Wikipedia if you want the gory details: http://en.wikipedia.org/wiki/Internet_Message_Access_Protocol

A similar story could be told about Twitter. Lots of people use Twitter over the web. Indeed, all Twitter traffic passes over the same *protocol* that web traffic passes over, HyperText Transfer Protocol, or HTTP. But when your mobile phone or tablet Twitter app sends a tweet, it doesn't go to a little miniature of the Twitter web site, post a tweet, and wait for the response – it uses something called an API, or an Application Protocol Interface. If you really wanted to scrape Twitter, you should use their API to do that.[3]

[3] If you're interested in this kind of thing, I can provide some Python source code I've used to do this.

So if you're looking to analyze e-mail, or Twitter, then the stuff you learn here won't help you directly. (Of course, it might help you indirectly). The stuff you learn here will, however, help you with more mundane stuff like getting text, links, files and tables out of a set of web pages.

Usage scenarios

Web scraping will help you in any situation where you find yourself copying and pasting information from your web browser. Here are some times when I've used web scraping:

- to download demo MP3s from `pitchfork.com`;
- to download legal decisions from courts around the world;
- to download results for Italian general elections, with breakdowns for each of the 8000+ Italian municipalities
- to download information from the Economic and Social Research Council about the amount of money going to each UK university

Some of these were more difficult than others. One of these – downloading MP3s from Pitchfork – we'll replicate in this booklet.

Alternatives

I scrape the web because I want to save time when I have to collect a lot of information. But there's no sense writing a program to scrape the web when you could save time some other way. Here are some alternatives to screen scraping:

ScraperWiki ScraperWiki (`https://scraperwiki.com/`) is a website set up in 2009 by a bunch of clever people previously involved in the very useful `http://theyworkforyou.com/`. ScraperWiki *hosts* programmes to scrape the web – and it also hosts the nice, tidy data these scrapers produce. Right now, there are around 10,000 scrapers hosted on the website.

You might be lucky, and find that someone has already written a program to scrape the website you're interested in. If you're not lucky, you can pay someone else to write that scraper – as long as you're happy for the data and the scraper to be in the public domain.

Outwit Outwit (`http://www.outwit.com/`) is freemium[4] software that acts either as an add-on for Firefox, or as a stand-alone product. It allows fairly intelligent automated extraction of tables, lists and links, and makes it easier to write certain kinds of scraper. The free version has fairly serious limitations on data extraction (maximum 100 rows in a table), and the full version retails for 50.

[4] Free with a paid-for upgrade to unlock professional features.

SocSciBot If you are just interested in patterns of links between web sites and use Windows, you might try SocSciBot (`http://socscibot.wlv.ac.uk/`) from the University of Wolverhampton. It's free to use, and has good integration with other tools to analyze networks.

Google Refine Google Refine is a very interesting tool from Google that is most often used to clean existing data. However, it's pretty good at pulling information in from tables and lists on web pages, including Wiki pages, and has some excellent tutorials. https://www.youtube.com/watch?v=cO8NVCs_Ba0 is a good start.

Asking people! If the information comes from a page run by an organisation or an individual, you can always try asking them if they have the original data in a more accessible format. A quick (and polite) email could save you a lot of work.

Is this ethical? Is it legal?

There are a number of ethical and legal issues to bear in mind when scraping. These can be summarized as follows:

Respect the hosting site's wishes Many large web sites have a file called `robots.txt`. It's a list of instructions to bots, or scrapers. It tells you wish parts of the site you shouldn't include when scraping. Here is (part of) the listing from the BBC's `robots.txt`:

Listing 1.1: BBC robots.txt

Listing 1.1: BBC robots.txt

```
User-agent: *
Disallow: /cgi-bin
Disallow: /cgi-perl
Disallow: /cgi-perlx
Disallow: /cgi-store
Disallow: /iplayer/cy/
Disallow: /iplayer/gd/
Disallow: /iplayer/bigscreen/
Disallow: /iplayer/cbeebies/episodes/
Disallow: /iplayer/cbbc/episodes/
Disallow: /iplayer/_proxy_
Disallow: /iplayer/pagecomponents/
Disallow: /iplayer/usercomponents/
Disallow: /iplayer/playlist/
Disallow: /furniture
Disallow: /navigation
Disallow: /weather/broadband/
Disallow: /education/bitesize
```

That means that the BBC doesn't want you scraping anything from iPlayer, or from their Bitesize GCSE revision micro-site. So don't.

Respect the hosting site's bandwidth It costs money to host a web site, and repeated scraping from a web site, if it is very intensive, can result in the site going down.[5] It's good manners to write your program in a way that doesn't hammer the web site you're scraping. We'll discuss this later.

[5] This is behind many denial-of-service attacks: http://en.wikipedia.org/wiki/Denial-of-service_attack

Respect the law Just because content is online doesn't mean it's yours to use as you see fit. Many sites which use paywalls will require you to sign up to Terms and Agreements. This means, for example, that you can't write a web scraper to download all the articles from your favourite journal for all time.[6] The legal requirements will differ from site to site. If in doubt, consult a lawyer.[7]

[6] In fact, this could land your university in real trouble – probably more than torrenting films or music.

[7] In fact, if in doubt, it's probably a sign you shouldn't scrape.

2
Introducing HTML

IN ORDER TO SCRAPE THE WEB, we need to use understand the language used to write web pages. Web pages are written in HTML – HyperText Markup Language.

HTML was invented[1] by British physicist Tim Berners-Lee in 1991. HTML has gone through different versions as new features have been added. The current version of HTML is HTML5. HTML's custodian is the World Wide Web Consortium or *W3C*. They publish the full technical specification of HTML. You can find the HTML5 specification online[2] – but it's a very boring read.

[1] Berners-Lee was inspired by another markup language called SGML, but that's not important.

[2] `http://dev.w3.org/html5/spec/single-page.html`

In this section, we'll go through the basics of HTML. Although we won't be producing any HTML, we'll need to understand the structure of HTML, because we'll be using these tags as signposts on our way to extracting the data we're interested in. Towards the end, we'll save a basic HTML file to the hard drive just to check that we can write plain-text files.

Basics

The basics of HTML are simple. HTML is composed of elements called *tags*. Tags are enclosed in left and right-pointing angle brackets. So, `<html>` is a tag.

Some tags are paired, and have opening and closing tags. So, `<html>` is an opening tag which appears at the beginning of each HTML document, and `</html>` is the closing tag which appears at the end of each HTML document. Other tags are unpaired. So, the `<img>` tag, used to insert an image, has no corresponding closing tag.

Listing 2.1 shows a basic and complete HTML page.

You can see this HTML code – and the web page it produces – at the W3C's TryIt editor.[3] Complete Exercise 1 before you go on.

[3] `http://www.w3schools.com/html/tryit.asp?filename=tryhtml_intro`

Now you've experimented with some basic HTML, we can go over the listing in 2.1 in more detail:

- Line 1 has a special tag to tell the browser that this is HTML, and not plain text.
- Line 2 begins the web page properly

Listing 2.1: Minimal HTML page

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

Exercise 1 HTML basics

Go to http://www.w3schools.com/html/tryit.asp?filename=tryhtml_intro. Make the following changes. Remember to click 'Submit Code' after you do so.

1. Try entering some text after 'my first paragraph'.
2. What happens if you delete the closing H1 tag?
3. What happens if you change the `h1` to `h2`? And `h6`? And `h7`?
4. Try making 'paragraph' bold by sticking opening and closing `<b>` tags. Does it work?
5. If you can make text **b**old with `<b>`, how might you *i*talicise text? Try it!

- Line 3 starts the body of the web page. Web pages have a `<body>` and a `<head>`. The `<head>` contains information like the title of the page.
- Line 5 starts a new heading, the biggest heading size, and closes it
- Line 7 starts a new paragraph, and closes it. HTML needs to be told when a new paragraph starts. Otherwise, it runs all the text together.
- Lines 9 and 10 close off the tags we began with.

Links and images

The minimal example shown in 2.1 is extremely boring. It lacks two basic ingredients of most modern web pages – images, and links.

The tags to insert an image and to create a link respectively are both tags which feature *attributes*. Here's an example of a link to the UEA homepage.

```
<a href="http://www.uea.ac.uk/">University of East Anglia</
    a>
```

and here's a code snippet which would insert the UEA logo.

```
<img src="http://www.uea.ac.uk/polopoly_fs/1.166636!
    ueastandardrgb.png">
```

The *attribute* for the link tag is `href` (short for hyper-reference), and it takes a particular value – in this case, the address of the web page we're linking to. The attribute for the image tag is `src` (short for source), and it too takes a particular value – the address of a PNG image file.[4] Try copying these in to the TryIt editor and see what happens. You can see that the tag for links has an opening and closing tag, and that the text of the link goes between these two. The image tag doesn't need a closing tag.

[4] PNG stands for Portable Network Graphics. It's a popular image format – not as popular as `.gif` or `.jpeg`, but technically superior.

Tables

Because of our interest in scraping, it's helpful to take a close look at the HTML tags used to create tables. The essentials can be summarized very quickly: each table starts with a `<table>` tag; Each table row starts with a `<tr>` tag; Each table cell starts with a `<td>` tag.[5]

[5] Strictly, `<td>` is a mnemonic for 'table data'.

Listing 2.2 shows the code for a basic HTML table.

Listing 2.2: Basic HTML table

```
<table>
<tr>
  <td>Country</td>
  <td>Capital</td>
</tr>
<tr>
  <td>Australia</td>
  <td>Canberra</td>
</tr>
<tr>
  <td>Brazil</td>
  <td>Brasilia</td>
</tr>
<tr>
  <td>Chile</td>
  <td>Santiago</td>
</tr>
</table>
```

Complete Exercise 2 before you continue.

Common HTML tags

You can find a full list of HTML tags by looking at the official HTML specification – which, as I've already suggested, is a very dull document. A list of common HTML tags can be found in Table 2.1.

Two tags in particular are worth pointing out: `SPAN` and `DIV`. These are two tags with opening and closing pairs, which mark out sequences of text. They don't do anything to them – well, DIV starts

Exercise 2 HTML tables

Go to http://www.w3schools.com/html/tryit.asp?filename=tryhtml_intro. Paste in the code snippet for the UEA logo. Paste in the table shown in Listing 2.2.

1. What happens to the image if you change the last three letters from `.png` to `.PNG` ?
2. Try adding another row.
3. What happens if you change the first two `<td>` tags to `<th>` ? What might `<th>` stand for?
4. What happens if you remove one of the closing `<tr>` tags?

a new line – but they are used very often in modern HTML pages to add formatting information, or to add interactive elements. Most web pages are now a mix of HTML – which we know about – and two other technologies, CSS (short for Cascading Style Sheets) and Javascript, a programming language. We don't need to know about them, but they'll crop up in most web pages we look at. You'll often see them used with attributes `id` or `class`. These provide hooks for the CSS or Javascript to latch on to.

Table 2.1: Common HTML tags

Tag	Stands for	Used in
A	Anchor	Links
B	Bold	Formatting text
BLOCKQUOTE	Block-quotes	Formatting text
BODY	Body	HTML structure
BR	Line BReak	Formatting text
DIV	DIVision	HTML structure
EM	EMphasis	Formatting text
HEAD	Head	HTML structure
H1...H6	Heading	Formatting text
I	Italics	Formatting text
IMG	Image	HTML structure
LI	List Item	Lists
OL	Ordered List	Lists
P	Paragraph	Formatting text
PRE	PRE-formatted text	Formatting text
SPAN	Span	HTML structure
TABLE	Table	Tables
TD	Table Data	Tables
TH	Table Header	Tables
TR	Table Row	Tables
UL	Unordered list	Lists

HTML in the wild

Most of the HTML files we've been looking at so far have been minimal examples – or, put less politely, toy examples. It's time to look at some HTML in the wild.

You can see the HTML used to write a page from within your browser. In most browsers, you need to right-click and select 'View Source'.[6]

[6] Apple, in its infinite wisdom, requires you first to change your preferences. Go to Preferences -¿ Advanced, and check 'Show Develop menu in menu bar'.

Let's pick a good example page to start with: Google's homepage circa 1998. The Internet Archive's Wayback Machine can show us what Google looked like on 2nd December 1998 at the following address: `http://web.archive.org/web/19981202230410/http://www.google.com/`.

If you go to that address, you can see the original HTML source code used by Google, starting at line 227. It's fairly simple. Even someone who had never seen HTML before could work out the function of many of the tags by comparing the source with the page as rendered in the browser.

Now look at some of the stuff before line 227. Urgh. Lines 12-111 have a whole load of Javascript – you can completely ignore that. Line 114 has a chunk of style information (CSS) – you can ignore that too. But even the rest of it is peppered with style information, and looks really ugly. It's a headache. But that's the kind of stuff we'll have to work with.

Before you close the source view, check you can do Exercise 3.

Exercise 3 Viewing source

Go to `http://web.archive.org/web/19981202230410/http://www.google.com/` and view source. Try and find the part of the source which says how many times the Google home-page has been captured (7,225 times, at the time of writing). Use Ctrl-F to help you.

1. What tags surround this text? Give a full listing!

2. If you had to describe how to reach this part of the document to another human using just descriptions of the HTML tags, how would you write it out?

Saving a plaintext HTML file

In the next chapter, which deals with Python, we'll need to know how to save plaintext files. So it's useful to check now that you can do this.

Go back to the TryIt examples used in Exercises 1 and 2. Paste these examples in to your text-editor. Try saving this document as a plain-text file with the extension `.html`.[7] It doesn't matter where you save it, but you might want to take this moment to create a new

[7] This extension isn't strictly necessary, but many operating systems and browsers live and die on the basis of correctly-assigned extensions.

folder to save all the work you'll be doing in the next chapter. It also doesn't matter what you call it – but `test.html` would be a good suggestion.

Once you've saved your file, it's time to open it in your browser. You should be able to open a local file in your browser. Some browsers will, as a default, only show you files ending in `.htm` or `.html`.

If your browser offers to save the file you've just tried to open, you've done something wrong. If you get a whole load of gibberish, you've done something wrong. Try googling for the name of your text editor and 'save plain text'.

So what have we done?

So far, we've done the following:

- We've learned about HTML tags and attributes
- We've identified some of the most common HTML tags
- We've seen how these tags are used (and abused) in practice
- We've learnt how to save a plain text file

We'll use all of these in future chapters – but for the next chapter, we can put away the browser and fire up our plain-text editor...

3
Introducing Python

PYTHON IS A PROGRAMMING LANGUAGE that was invented in 1996 by Guido van Rossum. It runs on all modern computers, whether they be Windows, Mac or Linux. It's free, and you can download Python for your computer at http://www.python.org/.

Python comes in two flavours – versions beginning 2.*x* and versions beginning 3.*x*. Versions beginning 2.*x* are more stable; versions beginnign 3.*x* are more experimental.[1] We'll be using the latest version of 2.*x*, currently 2.7.3.

[1] Basically, there's been a clear out under the hood between 2 and 3.

Why choose Python? Python is not the only programming language out there. It's not even the programming language I use most often. But Python has a number of advantages for us:

- It's tidy. Code written in Python is very easy to read, even for people who have no understanding of computer programming. This compares favourably with other languages.[2]
- It's popular. Python is in active development and there is a large installed user base. That means that there are lots of people learning Python, that there are lots of people helping others learn Python, and that it's very easy to Google your way to an answer.
- It's used for web scraping. The site ScraperWiki (which I mentioned in the introduction) hosts web scrapers written in three languages: Python, Ruby, and PHP. That means that you can look at Python scrapers that other people have written, and use them as templates or recipes for your own scrapers.

[2] The language I most often use, Perl, makes it quite easy to produce horrendous code. Some have joked that Perl stands for Pathologically Eclectic Rubbish Lister.

Before we begin

Installing Python

Installing `BeautifulSoup`

First steps

I'LL ASSUME YOU HAVE PYTHON INSTALLED. I'm also going to assume that you're running Python interactively. That is, you're staring at a screen which looks something like Figure 3.1, even though the colors might be different.

```
chris@:~/Documents/teaching/it_skills/scraping$ python
Python 2.7.3 (default, Aug  1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Listing 3.1: Python terminal

It is tradition in computer programming for your first program to say hello to the world. We're going to do that now. Type the following into the Python terminal (or window), and press return.

```
print "Hello World!"
```

That was fun, wasn't it? [3]

[3] If your program didn't say hello to the world, now would be a good time to check out what went wrong.

Now, printing Hello World! to the screen isn't that interesting, because we've fed Python that information. It would be more useful if we could get Python to give us something we didn't already know. So let's use Python as a calculator, and do some difficult sums for us. Type the following in to your Python terminal, and press return after each line.

```
1+1
22.0/7.0
pow(2,16)
```

The second line gives us an approximation to π. The third line gives us the 16th power of 2, or 2^{16}. In the third line we used a *function* for the first time. A function is something which, when given a value or values, returns another value or set of values. Functions are passed values in curved brackets.

Try Exercise 4 now.

Using Python as a calculator can occasionally be handy, but it's not really programming. In order to start programming, we're going to learn about *variables*. You can think of a variable as just a store for some value. It's like when you press M+ on your calculator.

Exercise 4 Python as a calculator

1. What happens if, instead of entering `22.0/7.0`, you enter `22/7`? What do you think Python is doing?

2. What happens if you divide a number by zero?

In order to *assign a value to a variable*, we use the equals sign.

```
pi = 22.0 / 7.0
pi
```

We can even assign variables the values of other variables. Look at the listing below. Before you type this in, ask yourself: what will the results be like?

```
a = 20
b = 10
a = b
print a
print b
```

Numbers aren't the only types of values that variables can hold. They can also hold *strings*.

```
myfirstname = "Christopher"
mylastname = "Hanretty"
```

What happens when you add `myfirstname` and `mylastname`?

Strings can be manipulated in lots of different ways. One way is to take only part of a string. We can do this by taking a *slice* of the string. We can take slices by using square brackets.

```
myfirstname[0:5]
myfirstname[1:5]
myfirstname[:5]
myfirstname[5:]
```

We can also manipulate strings by calling particular methods. A method is a bit like a function, except that instead of writing something like `myfunction(myvalue)`, we write something like `myvalue.mymethod()`. Common methods for strings are `upper`, `upper`, `lower`, and `strip`.[4] So, if we were printing my name for a name badge, we might do the following.

[4] `strip` will come into its own later. It's used for stripping whitespace – spaces, tabs, newlines – from strings.

```
print myfirstname[:5] + ' ' + mylastname.upper()
```

If we wanted to add some extra space between my first name and last name, we might want to insert a tab there. In order to insert a tab properly, we have to use a slightly special construction.

```
print myfirstname[:5] + '\t' + mylastname.upper()
```

The other common special character is the character for a newline,

n .

Looper

One important way in which programming saves effort is through looping. We can ask our program to do something to each item in a list, or to do something for a range of values. For example, if we're interested in calculating the number of possible combinations of n students (2^n), for up to eight students, we might write the following:

```
for i in range(1,9):
    print(pow(2,i))
```

This short listing shows us two new things. First, it shows us a new function, `range`. If you type range(1,9), it will give you the numbers from one, *up to but not including* nine. Second, it shows us the structure of a for loop. It begins with for, ends with a colon, and all subsequent lines which are within the loop are indented. That is, they begin with a tab.

We can also give Python a list to loop over. Strings like `myfirstname` are almost like lists for Python's purposes. So we can do the following:

```
for i in myfirstname:
    print i
```

Not terribly interesting, perhaps, but it illustrates the point.

If we wanted to, we could print the letters in reverse using `range()` and the length function, `len()`.

```
for i in range(len(myfirstname)-1,-1,-1):
    print myfirstname[i]
```

Regular expressions

Regular expressions[5] are a powerful language for matching text patterns. This page gives a basic introduction to regular expressions themselves sufficient for our Python exercises and shows how regular expressions work in Python. The Python "re" module provides regular expression support. In Python a regular expression search is typically written as:

[5] The following section is taken from Nick Parlante's Google Python class, at `http://code.google.com/edu/languages/google-python-class/regular-expressions.html`. That text is made available under the Creative Commons Attribution 2.5 Licence.

```
match = re.search(pat, str)
```

The re.search() method takes a regular expression pattern and a string and searches for that pattern within the string. If the search is successful, search() returns a match object or None otherwise. Therefore, the search is usually immediately followed by an if-statement to test if the search succeeded, as shown in the following example which searches for the pattern 'word:' followed by a 3 letter word (details below):

```
str = 'an example word:cat!!'
match = re.search(r'word:\w\w\w', str)
# If-statement after search() tests if it succeeded
  if match:
    print 'found', match.group() ## 'found word:cat'
  else:
    print 'did not find'
```

The code `match = re.search(pat, str)` stores the search result in a variable named "match". Then the if-statement tests the match – if true the search succeeded and match.group() is the matching text (e.g. 'word:cat'). Otherwise if the match is `false` (`None` to be more specific), then the search did not succeed, and there is no matching text. The 'r' at the start of the pattern string designates a python "raw" string which passes through backslashes without change which is very handy for regular expressions (Java needs this feature badly!). I recommend that you always write pattern strings with the 'r' just as a habit.

The power of regular expressions is that they can specify patterns, not just fixed characters. Here are the most basic patterns which match single chars:

- a, X, 9, ¡ – ordinary characters just match themselves exactly. The meta-characters which do not match themselves because they have special meanings are: . ˆ$ * + ? [ ] \ — ( ) (details below)
- . (a period) – matches any single character except newline '\n'
- \w – (lowercase w) matches a "word" character: a letter or digit or underbar [a-zA-Z0-9_]. Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.
- \b – boundary between word and non-word
- \s – (lowercase s) matches a single whitespace character – space, newline, return, tab, form [\n\r\t\f]. \S (upper case S) matches any non-whitespace character.
- \t, \n, \r – tab, newline, return
- \d – decimal digit [0-9] (some older regex utilities do not support but \d, but they all support \w and \s)
- ≙ start, $ = end – match the start or end of the string

- \ – inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can put a slash in front of it, \@, to make sure it is treated just as a character.

Conclusion

This chapter has given you a whistle-stop introduction to some features of Python. You haven't really been able to use any of these features in any programs, you've just learn that they exist. That's okay. The next chapters will show you what full programs look like, and hopefully you'll come to recognize some of the structure and features you've just learned in those programs.

4
Extracting some text

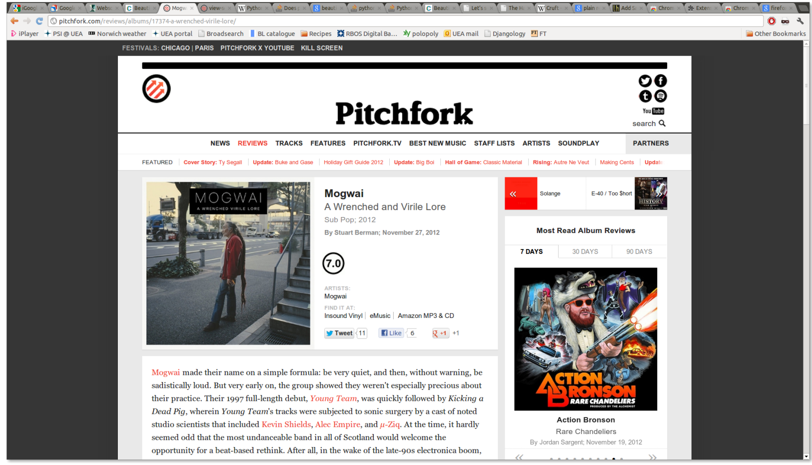
WEB PAGES INCLUDE A LOT OF IRRELEVANT FORMATTING. Very often, we're not interested in the images contained in a page, the mouse-overs that give us definitions of terms, or the buttons that allow us to post an article to Facebook or retweet it. Some browsers now allow you to read pages without all of this irrelevant information.[1] In this first applied chapter, we're going to write some Python scrapers to extract the text from a page, and print it either to the screen or to a separate file. This is going to be our first introduction to the `BeautifulSoup` package, which is going to make things very easy indeed.

[1] Safari's 'Reader' feature; some Instapaper features.

The example

The example I'm going to use for this chapter is a recent review from the music website, Pitchfork.com. In particular, it's a review of the latest (at the time of writing) Mogwai album, *A Wrenched and Virile Lore*.[2] You can find it at `http://pitchfork.com/reviews/albums/17374-a-wrenched-virile-lore/`. When you open it in your browser, it should look something like Figure 4.1.

[2] No, I don't know what that means either.



Listing 4.1: Pitchfork.com review

You can take a look at the source of the web page by right clicking and selecting View Source (or the equivalent in your browser). The

source is not that readable – it's 138 lines long, but many of those lines are very long indeed. We're interested in the start of the review text itself, beginning 'Mogwai made their name'. Try searching for 'made their name' in the source. You should find it on line 55 (a horrifically long line). See Listing 4.2.

How are we supposed to make any sense of that? Well, let's look at where the review begins. We seen that 'Mogwai' is between opening and closing link tags (`<a>` and `</a>`). Those take us to a round up of all Pitchfork articles on Mogwai, which we don't want. If we go back before that, we see that the first line is wrapped in opening and closing paragraph tags. That's helpful: we'll definitely be interested in text contained in paragraph tags (as opposed to free-floating text). But the tag that's really helpful is the one before the opening paragraph tag:

```
<div class="editorial">
```

Ah-ha! What is this editorial class? Is it something that marks out the beginning of a review? We're going to test later.

Our first Python scraper

I'm going to provide the source for our very first Python scraper. I'll list the code, and then explain it line by line. The code is in Listing 4.3

Here's a line-by-line run-down:

- Lines 1 and 2 load, or import, two different packages, called `urllib2` and `BeautifulSoup`. `urllib2` is used to handle getting stuff from the web.[3] `BeautifulSoup` is used to handle the HTML. We need to tell Python to import `BeautifulSoup` `from` bs4, because there are older versions kicking around which we could import `from`.
- Line 4 assigns the address of the review we're interested in to a variable called `start`.
- Line 5 says, 'take this `urllib2` package, and take its `urlopen` function, and apply that to the variable start. Then, when you've done that, apply the `read` function to the result. The `urlopen` function expects to be given a web address; we could have found that out by reading the documentation for the library[4], but here we're taking it on trust.
- Line 6 says (more or less) take the page you open and make some soup from it.
- Line 7 says, take the soup, get the text from it, and print it.

What kind of output does this give us? Listing 4.4 shows the first four lines.

[3] It's short for URL library 2, and URL is short for Uniform Resource Locator, or what you or I would call a web address.

[4]

Listing 4.2: Pitchfork.com code

html/mogwai.html

```
1 </div> <div class="info"> <h1> <a href="/features/update/"> Update </a> </h1> <h2><a href="/
       features/update/8965-mount-kimbie/">Mount Kimbie</a></h2> <div class="abstract"><p>
       Following their excellent 2010 debut LP,  <i>Crooks &amp; Lovers, </i>subtle UK bass duo
        Mount Kimbie were picked up by the estimable folks at Warp Records. They talk to Larry
       Fitzmaurice about their forthcoming album and why their studio is a dump.</p></div> </div
       > </li> </script> </div> </div> <div id="content"> <div id="ad-nav"> <div class="ad-unit
       autoload" id="ad-unit-Strip_Reviews" data-unit="Strip_Reviews"></div> </div> <div id="
       main"> <ul class="review-meta"> <li data-pk="18553"> <div class="artwork"> <img src="http
       ://cdn.pitchfork.com/albums/18553/homepage_large.b79481ae.jpg" /> </div> <div class="info
       "> <h1><a href="/artists/2801-mogwai/">Mogwai</a></h1> <h2>A Wrenched and Virile Lore</h2
       > <h3> Sub Pop; 2012 </h3> <h4> By <address>Stuart Berman</address>; <span class="pub-
       date">November 27, 2012</span> </h4> <span class="score score-7-0"> 7.0 </span> <div
       class="bnm-label"> </div> <ul class="outbound"> <li class="first"> <h1>Artists:</h1> <a
       href="/artists/2801-mogwai/" class="first last"> Mogwai </a> </li> <li > <h1>Find it at:<
       /h1> <a href="http://www.insound.com/Mogwai/A/21837/&from=47597" rel="nofollow" target="
       _blank" class="first">Insound Vinyl</a> <a rel="nofollow" target="_blank" href="http://
       www.emusic.com/artist/Mogwai/?fref=150242">eMusic</a> <a href="http://www.amazon.com/s/?
       url=search-alias%3Daps&tag=p4kalbrevs-20&field-keywords=Mogwai%20A%20Wrenched%20Virile%20
       Lore" rel="nofollow" target="_blank" class="last">Amazon MP3 &amp; CD</a> </li> <li class
       ="last"> <div id="social-httppitchforkcomreviewsalbums17374" class="social-deferred"> <
       div class="lazy" data-content=" &lt;script class=&quot;social&quot; type=&quot;text/
       javascript&quot;&gt; $(function() { p4k.ui.social( &#39;social-
       httppitchforkcomreviewsalbums17374&#39;, &#39;Mogwai: A Wrenched Virile Lore&#39;, &#39;
       http://pitchfork.com/reviews/albums/17374\u002Da\u002Dwrenched\u002Dvirile\u002Dlore
       /&#39;); }); &lt;/script&gt; "></div> </div> </li> </ul> </div> </li> </ul> <div class="
       object-detail"> <div class="editorial"> <p><a href="http://pitchfork.com/artists/2801-
       mogwai/" target="_blank">Mogwai </a>made their name on a simple formula: be very quiet,
       and then, without warning, be sadistically loud. But very early on, the group showed they
        weren't especially precious about their practice. Their 1997 full-length debut, <a href=
       "http://pitchfork.com/reviews/albums/11600-young-team-2008-edition/" target="_blank"><i>
       Young Team</i></a>, was quickly followed by <i>Kicking a Dead Pig</i>, wherein <i>Young
       Team</i>'s tracks were subjected to sonic surgery by a cast of noted studio scientists
       that included <a href="http://pitchfork.com/artists/4045-kevin-shields/" target="_blank">
       Kevin Shields</a>, <a href="http://pitchfork.com/artists/1342-alec-empire/" target="
       _blank">Alec Empire</a>, and <i><a href="http://pitchfork.com/artists/4402--ziq/">  </a><
       /i><a href="http://pitchfork.com/artists/4402--ziq/" target="_blank">-Ziq</a>. At the
       time, it hardly seemed odd that the most undanceable band in all of Scotland would
       welcome the opportunity for a beat-based rethink. After all, in the wake of the late-90s
       electronica boom, remix albums had effectively replaced live albums as the default cash-
       cow-milking measure for rock bands (as acknowledged by the collection's piss-take of a
       title), and the ample negative space in the band's music presents producers with a large
       canvass to color in. But despite its impressive cast and elaborate double-CD presentation
       , <i>Dead Pig</i> ultimately sounded like random attempts at applying Mogwai's metallic
       noise to the darker strains of electronic music of the era (drill and bass, digital
       hardcore), to the point of using its entire second disc to determine who could produce
       the most gonzo version of the band s epic signature track "Mogwai Fear Satan". (<a href
       ="http://www.youtube.com/watch?v=0T3l35zohFM" target="_blank" rel="nofollow">Shields'
       titanic take</a> won in a landslide.)</p> <p>Fourteen years later, the band's second
       remix album, <a href="http://pitchfork.com/news/48102-mogwai-share-remix-album-details/"
       target="_blank"><i>A Wrenched Virile Lore</i></a>, arrives as a more cohesive work,
       presenting a cerebral, alternate-universe reimgination of Mogwai's 2011 release, <a href=
       "http://pitchfork.com/reviews/albums/15100-hardcore-will-never-die-but-you-will/" target=
       "_blank"><i>Hardcore Will Never Die But You Will</i></a>. Despite its severe title, <i>
       Hardcore</i> counts as the band s most blissful and texturally rich to date. <i>
       Hardcore'</i>s sound provides producers with more jumping-off points than the band's
       mountainous art-rock would normally allow. If <i>Kicking a Dead Pig</i> was mostly about
       giving Mogwai's atomic guitar eruptions a mechanized makeover, <i>A Wrenched Virile Lore<
       /i> repositions the songs' central melodies in more splendorous surroundings. <a href="
       http://pitchfork.com/news/48102-mogwai-share-remix-album-details/" target="_blank">In the
        hands of Justin K. Broadrick</a>, the post-punky krautrock of "George Square Thatcher
       Death Party" becomes the sort of gently ascendant, anthemic opener that you could imagine
        blaring out of a stadium to kick-off an Olympics ceremony; Pittsburgh prog-rockers Zombi
        hear the mournful piano refrain of "Letters to the Metro" as the basis for a glorious,
       strobe-lit Trans Europe Express flashback. Or in some cases, the material is stripped
       down to its essence: Glaswegian troubador R.M. Hubbert re-routes the motorik pulse of "
       Mexican Grand Prix" off the speedway into the backwoods and transforms it into a hushed,
       Jose Gonzalez-like acoustic hymn, while San Fran neo-goth upstarts <a href="http://
       pitchfork.com/artists/28699-the-soft-moon/" target="_blank">the Soft Moon</a> scuff away
       the surface sheen of <a href="http://www.pitchfork.com/forkcast/15544-san-pedro/" target=
       "_blank">"San Pedro"</a> to expose the seething menace lurking underneath. However, there
        are limits to this approach: Umberto's distended, ambient distillation of "Too Raging to
        Cheers" simmers down this already serene track to the point of rendering it
       inconsequential.</p> <p>Like <i>Hardcore</i>, <i>A Wrenched Virile Lore</i> features 10
       tracks, though it only references eight of the originals. However, even the mixes that
       draw from the same songs are different enough in approach and sequenced in such a way
       that the reappearances feel like purposeful reprises: Klad Hest's drum and bass-rattled
       redux of "Rano Pano" finds its sobering aftershock in <a href="http://www.pitchfork.com/
       artists/1917-tim-hecker/" target="_blank">Tim Hecker's</a> haunted and damaged revision,
       while <a href="http://pitchfork.com/artists/876-cylob/" target="_blank">Cylob's</a>
       cloying synth-pop take on <i>Hardcore</i>'s opener "White Noise"-- which fills in the
       original's instrumental melody with lyrics sung through a vocoder-- is redeemed by UK
       composer Robert Hampson's 13-minute soft-focus dissolve of the same track. Renamed "La
       Mort Blanche", its appearance provides both a full-circle completion of this record (by
       book-ending it with mixes by former Godflesh members) while serving as a re-entry point
       back into <i>Hardcore</i> by reintroducing the source song's main motifs and widescreen
       vantage. But more than just inspire a renewed appreciation for <i>Hardcore</i>, <i>A
       Wrenched Virile Lore</i> potentially provides Mogwai with new avenues to explore now that
        they're well into their second decade, and perhaps instill a greater confidence in the
       idea that their identity can remain intact even in the absence of their usual skull-
       crushing squall.</p> </div> </div> </div> <div id="side"> <ul class="object-prevnext-
       minimal"> <li class="prev"> <a href="/reviews/albums/17373-true/"> <img src="http://cdn.
       pitchfork.com/albums/18552/list.cd511d55.jpg" /> <h1>Solange</h1> </a> </li> <li class="
       next"> <a href="/reviews/albums/17363-the-history-channel/"> <img src="http://cdn.
       pitchfork.com/albums/18540/list.745b89a0.jpg" /> <h1>E-40 / Too $hort</h1> </a> </li> </
       ul> <div class="ad-unit autoload" id="ad-unit-Rev_Albums_300x250" data-unit="
       Rev_Albums_300x250"></div> <div class="most-read albumreviews-recordreview"> <h1>Most
       Read Album Reviews</h1> <div class="tabbed"> <ul class="tabs"> <li class="first "> <a
       href="#">7 Days</a> </li> <li class=" "> <a href="#">30 Days</a> </li> <li class=" last">
        <a href="#">90 Days</a> </li> </ul> <ul class="content"> <li class="first "> <ul class="
       object-list carousel" data-transition="fade" data-autoadvance="5000" data-randomize-
       initial="on" > <li class="first "> <a href="/reviews/albums/17064-the-disintegration-
       loops/"> <div class="artwork"> <div class="lazy" data-content=" &lt;img src=&quot;http://
       cdn4.pitchfork.com/albums/18243/homepage_large.06fc9f79.jpg&quot; /&gt; "></div> </div> <
       div class="info"> <h1>William Basinski</h1> <h2>The Disintegration Loops</h2> <h3> By
       Mark Richardson; November 19, 2012 </h3> </div>
```

Listing 4.3: Scraper to print full text

python_code/mogwai1.py

```
import urllib2
from bs4 import BeautifulSoup

start = 'http://pitchfork.com/reviews/albums/17374-a-wrenched-virile-lore/'
page = urllib2.urlopen(start).read()
soup = BeautifulSoup(page)
print (soup.get_text())
```

Listing 4.4: Scraper output

python_code/mogwai1.out

```
htmlvar NREUMQ=NREUMQ||[];NREUMQ.push(["mark","firstbyte",
    new Date().getTime()]) Mogwai: A Wrenched Virile Lore |
     Album Reviews | Pitchfork
[if IE 7]> <link rel="stylesheet" type="text/css" href="
    http://cdn.pitchfork.com/desktop/css/ie7.css" /> <![
    endif][if IE]> <script src="http://cdn4.pitchfork.com/
    desktop/js/excanvas.js"></script> <![endif] var p4k =
    window.p4k || {}; p4k.init = []; p4k.init_once = [];
    p4k.ads = {};  var __js = []; var __jsq = []; __jsq.
    push(function() { $(function(){p4k.core.init()}) })__js
    .push("https://www.google.com/jsapi?key\
    u003DABQIAAAAd4VqGt0ds\
    u002DTq6JhwtckYyxQ7a1MeXZzsUvkG0s95E1kgV0L_HRTWzR1RoBGaK0NcJfQcDtUuCXrHcQ
    "); __DFP_ID__ = "1036323"; STATIC_URL = "http://cdn.
    pitchfork.com/";
 var _gaq = _gaq || [];
 google  var _gaq = _gaq || []; _gaq.push(['_setAccount', '
     UA-535622-1']); _gaq.push(['_trackPageview']); (
     function() { var ga = document.createElement('script')
     ; ga.type = 'text/javascript'; ga.async = true; ga.src
      = ('https:' == document.location.protocol ? 'https://
     ssl' : 'http://www') + '.google-analytics.com/ga.js';
     var s = document.getElementsByTagName('script')[0]; s.
     parentNode.insertBefore(ga, s); })();
```

Hmm.... not so good. We've still got a lot of crap at the top. We're going to have to work on that. For the moment, though, check you can do Exercise 5 before continuing.

Exercise 5 Your first scraper

Try running the code given in 4.3. Try it first as a saved program. Now try it interactively. Did it work both times?
Try removing the 'http://' from the beginning of the web address. Does the program still work?
You could have learnt about the `get_text()` function from the `BeautifulSoup` documentation at `http://www.crummy.com/software/BeautifulSoup/bs4/doc/`. Go there now and look at some of the other ways of accessing parts of the soup. Using Python interactively, try `title` and `find_all`. Did they work?

Looping over Ps and DIVs

The results shown in Listing 4.4 were pretty disappointing. They were disappointing because we were printing out any and all text on the page, even some text that wasn't really text but were really formatting or interactive elements. We can change that by being more specific, and in effect asking `BeautifulSoup` to just get us text which comes wrapped in paragraph tags.

Listing 4.5 shows a first attempt at this. Once again, I'll present the listing, then go over it line by line.

Listing 4.5: Looping over paragraphs

python_code/mogwai2.py

```
1 import urllib2
2 from bs4 import BeautifulSoup
3
4 start = 'http://pitchfork.com/reviews/albums/17374-a-
      wrenched-virile-lore/'
5 page = urllib2.urlopen(start).read()
6 soup = BeautifulSoup(page)
7
8 for para in soup.find_all('p'):
9     print(para.get_text())
```

Lines 1 to 7 are essentially the same as in the previous Listing 4.3. Line 8 is the novelty, and it introduces a `for` loop of the kind we saw in the previous chapter. What this line says is, take the soup, and using it, apply the `find_all` function, passing it as an argument the string 'p' (to represent the opening paragraph tag `<p>`). That returns a list, which the for loop iterates over. We call the list item we're working with at any given time `para`, but you could call it anything you like.

Line 9 then tells the program to take the current list item, and apply to it the `get_text` that we used before on all of the soup.

If you run the program in Listing 4.5, you'll see that it produces output which is much closer to what we want. It doesn't give us

a whole load of crap at the start, and it starts with the text of the review.

Unfortunately, it's still not perfect. You'll see at the bottom of the output that there are a number of sentence fragments ending in ellipses (...). If you go back to the page in your web browser, as shown in Figure 4.1, you'll see that these sentence fragments are actually parts of boxes linking to other Mogwai reviews. We don't want to include those in our output. We need to find some way of becoming more precise.

That's where the `div` we saw earlier comes in to play. Here's the listing; explanation follows.

Listing 4.6: Looping over divisions

python_code/mogwai3.py

```
1  import urllib2
2  from bs4 import BeautifulSoup
3
4  start = 'http://pitchfork.com/reviews/albums/17374-a-
       wrenched-virile-lore/'
5  page = urllib2.urlopen(start).read()
6  soup = BeautifulSoup(page)
7
8  for div in soup.find_all('div',{'class':'editorial'}):
9  ␣␣for para in div.find_all('p'):
10 ␣␣␣␣print (para.get_text() + '\n')
```

This scraper is identical to the previous scraper, except that we've now got two `for` loops, and the code that sets up the first of these `for` loops (on line 8) is a little bit more complicated. Line 8 uses the same `find_all` function we used before, except it (a) changes the main argument from `p` to `div`, and (b) adds a second argument, which Python calls a *dictionary*. The *key* for the dictionary is `class`, which corresponds to the *attribute* in the HTML source code we saw in Listing 4.2. The corresponding *value* for the dictionary is `editorial`, which corresponds to the value `editorial` used by Pitchfork editors. In plain English, Line 8 says, 'find me all the `divs` that have a class which equals editorial, and loop over them, storing the result in the variable called `div`.

We now need to work on this `div`. And so, we change Line 9. Instead of looping over all the paragraphs in the soup, we loop over all the paragraphs *in this particular div*. We then print the full text of each of these.

If you run this scraper, you'll see that it finally gives us what we wanted – the full text of the review, with no extraneous formatting.

Recap

So how did we arrive at this wonderful result? We proceeded in four steps.

First, we identified the portion of the web page that we wanted, and found the corresponding location in the HTML source code. This is usually a trivial step, but can become more complicated if you want to find multiple, non-contiguous parts of the page.

Second, we identified a particular HTML tag which could help us refine our output. In this case, it was a particular `div` which had a `class` called `editorial`. There was no guarantee that we would find something like this. We were lucky because well built web sites usually include classes like this to help them format the page.

Third, we used `BeautifulSoup` to help us loop over the tags we identified in the second step. We used information both on the particular `div` and the paragraphs containing the text within that div.

Fourth, we used `BeautifulSoup`'s `get_text` on each paragraph, and printed each in turn.

This is a common structure for extracting text. Whilst the tags you use to identify the relevant portion of the document might differ, this basic structure can and ought to guide your thinking.

Getting this out

So far, we've been happy just printing the results of our program to the screen. But very often, we want to save them somewhere for later use. It's simple to do that in Python.

Here's a listing which shows how to write our output to a file called `review.txt`

Listing 4.7: File output

python_code/mogwai4.py

```
import urllib2
from bs4 import BeautifulSoup
import codecs

start = 'http://pitchfork.com/reviews/albums/17374-a-
    wrenched-virile-lore/'
page = urllib2.urlopen(start).read()
soup = BeautifulSoup(page)

outfile = codecs.open('review.txt', 'w','utf-8')

for div in soup.find_all('div',{'class':'editorial'}):
  for para in div.find_all('p'):
    outfile.write(para.get_text() + '\n')

outfile.close()
```

There are two changes you need to take note of. First, we import a new package, called codecs. That's to take care of things like accented characters, which are represented in different ways on different web-pages. Second, instead of calling the `print` function,

we called the `write` function on a file we open in line 9. Remember to close the file when you're finished!

A taster

We've worked through printing out a review from a particular website. Now it's time to try a different example.

The BBC lists many of the interviewees who have appeared on the Radio 4 programme Desert Island Discs. One of the most recent 'castaways' was Edmund de Waal. You can see his selections at http://www.bbc.co.uk/radio4/features/desert-island-discs/castaway/2ada59ff#b01p314n.

We'll return to this example later – but try Exercise 6 to see how much you can extract from this page.

Exercise 6 Castaway scraper

Try looking at the source of the De Waal page.

1. What tags surround the artist of each track?
2. Is this on its own enough to extract the artist? If not, what `div` must you combine it with?
3. Write a program to extract the eight artists chosen by Edmund de Waal.
4. Look again at the `BeautifulSoup` documentation at http://www.crummy.com/software/BeautifulSoup/bs4/doc/. In particular, look at the section headed '`.next_sibling` and `.previous_sibling`'. Can you use `.next_sibling` to print out the track?

5
Downloading files

In the previous section, Python helped us clear a lot of the junk from the text of web pages. Sometimes, however, the information we want isn't plain text, but is a file – perhaps an image file (.jpg, .png, .gif), or a sound file (.mp3, .ogg), or a document file (.pdf, .doc). Python can help us by making the automated downloading of these files easier – particularly when they're spread over multiple pages.

In this chapter, we're going to learn the basics of how to identify links and save them. We're going to use some of the regular expression skills we learned back in 3. And we're going to make some steps in identifying sequences of pages we want to scrape.

The example

As our example we're going to use a selection of files hosted on UbuWeb (`www.ubu.com`). UbuWeb describes itself as "a completely independent resource dedicated to all strains of the avant-garde, ethnopoetics, and outsider arts". It hosts a number of out-of-circulation media, including the complete run of albums released by Briano Eno's short-lived experimental record label Obscure Records.

The first iteration

Let's start by imagining that we're interested in downloading a single file: the MP3 of David Toop's *Do The Bathosphere*.[1] You can find it at `http://ubumexico.centro.org.mx/sound/obscure_4/Obscure-4_Toop-Eastly_07-Do-The-Bathosphere_1975.mp3`

[1] Toop is a sound artist who is currently Senior Research Fellow at the London College of Communication.

We're going to write a Python program to download this mp3. This is (moderately) insane. It's using a sledgehammer to crack a peanut. Anyone normal would just open the address in their browser and save the file when they're prompted to do so. But by writing this program we'll learn techniques that we'll apply later to better use.

Let's walk through Listing 5.1. We begin in Lines 1-3 by importing some packages, as we normally do. We're importing `urllib2`, its predecessor `urllib`, and the regular expressions package we say in Chapter 3, `re`. In Line 5, we create a new variable, `url`, with the address of the file we're interested in downloading. In Line 7, we try

Listing 5.1: Saving a file

python_code/toop.py

```
from urllib2 import urlopen
from urllib import urlretrieve
import re

url = 'http://ubumexico.centro.org.mx/sound/obscure_4/
    Obscure-4_Toop-Eastly_07-Do-The-Bathosphere_1975.mp3'

filename = url.split("/")[-1]
mydir = "mp3s/"

urlretrieve(url,mydir + filename)
```

and be a little smart, and create a filename for the saved file. We have to give a filename – Python's not going to invent one for us. So we'll use the last part of the address itself – the part after the last forward slash.

In order to get that, we take the url, and call the `split` function on it. We give the split function the character we want to split on, the forward slash. That split function would normally return us a whole list of parts. But we only want the last part. So we use the minus notation to count backwards (as we saw before).

In Line 8, we create another variable to tell Python which directory we want to save it in. Remember to create this directory, or Python will fail. Remember also to include the trailing slash.

Finally, line 10 does the hard work of downloading stuff for us. We call the `urlretrieve` function, and pass it two arguments – the address, and a path to where we want to save the file, which has the directory plus the filename.

One thing you'll notice when you try to run this program – it will seem as if it's not doing anything for a long time. It takes time to download things, especially from sites which aren't used to heavy traffic. That's why it's important to be polite when scraping.

Looping over links

The previous example was insane because the effort of writing a Python script was much greater than the effort of right-clicking, selecting 'Save as...', and saving the file from the browser. That might change if we had a page with hundreds of links on it, all of which we wanted to save. The UbuWeb page `http://www.ubu.com/sound/obscure_04.html` has links to all of the tracks on the Eastley/Toop LP *New and Rediscovered Musical Instruments*. How might we save all of those? Listing 5.2 shows the way.

Notice that we've loaded a few more packages at the top of the code. The main action here is happening in lines 10 and 11. We're asking `BeautifulSoup` to find everything that has a particular attribute – in this case, an href attribute. Remember from 2 that href attributes are used in links to provide address to things – hyper-

Listing 5.2: Saving multiple files

python_code/toop2.py

```
import re
import urlparse
from urllib2 import urlopen
from urllib import urlretrieve
from bs4 import BeautifulSoup

url = 'http://www.ubu.com/sound/obscure_04.html'
soup = BeautifulSoup(urlopen(url))

for thetrack in soup.find_all(href=re.compile("mp3$")):
    filename = thetrack["href"].split("/")[-1]
    print filename
    urlretrieve(thetrack["href"], filename)
```

references. In this case, we're not giving `BeautifulSoup` a particular value for this attribute – we're giving it a *regular expression*. Specifically, we're asking it to identify all things that have an attribute `href` which has a value which has the text '.mp3' at the very end.[2]

We taken the results of that `find_all` operation, and as we iterate over each of them we save the result in the variable `thetrack`. Now, although we asked `BeautifulSoup` to select based on the value of the `href`, it's not giving us the href directly, but rather the containing `<a>` tag. So we need to tell Python that we want the `href` attribute before (in Line 11) we split it into pieces.

[2] We add the dollar sign to the end of the regular expression to say that we're only interested in stuff at the very end of the text. It's possible – but unlikely – that a file could have mp3 in the middle of it and be of the wrong type (say, a file called 'how-to-download-mp3s-illegally.doc').

Looping over pages, over links

This is all fine if we're just interested in a single page. But frankly, there are add-ons for your browser that will help you download all files of a particular type on a single page.[3] Where Python really comes in to its own is when you use it to download multiple files from multiple pages all spinning off a single index page. Got that? Good. We're going to use a different example to talk about it.

[3] DownloadThemAll! for Firefox is very good.

The Leveson Inquiry into Culture, Practice and Ethics of the Press delivered its report on 29th November 2012. It was based on several thousand pages of written evidence and transcripts, all of which are available online at the Inquiry website.[4] Downloading all of those submissions – perhaps you're going away to read them all on a desert island with no internet – would take a very, very long time. So let's write a scraper to do so.

[4] `http://www.levesoninquiry.org.uk/`

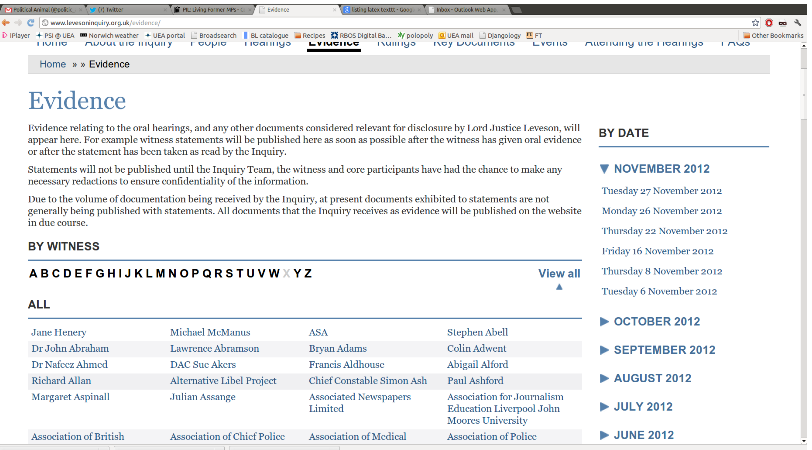
If you go to `http://www.levesoninquiry.org.uk/evidence/` and click on 'View all', you'll see the full list of individuals who submitted evidence. See Figure 5.3 if you're in any doubt.

We're interested in the links to the pages holding evidence of specified individuals: pages featuring Rupert Murdoch's evidence[5], or Tony Blair's [6]. The common pattern is in the link – all the pages we're interested in have `witness=` in them. So we'll build our parser on that basis.

[5] `http://www.levesoninquiry.org.uk/evidence/?witness=rupert-murdoch`

[6] `http://www.levesoninquiry.org.uk/evidence/?witness=tony-blair`

When we get to one of those pages, we'll be looking to download



Listing 5.3: Leveson inquiry website

all of the PDF transcripts. So we'll amend the same regular expression code we used to download MP3s above. Listing 5.4 shows the necessary steps.

Listing 5.4: Links on multiple pages

python_code/leveson.py

```
import re
import urlparse
import time
from urllib2 import urlopen
from urllib import urlretrieve
from bs4 import BeautifulSoup

start = 'http://www.levesoninquiry.org.uk/evidence'
baseurl = 'http://www.levesoninquiry.org.uk'

soup = BeautifulSoup(urlopen(start))

for witness in soup.find_all(href=re.compile("witness=")):
  newpage = baseurl + witness["href"]
  time.sleep(.5)

  soup = BeautifulSoup(urlopen(newpage))
  for pdftranscript in soup.find_all(href=re.compile("pdf$"),text=re.compile("Transcript")):
    print baseurl + pdftranscript["href"]
```

We start with the usual calling of certain packages, before defining two variables. The first variable, `start`, is the page we'll go to first. The second variable, `baseurl`, is something we're going to use to make sense of the links we get. More on that later.

We start the meat of the program on line 13, where we iterate over links which contain `witness=`. We can access the address for each of those links by `witness["href"]`. However, these links are *not sufficient on their own.* They're what's known as relative links. They miss out all the `http://www.leveson...` guff at the start. The only way of knowing whether links are relative or not is to look carefully at the code.

Because of this, we combine the base url, with the value of the href attribute. That gives us the full address. (Check if you don't

believe).

We then pause a little to give the servers a rest, with `time.sleep`, from the `time` package. We then open a new page, with the full address we just created! (We store it in the same soup, which might get confusing).

Now we're on the witness page, we need to find more links. Just searching for stuff that ends in .pdf isn't enough; we need just PDF transcripts. So we also add a regular expression to search on the text of the link.

To save bandwidth (and time!) we close by printing off the base URL together with the relative link from the `href` attribute of the `<a>` tag. If that leaves you unsatisfied, try Exercise 7.

Exercise 7 Leveson scraper

1. Amend the source found in Listing 5.4 to download all text transcripts. (Text files are much smaller than PDF files; the whole set will take much less time to download).

2. Turn your wireless connection off and try running the program again. What happens?

6
Extracting links

The idea of the link is the fundamental building block not only of the web but of many applications built on top of the web. Links – whether they're links between normal web pages, between followers on Twitter, or between friends on Facebook – are often based on latent structures that often not even those doing the linking are aware of. We're going to write a Python scraper to extract links from a particular web site, the AHRC website. We're going to write our results to a plain text spreadsheet file, and we're going to try and get that in to a spreadsheet program so we can analyze it later.

AHRC news

The AHRC has a news page at http://www.ahrc.ac.uk/News-and-Events/News/Pages/News-Listing.aspx. It has some image-based links at the top, followed by a list of the 10 latest news items. We're going to go through each of these, and extract the external links for each item.

Let's take a look at the code at the time of writing. An excerpt is featured in Listing 6.1.

Listing 6.1: AHRC code

```
<div class="item">
    <div class="head">
        <p><a id="ctl00_PlaceHolderMain_g_dab778c6_1dbb_41b4_8545_83e22d250b11_ctl00_rptResults_ctl00_hlNews" href="http://www.ahrc.ac.uk/News-and-Events/News/Pages/Join-in-the-Moot-today.aspx">Join in the Moot today</a></p>
    </div>
    <p><strong>Date: </strong>19/11/2012</p>
    <p>The Digital Transformations Moot takes place in London today. </p>
</div>
```

We're going to pivot off the `div` with `class` of `item`, and identify the links in those `div`s. Once we get those links, we'll go to those news items. Those items (and you'll have to trust me on this) have `div`s with `class` of `pageContent`. We'll use that in the same way.

Listing 6.2: AHRC link scraper

python_code/ahrc.py

```
import re
import urlparse
import codecs
from urllib2 import urlopen
from urllib import urlretrieve
from bs4 import BeautifulSoup

start = 'http://www.ahrc.ac.uk/News-and-Events/News/Pages/
    News-Listing.aspx'
outfile = codecs.open('ahrc_links.csv', 'w','utf-8')

soup = BeautifulSoup(urlopen(start))

for newsitem in soup.find_all('div',{'class':'item'}):
  for newslink in newsitem.find_all('a'):
    if newslink.has_key('href'):
      newpage = newslink['href']
      soup = BeautifulSoup(urlopen(newpage))

      for pagecontent in soup.find_all('div',{'class':'
    pageContent'}):
        for link in pagecontent.find_all('a'):
          if link.has_key('href'):
            linkurl = link['href']
            if linkurl[:4] == 'http':
              site = re.sub('/.*','',linkurl[7:])
              outfile.write(newpage + '\t' + site + '\n')

outfile.close()
```

Listing 6.2 shows the eventual link scraper. You should notice two things. First, we're starting to use if-tests, like we discussed back in Chapter 3. Second, we've got quite a lot of loops – we loop over news items, we have a (redundant) loop over content divs, and we loop over all links. The combination of these two things means there's quite a lot of indentation.

Let me explain three lines in particular. Line 15 tests whether or not the `<a>` tag that we for in the loop beginning Line 14 has an `href` attribute. It's good to test for things like that. There are some `<a>` tags which don't have href attributes.[1]. If you give get to line 16 with just such a tag, Python will choke.

Line 23 takes a particular *slice* out of our link text. It goes from the beginning to the fourth character. We could have made that clearer by writing `linkurl[0:4]` – remember, lists in Python start from zero, not one. We're relying on external links beginning with `http`.

Line 24 uses a regular expression. Specifically, it says, take any kind of character that follows a forward slash, and replace it with nothing – and do that to the variable `linkurl`, from the seventh character onwards. That's going to mean that we get only the website address, not any folders below that. (So, `digitrans.crowdvine.com/pages/watch-live` becomes `digitrans.crowdvine.com/`).

Finally, Line 25 gives us our output. We want to produce a spreadsheet table with two columns. The first column is going to be the AHRC page that we scraped. The second column is going to give us

[1] Instead they have a `name` tag, and act as anchor points. You use them whenever you go to a link with a hash symbol (#) after the `.html`

the link URL that we extracted. Because we want to load this in to a spreadsheet, we need to use a special character to separate the columns. I've gone for a tab, for which we use ‘ ’.

The first four lines of the output of Listing 6.2 are shown below.

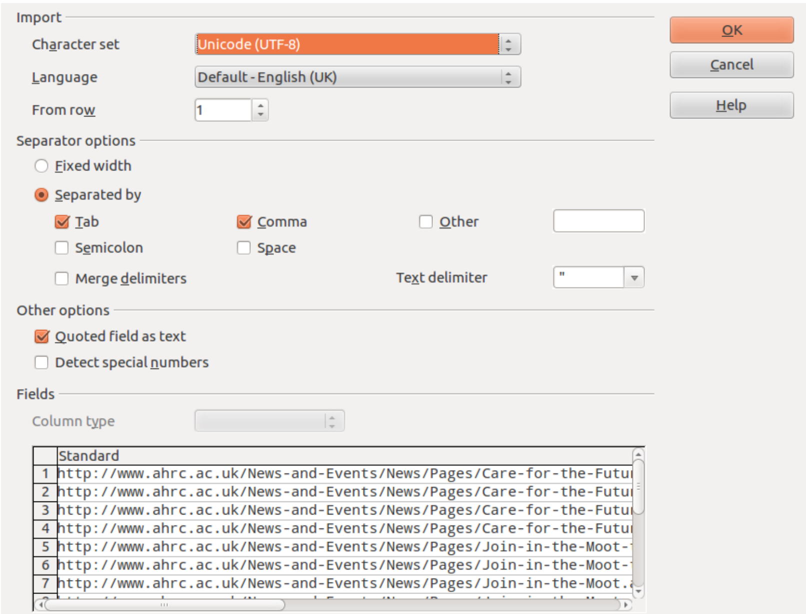
Listing 6.3: AHRC output

ahrc_links.csv

```
1  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣research.sas.ac.uk
2  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣humanities.exeter.ac.uk
3  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣www.exeter.ac.uk
4  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣www.sas.ac.uk
5  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Join-in-the-Moot-today.aspx␣␣digitrans.crowdvine.com
6  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Join-in-the-Moot-today.aspx␣␣digitrans.crowdvine.com
7  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Join-in-the-Moot.aspx␣␣digitrans.crowdvine.com
8  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Join-in-the-Moot.aspx␣␣digitrans.crowdvine.com
9  http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣research.sas.ac.uk
10 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣humanities.exeter.ac.uk
11 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣www.exeter.ac.uk
12 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Care-for-the-Future-and-Science-and-Culture-leaderships-fellows-announced.aspx␣www.sas.ac.uk
13 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Archaeologists-reveal-rare-Anglo-Saxon-feasting-hall.aspx␣␣www.lymingearchaeology.org
14 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Archaeologists-reveal-rare-Anglo-Saxon-feasting-hall.aspx␣␣blogs.reading.ac.uk
15 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Investment-to-promote-innovation-in-additive-manufacturing.aspx␣␣www.innovateuk.org
16 http://www.ahrc.ac.uk/News-and-Events/News/Pages/Hajj-Journey-to-the-Heart-of-Islam.aspx␣␣www.britishmuseum.org
17 http://t.co/3Q41s16I␣␣charades.hypotheses.org
```

I told Python to write this to the file `ahrc_links.csv`. Files that end in CSV are normally comma-separated values files. That is, they use a comma where I used a tab. I still told Python to write to a file ending in .csv, because my computer recognises .csv as an extension for comma-separated values files, and tries to open the thing in a spreadsheet. There is an extension for tab separated values files, .tsv, but my computer doesn't recognise that. So I cheat, and use .csv.

I can do this, because OpenOffice (the spreadsheet I use) intelligently recognises that I'm trying to open a plain-text file, and asks me to check the separator I use. The dialog box I get is shown in Figure 6.4. You can see that there are a whole host of separator options for me to play with.



Listing 6.4: OpenOffice dialog

This is going to form the route for getting our information into a form in which we can analyze things. We're going to get a whole load of information, smash it together with tabs separating it, and open it in a spreadsheet.

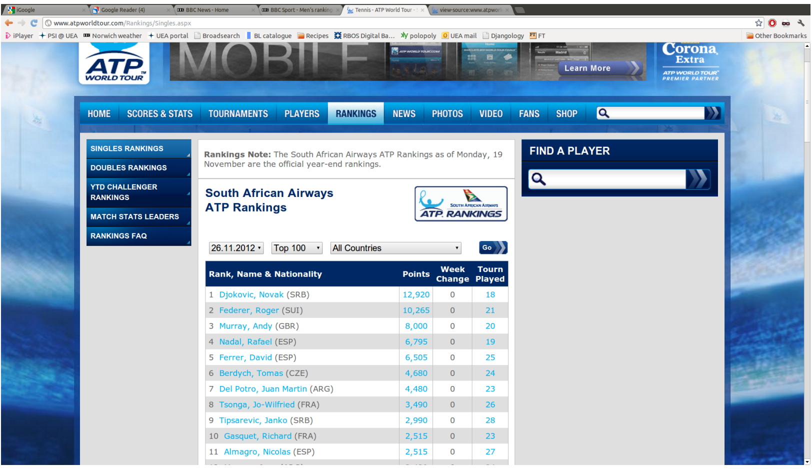
That strategy pays off most obviously when looking at tables – and that's what we're going to look at next.

7
Extracting tables

A lot of information on the web – and particularly the kind of information that we're interested in extracting – is contained in tables. Tables are also a natural kind of thing to pipe through to a spreadsheet. But sometimes the table formatting used for the web gets in the way. We're going to extract tables with a degree of control that wouldn't be possible with manual copy-and-pasting.

Our example

I believe (mostly without foundation) that most people derive most exposure to tables from sports. Whether it's league or ranking tables, almost all of us understand how to understand these kinds of tables. I'm going to use one particular ranking table, the ATP tennis tour ranking. You can see the current men's singles ranking at `http://www.atpworldtour.com/Rankings/Singles.aspx`. The website is shown in Figure 7.1.



Listing 7.1: ATP rankings

Try copying and pasting the table from the ATP tour page into your spreadsheet. You should find that the final result is not very useful. Instead of giving you separate columns for ranking, player name, and nationality, they're all collapsed into one column, making it difficult to search or sort.

You can see why that is by looking at the source of the web page.

Listing 7.2: ATP code

html/atp.html

```
                    <tr class="oddRow">
                        <td class="first">
                <span class="rank">88</span>

                <a href="/Tennis/Players/Zo/J/Jurgen-Zopp.aspx">Zopp, Jurgen</a> (
    EST)</td>

              <td><a href="/Tennis/Players/Zo/J/Jurgen-Zopp.aspx?t=rb">579</a></td>
                        <td>-2</td>
                        <td class="last"><a href="/Tennis/Players/Zo/J/Jurgen-Zopp.aspx
                            ?t=pa&m=s">22</a></td>
                    </tr>
```

You should be able to see that there are only four cells in this table row, whereas we want to extract six pieces of information (rank, name, nationality, points, week change, and tournaments played). What we're going to do is produce an initial version of the scraper which extracts the table as it stands, and then improve things by separating out the first column.

Listing 7.3: ATP code

python_code/atp.py

```python
import re
import urlparse
import codecs
from urllib2 import urlopen
from urllib import urlretrieve
from bs4 import BeautifulSoup
from bs4 import SoupStrainer

start = 'http://www.atpworldtour.com/Rankings/Singles.aspx'
outfile = codecs.open('atp_ranks.csv', 'w','utf-8')

soup = BeautifulSoup(urlopen(start),parse_only =
    SoupStrainer('table',{'class':'bioTableAlt stripeMe'}))

for therow in soup.find_all('tr'):
  for thecell in therow.find_all('td'):
    cellcontents = thecell.get_text().strip()
    cellcontents = re.sub('\n',' ',cellcontents)
    outfile.write(cellcontents + '\t')

  outfile.write('\n')

outfile.close()
```

The first version of our scraper is shown in Listing 7.3. A couple of lines are worth commenting on. First, instead of redundantly looping round all tables with a particular class (in this case, 'bioTableAlt stripeMe', which is the class that the ATP have used for the table we're interested in), I've used `BeautifulSoup`'s SoupStrainer function to extract just that table. This reduces memory usage, making our program more efficient. It also helps make the program more easily intelligible.

Lines 16 through 18 do the formatting of the output. I carry out two operations. First, I strip out whitespace (spaces, tabs, newlines) from the beginning and the end of the cell contents, using Python's built-in `strip` function. Second, because there's still a new line in there, I substitute that with a space. I then write that to file, followed

by my cell separator, which is a tab.[1] Finally, after the end of the for loop, I add a new line so that my spreadsheet isn't just one long row.

How are we going to improve on that? We're going to use some `if` and `else` statements in our code. Essentially, we're going to process the cell contents one way if it has `class` of `first`, but process it in a quite different way if it doesn't. Listing 7.4 shows the listing.

The major differences with respect to the previous listing are as follows. There's a little bit of a trick in Line 14. Because we're going to parse table cells with class first *on the assumption that* they contain spans with the rank, and links, and so on, we're going to ignore the first row of the table, because it has a table cell with class first which doesn't contain a span with the rank, etc., and because if we ask Python to get spans from a table cell which doesn't contain them, it's going to choke.[2] So we take a *slice* of the results returned by BeautifulSoup, omitting the first element.[3]

[1] You could use a comma, but then you would have to deal with commas separating players' first and last names. That's why although we talk about comma separated values files, we mostly used a tab.

[2] We could write code to consider this possibility, but that would make the program more complicated.

[3] Remember Python counts from zero.

Listing 7.4: Improved ATP code

python_code/atp2.py

```
import re
import urlparse
import codecs
from urllib2 import urlopen
from urllib import urlretrieve
from bs4 import BeautifulSoup
from bs4 import SoupStrainer

start = 'http://www.atpworldtour.com/Rankings/Singles.aspx'
outfile = codecs.open('atp_ranks2.csv', 'w','utf-8')

soup = BeautifulSoup(urlopen(start),parse_only =
    SoupStrainer('table',{'class':'bioTableAlt stripeMe'}))

for therow in soup.find_all('tr')[1:]:
  for thecell in therow.find_all('td'):
    if (thecell.has_key('class') and (thecell['class'][0]
    == 'first')):
      rank = thecell.find('span')
      rank = rank.get_text()
      name = thecell.find('a')
      name = name.get_text()
      nationality = re.search('\([A-Z]{3}\)',thecell.
    get_text()).group()
      outfile.write(rank + '\t' + name + '\t' + nationality
     + '\t')
    else:
      cellcontents = thecell.get_text().strip()
      cellcontents = re.sub('\n',' ',cellcontents)
      outfile.write(cellcontents + '\t')

  outfile.write('\n')

outfile.close()
```

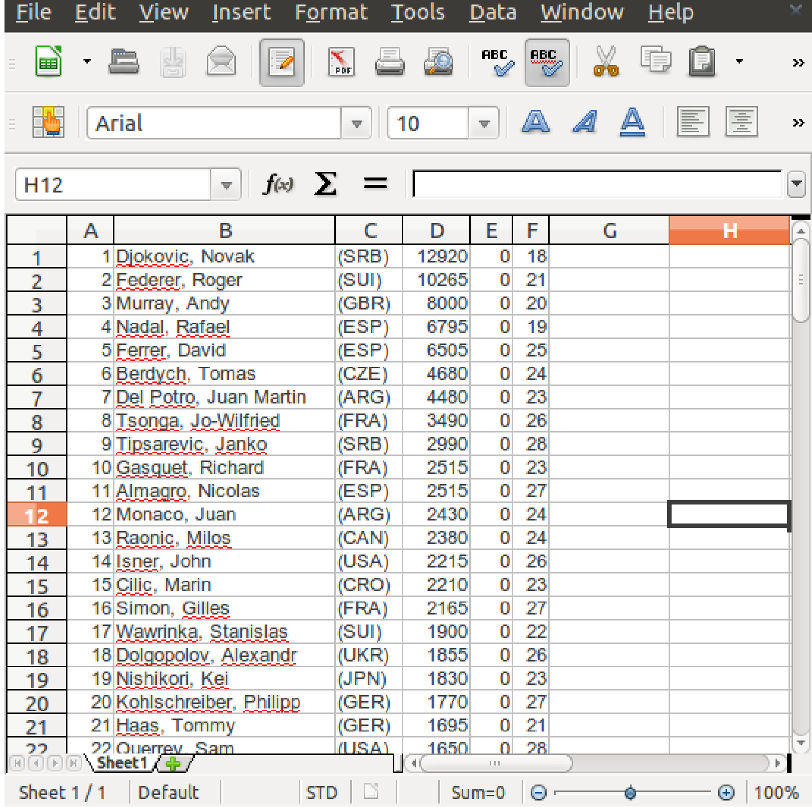
Line 16 begins our `if` branch. We are saying that if the cell has `class` of `first`, we want to do the stuff indented below Line 16. Note that we can't just check whether `thecell['class'] == 'first'`, because `thecell['class']` is a list, and we can't compare lists with strings. We can however, compare the first (zeroth) (and

only) element of this element.

Lines 17 through 20 extract the rank and name based on the structure of the HTML tags they're contained in. Line 21 is a regular expression of a complexity we've not really encountered before. What it says is, extract all sequences of three upper-case letters which are in curved brackets. Note that the brackets are preceded by slashes ( ) because brackets have a special meaning in regular expressions.

By the time we get to Line 22, and have written our results to the file, we're done with this special branch of the program. So we stick in an else statement, and below that put the same code we used to output results in our previous listing.

The output that I get from running, when opened in OpenOffice, looks a little bit like Figure 7.5.



	A	B	C	D	E	F	G	H
1	1	Djokovic, Novak	(SRB)	12920	0	18		
2	2	Federer, Roger	(SUI)	10265	0	21		
3	3	Murray, Andy	(GBR)	8000	0	20		
4	4	Nadal, Rafael	(ESP)	6795	0	19		
5	5	Ferrer, David	(ESP)	6505	0	25		
6	6	Berdych, Tomas	(CZE)	4680	0	24		
7	7	Del Potro, Juan Martin	(ARG)	4480	0	23		
8	8	Tsonga, Jo-Wilfried	(FRA)	3490	0	26		
9	9	Tipsarevic, Janko	(SRB)	2990	0	28		
10	10	Gasquet, Richard	(FRA)	2515	0	23		
11	11	Almagro, Nicolas	(ESP)	2515	0	27		
12	12	Monaco, Juan	(ARG)	2430	0	24		
13	13	Raonic, Milos	(CAN)	2380	0	24		
14	14	Isner, John	(USA)	2215	0	26		
15	15	Cilic, Marin	(CRO)	2210	0	23		
16	16	Simon, Gilles	(FRA)	2165	0	27		
17	17	Wawrinka, Stanislas	(SUI)	1900	0	22		
18	18	Dolgopolov, Alexandr	(UKR)	1855	0	26		
19	19	Nishikori, Kei	(JPN)	1830	0	23		
20	20	Kohlschreiber, Philipp	(GER)	1770	0	27		
21	21	Haas, Tommy	(GER)	1695	0	21		
22	22	Querrey, Sam	(USA)	1650	0	28		

Listing 7.5: ATP output

Extending the example

As with many of these examples, what we've just been able to accomplish is at the outer limits of what could be done manually. But now that we've been able to parse this particular ranking table, we are able to extend our work, and accomplish much more with just a few additional lines of code.

Looking at the ATP web page, you should notice that there are two drop down menus, which offer us the chance to look at rankings for different weeks, and rankings for different strata (1-100, 101-

200, 201-300, and so on...). We could adapt the code we've just written to scrape this information as well. In this instance, the key would come about not through replicating the actions we would go through in the browser (selecting each item, hitting on 'Go', copying the results), but on examining what happens to the address when we try and example change of week, or change of ranking strata. For example: if we select 101 - 200, you should see that the URL in your browser's address bar changes from `http://www.atpworldtour.com/Rankings/Singles.aspx` to `http://www.atpworldtour.com/Rankings/Singles.aspx?d=26.11.2012&r=101&c=#{}`. In fact, if we play around a bit, we can get to arbitrary start points by just adding something after `r` – we don't even need to include these `d=` and `c=` parameters. Try `http://www.atpworldtour.com/Rankings/Singles.aspx?r=314#{}` as an example.

We might therefore wrap most of the code from Listing 7.4 in a for loop of the form:

```
for strata in range(1,1501,100)
```

and then paste the variable `strata` on to our web address.

Extracting information week-by-week is a bit more difficult, since (a) dates are funny things compared to numbers, and (b) we don't know if all weeks are present. We would have to start by asking `BeautifulSoup` to get all values of the `<option>` tags used. Specifically, we'd create a blank list and append all values:

```
weekends = []
soup = BeautifulSoup(urlopen(start),parse_only =
    SoupStrainer('select',{'id':'singlesDates'}))
for option in soup.find_all('option'):
    weekends.append(option.get_text())
```

We could then create two nested `for` loops with `strata` and `weekends`, paste these on to the address we started with, and extract ourr table information.

8
Final notes

CONGRATULATIONS, YOU'VE GOT THIS FAR. You've learned how to understand the language that web pages are written in, and to take your first steps in a programming language called Python. You've written some simple programs that extract information from web pages, and turn them in to spreadsheets. I would estimate that those achievements place you in the top 0.5% of the population when it comes to digital literacy.

Whilst you've learned a great deal, the knowledge you have is quite shallow and brittle. From this booklet you will have learned a number of recipes that you can customize to fit your own needs. But pretty soon, you'll need to do some research on your own. You'll need independent knowledge to troubleshoot problems you have customizing these recipes – and for writing entirely new recipes.

If scraping the web is likely to be useful to you, you need to do the following.

First, you need to get a book on how to program Python. Most university libraries should have a couple of books, either in print or (more unwieldy) online through O'Reilly. An introductory text will give you a much fuller understanding. In particular, it will show you techniques that we haven't needed, but could have employed to make our code more robust or tidier.

Second, you need to get a grip on functions and regular expressions. We haven't really talked about functions here. We've used the functions that are built in to Python and to the several packages we've used, but we haven't rolled our own. Being able to write your own functions – or at least, to understand functions other people write – is very important. Very few of the scrapers available on ScraperWiki, for example, are written without using functions. Being able to wield regular expressions is also tremendously helpful. A lot of problems in life can be solved with regular expressions.[1]

[1] No, not really. But almost.

Third, you need to look at what other people are doing. Checking out some of the scrapers on ScraperWiki is invaluable. Look for some scrapers that use `BeautifulSoup`. Hack them. Break them.

Then look for other scrapers that are written in plain Python, or using other libraries[2] to parse the web. Hack them. Break them. Lather. Rinse. Repeat.

[2] Like `lxml`, one popular `BeautifulSoup` alternative.

Difficulties

I won't pretend that scraping the web is always easy. I have almost never written a program that worked the way I wanted to the first time I ran it. And there are some web pages that are difficult or impossible to scrape. Here's a list of some things you won't be able to do.

Facebook You can't scrape Facebook. It's against the terms and conditions, and Facebook is pretty good at detecting automated access from scrapers. When I first started on Facebook (late 2004/early 2005), I wrote a scraper to collect information on students' political views. I got banned for two weeks, and had to promise I wouldn't do the same again. Don't try. You can write to Facebook, and ask them to give you the data they hold on you.

Twitter You can get Twitter data, but not over the web. You'll need to use a different package,[3] and you'll need to get a *key* from Twitter. It's non-trivial.

[3] Say, `python-twitter` at https://github.com/bear/python-twitter

ASP.net pages You may have noticed that we stopped scraping links from AHRC news items after the first ten items. You might have wondered why we didn't go on to scrape the first twenty, first thirty, and so on.

The answer is that the AHRC's web pages are generated using ASP.net. ASP.net is a web application framework written by Microsoft. Many of the links in pages written using ASP.net aren't normal links. They're calls to Javascript programs which call ASP.net programs on the server. Only Microsoft could create a framework so inelegant that it basically breaks the one thing the web does well – link pages.

It is just possible to scrape ASP.net pages, but it takes a hell of a lot of work. See this ScraperWiki page for the gory details: http://blog.scraperwiki.com/2011/11/09/how-to-get-along-with-an-asp-webpage/.

Nexis It is technically possible to scrape Nexis results – all you need to do is to save your results in a big HTML file – but I'm almost certainly it violates the terms and conditions...