```
spans_inside_divs = [span
                     for div in soup('div')     # for each <div> on the page
                     for span in div('span')]   # find each <span> inside it
```

Just this handful of features will allow us to do quite a lot. If you end up needing to do more-complicated things (or if you're just curious), check the documentation.

Of course, whatever data is important won't typically be labeled as `class="important"`. You'll need to carefully inspect the source HTML, reason through your selection logic, and worry about edge cases to make sure your data is correct. Let's look at an example.

Example: O'Reilly Books About Data

A potential investor in DataSciencester thinks data is just a fad. To prove him wrong, you decide to examine how many data books O'Reilly has published over time. After digging through its website, you find that it has many pages of data books (and videos), reachable through 30-items-at-a-time directory pages with URLs like:

```
http://shop.oreilly.com/category/browse-subjects/data.do?
sortby=publicationDate&page=1
```

Unless you want to be a jerk (and unless you want your scraper to get banned), whenever you want to scrape data from a website you should first check to see if it has some sort of access policy. Looking at:

```
http://oreilly.com/terms/
```

there seems to be nothing prohibiting this project. In order to be good citizens, we should also check for a *robots.txt* file that tells webcrawlers how to behave. The important lines in *http://shop.oreilly.com/robots.txt* are:

```
Crawl-delay: 30
Request-rate: 1/30
```

The first tells us that we should wait 30 seconds between requests, the second that we should request only one page every 30 seconds. So basically they're two different ways of saying the same thing. (There are other lines that indicate directories not to scrape, but they don't include our URL, so we're OK there.)

There's always the possibility that O'Reilly will at some point revamp its website and break all the logic in this section. I will do what I can to prevent that, of course, but I don't have a ton of influence over there. Although, if every one of you were to convince everyone you know to buy a copy of this book…

To figure out how to extract the data, let's download one of those pages and feed it to Beautiful Soup:

```
# you don't have to split the url like this unless it needs to fit in a book
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

If you view the source of the page (in your browser, right-click and select "View source" or "View page source" or whatever option looks the most like that), you'll see that each book (or video) seems to be uniquely contained in a `<td>` table cell element whose `class` is `thumbtext`. Here is (an abridged version of) the relevant HTML for one book:

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014    </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                            Ebook:

                            <span class="price">&nbsp;$29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

A good first step is to find all of the `td thumbtext` tag elements:

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

Next we'd like to filter out the videos. (The would-be investor is only impressed by books.) If we inspect the HTML further, we see that each `td` contains one or more `span` elements whose `class` is `pricelabel`, and whose `text` looks like `Ebook:` or `Video:` or `Print:`. It appears that the videos contain only one `pricelabel`, whose `text` starts with `Video` (after removing leading spaces). This means we can test for videos with:

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if
```

```
        the stripped text inside that pricelabel starts with 'Video'"""
        pricelabels = td('span', 'pricelabel')
        return (len(pricelabels) == 1 and
                pricelabels[0].text.strip().startswith("Video"))

    print len([td for td in tds if not is_video(td)])
    # 21 for me, might be different for you
```

Now we're ready to start pulling data out of the `td` elements. It looks like the book title is the text inside the `<a>` tag inside the `<div class="thumbheader">`:

```
    title = td.find("div", "thumbheader").a.text
```

The author(s) are in the text of the `AuthorName` `<div>`. They are prefaced by a `By` (which we want to get rid of) and separated by commas (which we want to split out, after which we'll need to get rid of spaces):

```
    author_name = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

The ISBN seems to be contained in the link that's in the `thumbheader` `<div>`:

```
    isbn_link = td.find("div", "thumbheader").a.get("href")

    # re.match captures the part of the regex in parentheses
    isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

And the date is just the contents of the `<span class="directorydate">`:

```
    date = td.find("span", "directorydate").text.strip()
```

Let's put this all together into a function:

```
    def book_info(td):
        """given a BeautifulSoup <td> Tag representing a book,
        extract the book's details and return a dict"""

        title = td.find("div", "thumbheader").a.text
        by_author = td.find('div', 'AuthorName').text
        authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
        isbn_link = td.find("div", "thumbheader").a.get("href")
        isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
        date = td.find("span", "directorydate").text.strip()

        return {
            "title" : title,
            "authors" : authors,
            "isbn" : isbn,
            "date" : date
        }
```

And now we're ready to scrape:

```
    from bs4 import BeautifulSoup
    import requests
```

```
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # at the time of writing, probably more by now

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

    # now be a good citizen and respect the robots.txt!
    sleep(30)
```

Extracting data from HTML like this is more data art than data science. There are countless other find-the-books and find-the-title logics that would have worked just as well.

Now that we've collected the data, we can plot the number of books published each year (Figure 9-1):

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
    return int(book["date"].split()[1])

# 2014 is the last complete year of data (when I ran this)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("# of data books")
plt.title("Data is Big!")
plt.show()
```

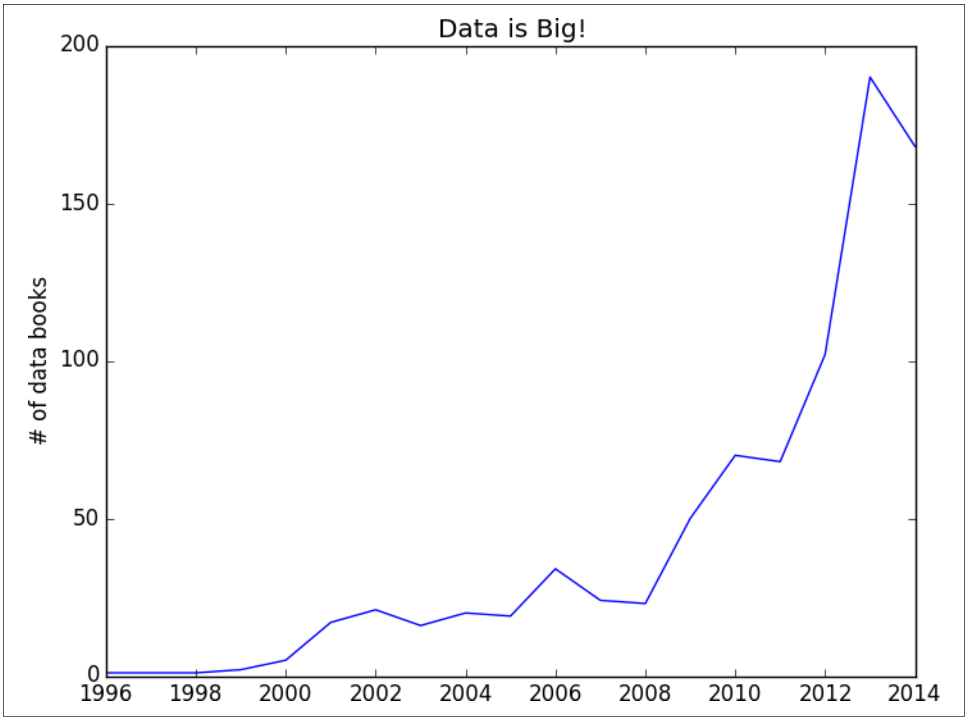

Figure 9-1. Number of data books per year

Unfortunately, the would-be investor looks at the graph and decides that 2013 was "peak data."

Using APIs

Many websites and web services provide application programming interfaces (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

JSON (and XML)

Because HTTP is a protocol for transferring *text*, the data you request through a web API needs to be *serialized* into a string format. Often this serialization uses JavaScript Object Notation (JSON). JavaScript objects look quite similar to Python `dicts`, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```