Web scraping the President's lies in 16 lines of Python

Created by Kevin Markham of [Data School](). Hosted on [GitHub]().

Summary

This an introductory tutorial on web scraping in Python. All that is required to follow along is a basic understanding of the Python programming language.

By the end of this tutorial, you will be able to scrape data from a static web page using the **requests** and **Beautiful Soup** libraries, and export that data into a structured text file using the **pandas** library.

Outline

What is web scraping?

On July 21, 2017, the New York Times updated an opinion article called [Trump's Lies](), detailing every public lie the President has told since taking office. Because this is a newspaper, the information was (of course) published as a block of text. This is a great format for human consumption, but it can't easily be understood by a computer. **In this tutorial, we'll extract the President's lies from the New York Times article and store them in a structured dataset.**

This is a common scenario: You find a web page that contains data you want to analyze, but it's not presented in a format that you can easily download and read into your favorite data analysis tool. You might imagine manually copying and pasting the data into a spreadsheet, but in most cases, that is way too time consuming. A technique called **web scraping** is a useful way to automate this process.

What is web scraping? It's the process of extracting information from a web page **by taking advantage of patterns** in the web page's underlying code. Let's start looking for these patterns!

Examining the New York Times article

Here's the way the article presented the information:

When converting this into a dataset, **you can think of each lie as a "record" with four fields:**

1. The date of the lie.
2. The lie itself (as a quotation)

2. The lie itself (as a quotation).
3. The writer's brief explanation of why it was a lie.
4. The URL of an article that substantiates the claim that it was a lie.

Importantly, those fields have different formatting, which is consistent throughout the article: the date is bold red text, the lie is "regular" text, the explanation is gray italics text, and the URL is linked from the gray italics text.

Why does the formatting matter? Because it's very likely that the code underlying the web page "tags" those fields differently, and we can take advantage of that pattern when scraping the page. Let's take a look at the source code, known as HTML:

Examining the HTML

To view the HTML code that generates a web page, you right click on it and select "View Page Source" in Chrome or Firefox, "View Source" in Internet Explorer, or "Show Page Source" in Safari. (If that option doesn't appear in Safari, just open Safari Preferences, select the Advanced tab, and check "Show Develop menu in menu bar".)

Here are the first few lines you will see if you view the source of the New York Times article:

Let's locate the **first lie** by searching the HTML for the text "iraq":

Thankfully, you only have to understand **three basic facts** about HTML in order to get started with web scraping!

Fact 1: HTML consists of tags

You can see that the HTML contains the article text, along with "tags" (specified using angle brackets) that "mark up" the text. ("HTML" stands for Hyper Text Markup Language.)

For example, one tag is `<strong>`, which means "use bold formatting". There is a `<strong>` tag before "Jan. 21" and a `</strong>` tag after it. The first is an "opening tag" and the second is a "closing tag" (denoted by the `/`), which indicates to the web browser **where to start and stop applying the formatting.** In other words, this tag tells the web browser to make the text "Jan. 21" bold. (Don't worry about the `&nbsp;` - we'll deal with that later.)

Fact 2: Tags can have attributes

HTML tags can have "attributes", which are specified in the opening tag. For example, `<span class="short-desc">` indicates that this particular `<span>` tag has a `class` attribute with a value of `short-desc`.

For the purpose of web scraping, **you don't actually need to understand** the meaning of `<span>`, `class`, or `short-desc`. Instead, you just need to recognize that tags can have attributes, and that they are specified in this particular way.

Fact 3: Tags can be nested

Let's pretend my HTML code said:

```
Hello <strong><em>Data School</em> students</strong>
```

The text **Data School students** would be bold, because all of that text is between the opening `<strong>` tag and the closing `</strong>` tag. The text ***Data School*** would also be in italics, because the `<em>` tag means "use italics". The text "Hello" would not be bold or italics, because it's not within either the `<strong>` or `<em>` tags. Thus, it would appear as follows:

Hello ***Data School*** **students**

The central point to take away from this example is that **tags "mark up" text from wherever they open to wherever they close,** regardless of whether they are nested within other tags.

Got it? You now know enough about HTML in order to start web scraping!

Reading the web page into Python

The first thing we need to do is to read the HTML for this article into Python, which we'll do using the requests library. (If you don't have it, you can `pip install requests` from the command line.)

In [1]:

```
import requests
r = requests.get('https://www.nytimes.com/interactive/2017/06/23/opinion/trumps-lies.html')
```

The code above fetches our web page from the URL, and stores the result in a "response" object called `r`. That response object has a `text` attribute, which contains the same HTML code we saw when viewing the source from our web browser:

In [2]:

```
# print the first 500 characters of the HTML
print(r.text[0:500])
```

```
<!DOCTYPE html>
<!--[if (gt IE 9)|!(IE)]> <!--><html lang="en" class="no-js page-interactive section-opinion page-
theme-standard tone-opinion page-interactive-default limit-small layout-xlarge app-interactive" it
emid="https://www.nytimes.com/interactive/2017/06/23/opinion/trumps-lies.html"
itemtype="http://schema.org/NewsArticle" itemscope
xmlns:og="http://opengraphprotocol.org/schema/"><!--<![endif]-->
<!--[if IE 9]> <html lang="en" class="no-js ie9 lt-ie10 page-interactive section-opinion page
```

Parsing the HTML using Beautiful Soup

We're going to parse the HTML using the Beautiful Soup 4 library, which is a popular Python library for web scraping. (If you don't have it, you can `pip install beautifulsoup4` from the command line.)

In [3]:

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(r.text, 'html.parser')
```

The code above parses the HTML (stored in `r.text`) into a special object called `soup` that the Beautiful Soup library understands. In other words, Beautiful Soup is **reading the HTML and making sense of its structure.**

(Note that `html.parser` is the parser included with the Python standard library, though other parsers can be used by Beautiful Soup. See differences between parsers to learn more.)

Collecting all of the records

The Python code above is the standard code I use with every web scraping project. Now, we're going to start **taking advantage of the patterns we noticed in the article formatting** to build our dataset!

Let's take another look at the article, and compare it with the HTML:

You might have noticed that each record has the following format:

`<span class="short-desc"><strong> DATE </strong> LIE <span class="short-truth"><a href="URL"> EXPLANATION </a></span></span>`

There's an outer `<span>` tag, and then nested within it is a `<strong>` tag plus another `<span>` tag, which itself contains an `<a>` tag. All of these tags affect the formatting of the text. And because the New York Times wants each record to appear in a consistent way in your web browser, we know that **each record will be tagged in a consistent way in the HTML.** This is the pattern that allows us to build our dataset!

Let's ask Beautiful Soup to **find all of the records:**

In [4]:

```
results = soup.find_all('span', attrs={'class':'short-desc'})
```

This code searches the `soup` object for all `<span>` tags with the attribute `class="short-desc"`. It returns a special Beautiful

This code searches the `soup` object for all `<span>` tags with the attribute `class="short-desc"`. It returns a special Beautiful Soup object (called a "ResultSet") containing the search results.

`results` acts like a **Python list**, so we can check its length:

In [6]:

```
len(results)
```

Out[6]:

```
180
```

There are 116 results, which seems reasonable given the length of the article. (If this number did not seem reasonable, we would examine the HTML further to determine if our assumptions about the patterns in the HTML were incorrect.)

We can also slice the object like a list, in order to examine the **first three results:**

In [7]:

```
results[0:3]
```

Out[7]:

```
[<span class="short-desc"><strong>Jan. 21 </strong>“I wasn't a fan of Iraq. I didn't want to go in
to Iraq.” <span class="short-truth"><a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-dona
ld-trump-said-he-supported-invading-iraq-on-the" target="_blank">(He was for an invasion before he
was against it.)</a></span></span>,
 <span class="short-desc"><strong>Jan. 21 </strong>“A reporter for Time magazine — and I have been
on their cover 14 or 15 times. I think we have the all-time record in the history of Time
magazine.” <span class="short-truth"><a href="http://nation.time.com/2013/11/06/10-things-you-didn
t-know-about-time/" target="_blank">(Trump was on the cover 11 times and Nixon appeared 55 times.)
</a></span></span>,
 <span class="short-desc"><strong>Jan. 23 </strong>“Between 3 million and 5 million illegal votes
caused me to lose the popular vote.” <span class="short-truth"><a
href="https://www.nytimes.com/2017/01/23/us/politics/donald-trump-congress-democrats.html"
target="_blank">(There's no evidence of illegal voting.)</a></span></span>]
```

We'll also check that the **last result** in this object matches the last record in the article:

In [8]:

```
results[-1]
```

Out[8]:

```
<span class="short-desc"><strong>Nov. 11 </strong>“I'd rather have him  – you know, work with him
on the Ukraine than standing and arguing about whether or not  – because that whole thing was set
up by the Democrats.” <span class="short-truth"><a
href="https://www.nytimes.com/interactive/2017/12/10/us/politics/trump-and-russia.html"
target="_blank">(There is no evidence that Democrats "set up" Russian interference in the
election.)</a></span></span>
```

Looks good!

We have now collected all 116 of the records, but we still need to **separate each record into its four components** (date, lie, explanation, and URL) in order to give the dataset some structure.

Extracting the date

Web scraping is often an iterative process, in which you experiment with your code until it works exactly as you desire. To simplify the experimentation, we'll start by only working with the **first record** in the `results` object, and then later on we'll modify our code to use a loop:

In [9]:

```
first_result = results[0]
```

```
first_result
```

Out[9]:

```
<span class="short-desc"><strong>Jan. 21 </strong>“I wasn't a fan of Iraq. I didn't want to go int
o Iraq.” <span class="short-truth"><a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-
donald-trump-said-he-supported-invading-iraq-on-the" target="_blank">(He was for an invasion befor
e he was against it.)</a></span></span>
```

Although `first_result` may look like a Python string, you'll notice that there are no quote marks around it. Instead, it's another special Beautiful Soup object (called a "Tag") that has specific methods and attributes.

In order to locate the date, we can use its `find()` method to **find a single tag** that matches a specific pattern, in contrast to the `find_all()` method we used above to **find all tags** that match a pattern:

In [10]:

```
first_result.find('strong')
```

Out[10]:

```
<strong>Jan. 21 </strong>
```

This code searches `first_result` for the first instance of a `<strong>` tag, and again returns a Beautiful Soup "Tag" object (not a string).

Since we want to **extract the text between the opening and closing tags**, we can access its `text` attribute, which does in fact return a regular Python string:

In [11]:

```
first_result.find('strong').text
```

Out[11]:

```
'Jan. 21\xa0'
```

What is `\xa0`? You don't actually need to know this, but it's called an "escape sequence" that represents the `&nbsp;` character we saw earlier in the HTML source.

However, you do need to know that **an escape sequence represents a single character** within a string. Let's slice it off from the end of the string:

In [12]:

```
first_result.find('strong').text[0:-1]
```

Out[12]:

```
'Jan. 21'
```

Finally, we're going to add the year, since we don't want our dataset to include ambiguous dates:

In [13]:

```
first_result.find('strong').text[0:-1] + ', 2017'
```

Out[13]:

```
'Jan. 21, 2017'
```

Extracting the lie

Let's take another look at `first_result`:

In [14]:

```
first_result
```

Out[14]:

```
<span class="short-desc"><strong>Jan. 21 </strong>“I wasn't a fan of Iraq. I didn't want to go int
o Iraq.” <span class="short-truth"><a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-
donald-trump-said-he-supported-invading-iraq-on-the" target="_blank">(He was for an invasion befor
e he was against it.)</a></span></span>
```

Our goal is to extract the two sentences about Iraq. Unfortunately, there isn't a pair of opening and closing tags that starts **immediately before the lie** and ends **immediately after the lie**. Therefore, we're going to have to use a different technique:

In [15]:

```
first_result.contents
```

Out[15]:

```
[<strong>Jan. 21 </strong>,
 "“I wasn't a fan of Iraq. I didn't want to go into Iraq.” ",
 <span class="short-truth"><a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-
said-he-supported-invading-iraq-on-the" target="_blank">(He was for an invasion before he was agai
nst it.)</a></span>]
```

The `first_result` "Tag" has a `contents` attribute, which returns a Python list containing its "children". What are children? They are the **Tags and strings that are nested within a Tag.**

We can slice this list to extract the second element:

In [16]:

```
first_result.contents[1]
```

Out[16]:

```
"“I wasn't a fan of Iraq. I didn't want to go into Iraq.” "
```

Finally, we'll slice off the curly quotation marks as well as the extra space at the end:

In [17]:

```
first_result.contents[1][1:-2]
```

Out[17]:

```
"I wasn't a fan of Iraq. I didn't want to go into Iraq."
```

Extracting the explanation

Based upon what you've seen already, you might have figured out that we have at least **two options** for how we extract the third component of the record, which is the writer's explanation of why the President's statement was a lie.

The **first option** is to slice the `contents` attribute, like we did when extracting the lie:

In [18]:

```
first_result.contents[2]
```

Out[18]:

```
<span class="short-truth"><a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-s
aid-he-supported-invading-iraq-on-the" target="_blank">(He was for an invasion before he was again
st it.)</a></span>
```

The **second option** is to search for the surrounding tag, like we did when extracting the date:

In [19]:

```
first_result.find('a')
```

Out[19]:

```
<a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-said-he-supported-invading-
iraq-on-the" target="_blank">(He was for an invasion before he was against it.)</a>
```

Either way, we can access the `text` attribute and then slice off the opening and closing parentheses:

In [20]:

```
first_result.find('a').text[1:-1]
```

Out[20]:

```
'He was for an invasion before he was against it.'
```

Extracting the URL

Finally, we want to extract the URL of the article that substantiates the writer's claim that the President was lying.

Let's examine the `<a>` tag within `first_result`:

In [21]:

```
first_result.find('a')
```

Out[21]:

```
<a href="https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-said-he-supported-invading-
iraq-on-the" target="_blank">(He was for an invasion before he was against it.)</a>
```

So far in this tutorial, we have been extracting text that is **between tags**. In this case, the text we want to extract is located **within the tag itself**. Specifically, we want to access the value of the `href` attribute within the `<a>` tag.

Beautiful Soup treats tag attributes and their values like **key-value pairs in a dictionary:** you put the attribute name in brackets (like a dictionary key), and you get back the attribute's value:

In [22]:

```
first_result.find('a')['href']
```

Out[22]:

```
'https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-said-he-supported-invading-iraq-on-
the'
```

Recap: Beautiful Soup methods and attributes

Before we finish building the dataset, I want to summarize a few ways you can interact with Beautiful Soup objects.

You can apply these **two methods** to either the initial `soup` object or a Tag object (such as `first_result`):

- `find()`: searches for the first matching tag, and returns a Tag object
- `find_all()`: searches for all matching tags, and returns a ResultSet object (which you can treat like a list of Tags)

You can extract information from a Tag object (such as `first_result`) using these **two attributes:**

- `text`: extracts the text of a Tag, and returns a string
- `contents`: extracts the children of a Tag, and returns a list of Tags and strings

It's important to keep track of whether you are interacting with a Tag, ResultSet, list, or string, because that affects which methods and attributes you can access.

And of course, there are many more methods and attributes available to you, which are described in the Beautiful Soup documentation.

Building the dataset

Now that we've figured out how to extract the four components of `first_result`, we can **create a loop to repeat this process** on all 116 `results`. We'll store the output in a **list of tuples** called `records`:

In [23]:

```
records = []
for result in results:
    date = result.find('strong').text[0:-1] + ', 2017'
    lie = result.contents[1][1:-2]
    explanation = result.find('a').text[1:-1]
    url = result.find('a')['href']
    records.append((date, lie, explanation, url))
```

Since there were 116 `results`, we should have 116 `records`:

In [24]:

```
len(records)
```

Out[24]:

```
180
```

Let's do a quick spot check of the first three records:

In [25]:

```
records[0:3]
```

Out[25]:

```
[('Jan. 21, 2017',
  "I wasn't a fan of Iraq. I didn't want to go into Iraq.",
  'He was for an invasion before he was against it.',
  'https://www.buzzfeed.com/andrewkaczynski/in-2002-donald-trump-said-he-supported-invading-iraq-o
n-the'),
 ('Jan. 21, 2017',
  'A reporter for Time magazine — and I have been on their cover 14 or 15 times. I think we have t
he all-time record in the history of Time magazine.',
  'Trump was on the cover 11 times and Nixon appeared 55 times.',
  'http://nation.time.com/2013/11/06/10-things-you-didnt-know-about-time/'),
 ('Jan. 23, 2017',
  'Between 3 million and 5 million illegal votes caused me to lose the popular vote.',
  "There's no evidence of illegal voting.",
  'https://www.nytimes.com/2017/01/23/us/politics/donald-trump-congress-democrats.html')]
```

Looks good!

Applying a tabular data structure

The last major step in this process is to apply a tabular data structure to our existing structure (which is a list of tuples). We're going to do this using the pandas library, an incredibly popular Python library for data analysis and manipulation. (If you don't have it, here are the installation instructions.)

The primary data structure in pandas is the "DataFrame", which is suitable for tabular data with columns of different types, **similar to an Excel spreadsheet or SQL table.** We can convert our list of tuples into a DataFrame by passing it to the DataFrame constructor and specifying the desired column names:

In [26]:

```
import pandas as pd
df = pd.DataFrame(records, columns=['date', 'lie', 'explanation', 'url'])
```

The DataFrame includes a `head()` method, which allows you to examine the top of the DataFrame:

In [27]:

```
df.head()
```

Out[27]:

	date	lie	explanation	url
0	Jan. 21, 2017	I wasn't a fan of Iraq. I didn't want to go in...	He was for an invasion before he was against it.	https://www.buzzfeed.com/andrewkaczynski/in-20...
1	Jan. 21, 2017	A reporter for Time magazine — and I have been...	Trump was on the cover 11 times and Nixon appe...	http://nation.time.com/2013/11/06/10-things-yo...
2	Jan. 23, 2017	Between 3 million and 5 million illegal votes ...	There's no evidence of illegal voting.	https://www.nytimes.com/2017/01/23/us/politics...
3	Jan. 25, 2017	Now, the audience was the biggest ever. But th...	Official aerial photos show Obama's 2009 inaug...	https://www.nytimes.com/2017/01/21/us/politics...
4	Jan. 25, 2017	Take a look at the Pew reports (which show vot...	The report never mentioned voter fraud.	https://www.nytimes.com/2017/01/24/us/politics...

The numbers on the left side of the DataFrame are known as the "index", which act as identifiers for the rows. Because we didn't specify an index, it was automatically assigned as the integers 0 to 115.

We can examine the bottom of the DataFrame using the `tail()` method:

In [28]:

```
df.tail()
```

Out[28]:

	date	lie	explanation	url
175	Oct. 25, 2017	We have trade deficits with almost everybody.	We have trade surpluses with more than 100 cou...	https://www.bea.gov/newsreleases/international...
176	Oct. 27, 2017	Wacky & totally unhinged Tom Steyer, who has b...	Steyer has financially supported many winning ...	https://www.opensecrets.org/donor-lookup/resul...
177	Nov. 1, 2017	Again, we're the highest-taxed nation, just ab...	We're not.	http://www.politifact.com/truth-o-meter/statem...
178	Nov. 7, 2017	When you look at the city with the strongest g...	Several other cities, including New York and L...	http://www.politifact.com/truth-o-meter/statem...
179	Nov. 11, 2017	I'd rather have him – you know, work with him...	There is no evidence that Democrats "set up" R...	https://www.nytimes.com/interactive/2017/12/10...

Did you notice that "January" is abbreviated, while "July" is not? It's best to format your data consistently, and so we're going to convert the date column to pandas' special "datetime" format:

In [29]:

```
df['date'] = pd.to_datetime(df['date'])
```

The code above converts the "date" column to datetime format, and then overwrites the existing "date" column. (Notice that we did not have to tell pandas that the column was originally in "MONTH DAY, YEAR" format - **pandas just figured it out!**)

Let's take a look at the results:

In [30]:

```
df.head()
```

Out[30]:

	date	lie	explanation	url
0	2017-01-21	I wasn't a fan of Iraq. I didn't want to go in...	He was for an invasion before he was against it.	https://www.buzzfeed.com/andrewkaczynski/in-20...
1	2017-01-21	A reporter for Time magazine — and I have been...	Trump was on the cover 11 times and Nixon appe...	http://nation.time.com/2013/11/06/10-things-yo...
2	2017-01-23	Between 3 million and 5 million illegal votes ...	There's no evidence of illegal voting.	https://www.nytimes.com/2017/01/23/us/politics...
3	2017-01-25	Now, the audience was the biggest ever. But th...	Official aerial photos show Obama's 2009 inaug...	https://www.nytimes.com/2017/01/21/us/politics...
4	2017-01-25	Take a look at the Pew reports (which show vot...	The report never mentioned voter fraud.	https://www.nytimes.com/2017/01/24/us/politics...

In [31]:

```
df.tail()
```

Out[31]:

	date	lie	explanation	url
175	2017-10-25	We have trade deficits with almost everybody.	We have trade surpluses with more than 100 cou...	https://www.bea.gov/newsreleases/international...
176	2017-10-27	Wacky & totally unhinged Tom Steyer, who has b...	Steyer has financially supported many winning ...	https://www.opensecrets.org/donor-lookup/resul...
177	2017-11-01	Again, we're the highest-taxed nation, just ab...	We're not.	http://www.politifact.com/truth-o-meter/statem...
178	2017-11-07	When you look at the city with the strongest g...	Several other cities, including New York and L...	http://www.politifact.com/truth-o-meter/statem...
179	2017-11-11	I'd rather have him – you know, work with him...	There is no evidence that Democrats "set up" R...	https://www.nytimes.com/interactive/2017/12/10...

Not only is the date column now consistently formatted, but pandas also provides a wealth of date-related functionality because it's in datetime format.

Exporting the dataset to a CSV file

Finally, we'll use pandas to export the DataFrame to a CSV (comma-separated value) file, which is the simplest and most common way to **store tabular data in a text file:**

In [32]:

```
df.to_csv('trump_lies.csv', index=False, encoding='utf-8')
```

We set the `index` parameter to `False` to tell pandas that we don't need it to include the index (the integers 0 to 115) in the CSV file. You should be able to find this file in your working directory, and open it in any text editor or spreadsheet program!

In the future, you can rebuild this DataFrame by reading the CSV file back into pandas:

In [33]:

```
df = pd.read_csv('trump_lies.csv', parse_dates=['date'], encoding='utf-8')
```

If you want to learn a lot more about the pandas library, you can watch my video series, Easier data analysis in Python with pandas , or check out my top 8 resources for learning pandas.

Summary: 16 lines of Python code

Here are the 16 lines of code that we used to scrape the web page, extract the relevant data, convert it into a tabular dataset, and export it to a CSV file:

In [34]:

```
import requests
r = requests.get('https://www.nytimes.com/interactive/2017/06/23/opinion/trumps-lies.html')

from bs4 import BeautifulSoup
soup = BeautifulSoup(r.text, 'html.parser')
results = soup.find_all('span', attrs={'class':'short-desc'})

records = []
for result in results:
    date = result.find('strong').text[0:-1] + ', 2017'
    lie = result.contents[1][1:-2]
    explanation = result.find('a').text[1:-1]
    url = result.find('a')['href']
    records.append((date, lie, explanation, url))

import pandas as pd
df = pd.DataFrame(records, columns=['date', 'lie', 'explanation', 'url'])
df['date'] = pd.to_datetime(df['date'])
df.to_csv('trump_lies.csv', index=False, encoding='utf-8')
```

Appendix A: Web scraping advice

- Web scraping works best with **static, well-structured web pages**. Dynamic or interactive content on a web page is often not accessible through the HTML source, which makes scraping it much harder!
- Web scraping is a "fragile" approach for building a dataset. The HTML on a page you are scraping can **change at any time**, which may cause your scraper to stop working.
- If you can **download the data** you need from a website, or if the website provides an **API with data access**, those approaches are preferable to scraping since they are easier to implement and less likely to break.
- If you are **scraping a lot of pages** from the same website (in rapid succession), it's best to insert delays in your code so that you don't overwhelm the website with requests. If the website decides you are causing a problem, they can block your IP address (which may affect everyone in your building!)
- Before scraping a website, you should review its **robots.txt file** (also known as the Robots exclusion standard) to check whether you are "allowed" to scrape their website. (Here is the robots.txt file for nytimes.com.)

Appendix B: Web scraping resources

- The Beautiful Soup documentation is written like a tutorial, and is worth reading to gain a detailed understanding of the library.
- For more Beautiful Soup examples, see Web Scraping 101 with Python, More web scraping with Python, and this web scraping lesson from Stanford's "Text As Data" course.
- Web Scraping with Python is a 3-hour video tutorial covering Beautiful Soup and other scraping tools. (The slides and code are also available.)
- Scrapy is a popular application framework that is useful for more complex web scraping projects.
- How a Math Genius Hacked OkCupid to Find True Love and How Netflix Reverse Engineered Hollywood are two fun examples of using web scraping to build an interesting dataset.

Appendix C: Alternative syntax for Beautiful Soup

It's worth noting that Beautiful Soup actually offers multiple ways to express the same command. I tend to use the most verbose option, since I think it makes the code readable, but it's useful to be able to recognize the alternative syntax since you might see it used elsewhere.

For example, you can **search for a tag** by accessing it like an attribute:

In [34]:

```
# search for a tag by name
first_result.find('strong')

# shorter alternative: access it like an attribute
first_result.strong
```

Out[34]:

```
<strong>Jan. 21 </strong>
```

You can also **search for multiple tags** a few different ways:

In [35]:

```python
# search for multiple tags by name and attribute
results = soup.find_all('span', attrs={'class':'short-desc'})

# shorter alternative: if you don't specify a method, it's assumed to be find_all()
results = soup('span', attrs={'class':'short-desc'})

# even shorter alternative: you can specify the attribute as if it's a parameter
results = soup('span', class_='short-desc')
```

For more details, check out the Beautiful Soup documentation.