Web Scraping and Automatic Data Collection

16

by **Patrick Boily**, with contributions from **Andrew Macfie**

Data analysis tools and techniques work in conjunction with collected data. The type of data that needs to be collected to carry out such analyses, as well as the priority placed on the collection of quality data relative to other demands, dictate the choice of data collection strategies. The manner in which the resulting outputs of these analyses are used for decision support will, in turn, influence appropriate data presentation strategies and system functionality.

We have already discussed how data can be processed and transformed to make it more suitable for analysis (see Section 15), and how questionnaire design and probabilistic sampling can be used to obtain representative datasets (see Section 10); in this chapter we explore the technical aspects of automated data collection and web scraping, as well as the many ways in which this activity can go awry.*

16.1   Data Analysis and Web Scraping

Although analysts should always endeavour to work with **representative** and **unbiased data**, there will be times when the available data is flawed and not easily repaired.

We have a professional responsibility to explore the data, looking for potential fatal flaws **prior** to the start of the analysis and to inform clients and stakeholders of any findings that could halt, skew, or simply hinder the analytical process or its applicability to the situation at hand.[1]

We might also be called upon to provide suggestions to evaluate or fix the **data collection system**. The following items could help with that.

Data validity: the system must collect the data in such a way that data validity is ensured during initial collection. In particular, data must be collected in a way that ensures sufficient accuracy and precision of the data, relative to its intended use.

Data granularity, scale of data: the system must collect the data at a level of granularity appropriate for future analysis.

Data coverage: the system must collect data that comprehensively, rather than only partially or unevenly, represents the objects of interest; the system must collect and store the required data over a sufficient amount of time, and at the required intervals, to support data analyses that require data spanning a certain duration;

1: It is **EXTREMELY IMPORTANT** that these flaws not simply be swept under the carpet; they need to be addressed, and the analysis outcomes that result must be presented or reported on with an appropriate *caveat*.

---

* Some of the material is modified, in part, from [6, 5].

Data storage: the system must have the functionality to store the types and amount of data required for a particular analysis.

Data accessibility: the system must provide access to the data relevant for a particular analysis, in a format that is appropriate for this analysis.

Computational/analytic functionality: the system must have the ability to carry out the computations required by relevant data analysis techniques.

Reporting, dashboard, visualization: the system must be able to present the results of the data analysis in a meaningful, usable and responsive fashion.

A number of different overarching strategies for data collection can be employed. Each of these different strategies will be more or less appropriate under certain data collection circumstances, and will result in different system **functional requirements**.

This is partly why analysts must take the time to **understand their systems** before embarking on data analysis (see Chapter 14, *Data Science Basics* for details).

World Wide Web

It has been said that the "streets of the Web are paved with data just waiting to be collected" [6], but you might be surprised to discover how much of that data is "trash" [Boily].

The way we **share**, **collect**, and **publish** data has changed over the past few years due to the ubiquity of the *World Wide Web*. **Private businesses**, **governments**, and **individual users** are posting and sharing all kinds of data and information. At every moment, new channels generate vast amounts of data.

There was a time in the recent past where both scarcity and inaccessibility of data was a problem for researchers and decision-makers. That is **emphatically** not the case anymore.

But **data abundance** carries its own set of problems, in the form of:

- tangled masses of data, and
- traditional data collection methods and classical data analysis techniques not being up to the task anymore.[2]

2: Which is not to say that the results they would give would be incorrect; it's rather their lack of efficiency that comes into play.

The growth and increasing popularity and power of **open source software**, such as R and Python, for which the source code can be inspected, modified, and enhanced by anyone, makes program-based automated data collection quite appealing.

One note of warning, however: time marches on and packages become **obsolete** in the blink of an eye. If the analyst is unable (or unwilling) to **maintain their extraction/analysis code** and to **monitor the sites** from which the data is extracted, the choice of software will not make much of a difference.

16.1.1 The What and Why of Web Scraping

So why bother with **automated data collection**? Common considerations include:

- the sparsity of financial resources;
- the lack of time or desire to collect data manually;
- the desire to work with up-to-date, high-quality data-rich sources, and
- the need to document the analytical process from beginning (data collection) to end (publication).

Manual collection, on the other hand, tends to be cumbersome and prone to error; non-reproducible processes are also subject to heightened risks of "death by boredom", whereas program-based solutions are typically more reliable, reproducible, time-efficient, and produce datasets of higher quality (this assumes, of course, that coherently presented data exists in the first place).

Automated Data Checklist

That being said, **web scraping** is not always recommended. As a starting point, it is possible that no online and freely available source of data meets the analysis' needs, in which case an approach based on survey sampling is preferable, in all likelihood.

If most of the answers to the following questions are positive, however, then an automated approach may be the right choice:

- is there a need to repeat the task from time to time?[3]
- is there a need for others to to replicate the data collection process?
- are online sources of data frequently used?
- is the task non-trivial in terms of scope and complexity?
- if the task can be done manually, are the financial resources required to let others do the work lacking?
- is the will to automate the process by means of programming there?

3: E.g., to update a database, say.

The objective is simple: automatic data collection should yield a collection of unstructured or unsorted datasets, at a reasonable cost.

16.1.2 Web Data Quality

Data quality issues are inescapable. It is not rare for stakeholders or clients to have spent thousands of dollars on data collection (automatic or manual) and to respond to the news that the data is flawed or otherwise unusable with: "well, it's the best data we have, so find a way to use it."

These issues can be side-stepped to some extent if consultants get involved in the project during or prior to the data collection stage, asking questions such as:

- what **type of data** is best-suited to answer the client's question(s)?
- is the available data of **sufficiently high quality** to answer the client's question(s)?
- is the available information **systematically flawed**?

Web data can be **first-hand** information (a tweet or a news article), or **second-hand** (copied from an offline source or scraped from some online location, which may make it difficult to retrace).

Cross-referencing is a standard practice when dealing with secondary data. Data quality also depends on its **use(s)** and **purpose(s)**. For example, a sample of tweets collected on a random day could be used to analyse the use of a hashtags or the gender-specific use of words, but that dataset might not prove as useful if it had been collected on the day of the 2016 U.S. Presidential Election to predict the election outcomes.[4]

4: Due to **collection bias**.

Example   Say a client is interested in using a standard telephone survey to find out what people think of a new potato peeler. Such an approach has a number of pitfalls:

- **unrepresentative sample** – the selected sample might not represent the intended population;
- **systematic non-response** – people who do not like phone surveys might be less (or more) likely to dislike the new potato peeler;
- **coverage error** – people without a landline cannot be reached, say, and
- **measurement error** – are the survey questions providing suitable info for the problem at hand?

Traditional solutions to these require the use of **survey sampling**, **questionnaire design**, **omnibus surveys**, reward systems, audits, etc.[5]

5: See Chapter 10 for a discussion of the first two items.

These solutions can be **costly**, **time-consuming**, and **ineffective**. **Proxies** – indicators that are strongly related to the product's popularity without measuring it directly, could be used instead.

If **popularity** is defined as large groups of people preferring a potato peeler over another one, then sales statistics on a commercial website may provide a proxy for popularity. Rankings on 'Amazon.ca' ⬀ (or a similar website) could, in fact, paint a more comprehensive portrait of the potato peeler market than would a traditional survey.

It could suffice, then, to build a scraper that is compatible with Amazon's **application program interface** (API) to gather the appropriate data. Of course, there are potential issues with this approach as well:

- **representativeness** of the **listed products** – are all potato peelers listed? If not, is it because that website does not sell them or is there some other reason?
- **representativeness** of the **customers** – are there specific groups buying/not-buying online products? Are there specific groups buying from specific sites? Are there specific groups leaving/not-leaving reviews?
- **truthfulness** of customers and **reliability** of reviews – how can we distinguish between paid (fake) reviews and real reviews?

Web scraping is usually well-suited for collecting data on products (such as the aforementioned potato-peeler), but there are numerous questions for which it is substantially more difficult to imagine where data could be found online: what data could be collected online to measure the popularity of a government policy, say?

16.1.3 Ethical Considerations

So is all the data on the Internet ACTUALLY "freely" available?

A **spider** is a program that grazes or crawls the web rapidly, looking for information. It jumps from one page to another, grabbing the entire page content. **Scraping**, on the other hand, is defined as taking specific information from specific websites: how are these different?

> "Scraping inherently involves **copying**, and therefore one of the most obvious claims against scrapers is copyright infringement." [6]

What can be done to minimize the risk? Analysts should:

- work as **transparently** as possible;
- **document** data sources at all time;
- **give credit** to those who originally collected/published the data;
- keep in mind that if someone else collected the data, **permission is probably required** to reproduce it, and, more importantly,
- **not do anything illegal**.

A number of cases have shown that the courts have not yet found their footing in this matter – see *eBay* vs. *Bidder's Edge*, *Associated Press* vs. *Meltwater*, *Facebook* vs. *Pete Warden*, *United States* vs. *Aaron Swartz*, for instance [5].

There are legal issues that we are not qualified to discuss, but in general, it seems as though larger companies/organisations usually emerge victorious from such battles.

Part of the difficulty is that it is not clear which scraping actions are illegal and which are legal, but there are rough guidelines: re-publishing content for commercial purposes is considered more problematic than downloading pages for research/analysis, say.

A site's `robots.txt` (Robots Exclusion Protocol) file tells scrapers what information on the site may be harvested with the publisher's consent – analysts must heed that file (see Figure 16.1 for examples of such files).

Figure 16.1: The *Robots Exclusion Protocol* file for cqads.carleton.ca ⬈ , theweathernetwork.com ⬈ , cfl.ca ⬈ (as of Dec 2022).

Perhaps more importantly, **be friendly!** Not everything that can be scraped needs to be scraped. Scraping programs should
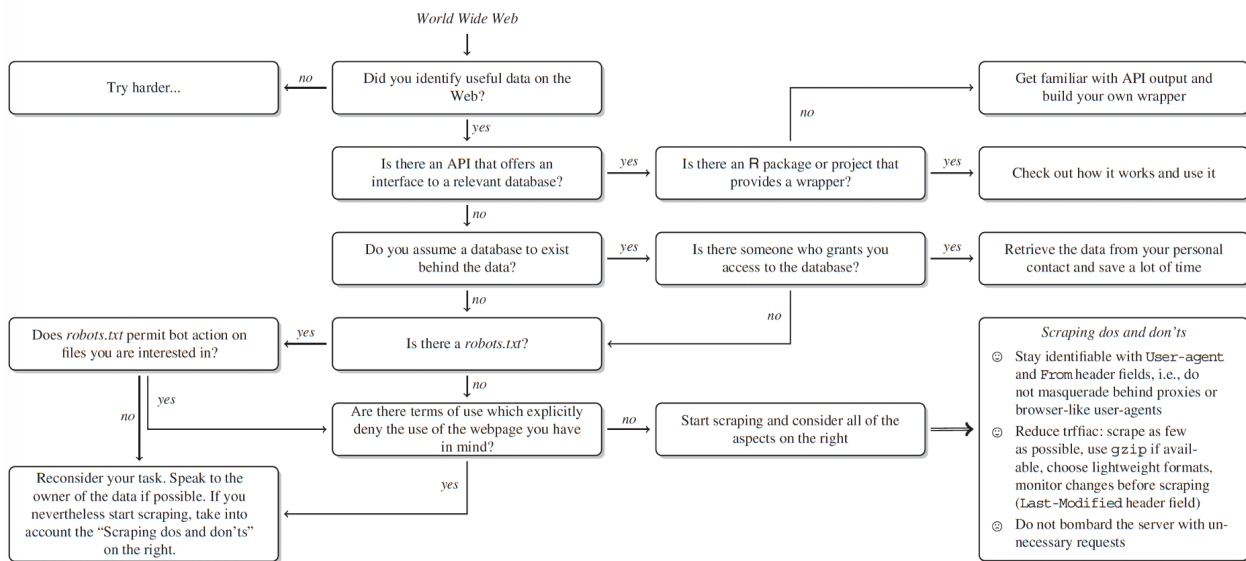
Figure 16.2: Etiquette flow diagram for web scraping. [6]

1. behave "nicely";
2. provide useful data, and
3. be efficient, in that order.

Any data accessed by HTTP forms is stored in some sort of database. When in doubt, contact the data provider to see if they will grant access to the databases or files.

The larger the amount of data required, the better it is for both parties to communicate before starting to harvest data – for "small" amounts of data, that may be less important, but small *for someone* does not necessarily mean small *for all*.

Finally, note the importance of following the **Scraping Do's and Don't's**:

1. **stay identifiable**;
2. **reduce traffic** – accept compressed files, check that a file has been changed before accessing it again, retrieve only parts of a file;
3. **do not bother server with multiple requests** – many requests per second can bring smaller server downs, webmasters may block a scraper if it is too greedy (a few requests per second is fine), and
4. **write efficient and polite scrapers** – there is no reason to scrape pages daily or to repeat the same task over and over, select specific resources and leave the rest untouched.

6: Really quickly and really often, in fact.

The **design of webpages** tends to change quickly and often.[6] A broken scraper will still consume bandwidth, however, without payoff; scraper **maintenance** is paramount to successful data collection.

This is all put together in an etiquette flow diagram (or perhaps that should be "ethiquette"?) provided by [6] (see Figure 16.2).

16.1.4 Automated Data Collection Decision Process

Let us end this section by providing a short summary of the **automated data collection decision process** [6, 5], from the point of view of analysts or quantitative consultants:

1. **Know exactly what kind of information the client needs**, either **specific** (e.g. GDP of all OECD countries for last 10 years, sales of top 10 tea brands in 2017, etc.) or **vague** (people's opinion on tea brand X, etc.)

2. **Find out if there are any web data sources that could provide direct or indirect information on the client's problem.** That is easier to achieve for specific facts (a tea store's webpage will provide information about teas that are currently in demand) than it is for vague facts (where would one find opinions on a collection of tea brands?). Tweets and social media platforms may contain opinion trends; commercial platforms can provide information on product satisfaction.

3. **Develop a theory of the data generation process when looking into potential data sources.** When was the data generated? When was it uploaded to the Web? Who uploaded the data? Are there any potential areas that are not covered, consistent, or accurate? How often is the data updated?

4. **Balance the advantages and disadvantages of potential data sources.** Validate the quality of data used – are there other independent sources that provide similar information against which to crosscheck? Can original source of secondary data be identified?

5. **Make a data collection decision.** Choose the data sources that seem most suitable, and document reasons for this decision. Collect data from several sources to validate the final choice.

16.2 Web Technologies Basics

Online data can be found in **text**, **tables**, **lists**, **links**, and other structures, but the way data is presented in browsers is not necessarily how it is stored in HTML/XML.

For instance, consider the NHL's Atlantic Division standings on 20-Mar-2018 below.

Atlantic	GP	W	L	OT	PTS	ROW	GF	GA	DIFF	HOME	AWAY	S/O	L10	STRK	Last Game	Next Game
x-Tampa Bay	72	49	19	4	102	43	260	202	+58	26-8-2	23-11-2	6-2	7-2-1	W1	Mar 18: TBL 3 - EDM 1	Mar 20 vs TOR
Boston	71	45	17	9	99	42	239	184	+55	25-7-5	20-10-4	3-2	7-2-1	OT1	Mar 19: BOS 4 - CBJ 5	Mar 21 @ STL
Toronto	72	43	22	7	93	36	243	204	+39	25-8-2	18-14-5	7-2	6-2-2	W4	Mar 17: TOR 4 - MTL 0	Mar 20 @ TBL
Florida	70	36	27	7	79	33	212	216	-4	22-11-3	14-16-4	3-3	7-2-1	W1	Mar 19: FLA 2 - MTL 0	Mar 20 @ OTT
Montréal	73	26	35	12	64	24	182	232	-50	17-12-8	9-23-4	2-6	2-6-2	L3	Mar 19: MTL 0 - FLA 2	Mar 21 @ PIT
Ottawa	71	26	34	11	63	24	197	244	-47	15-14-6	11-20-5	2-7	5-4-1	L1	Mar 17: OTT 1 - CBJ 2	Mar 20 vs FLA
Detroit	72	26	35	11	63	22	184	224	-40	13-14-8	13-21-3	4-1	0-9-1	L6	Mar 18: DET 1 - COL 5	Mar 20 vs PHI
Buffalo	72	23	37	12	58	22	172	236	-64	11-21-5	12-16-7	1-2	5-4-1	L1	Mar 19: BUF 0 - NSH 4	Mar 21 vs ARI

Figure 16.3: NHL's Atlantic Division standings on 20-Mar-2018 [nhl.com ↗]

This table is human-readable: most people familiar with professional competitions can recognize what it "means", even if they know very little about hockey or the National Hockey League.

In a **web browser**, this is not how the information is found, however (see Figure 16.4).

Figure 16.4: NHL's Atlantic Division standings on 20-Mar-2018 (under the hood) [nhl.com ⟲]

Furthermore, when web pages are **dynamic**, there is a "cost" associated with automated collection. Consequently, a basic knowledge of the web and web-related techs and documents is crucial.ˆ[Information can readily be found online and in [5, 6].}

There are three areas of importance for data collection on the web:

- technologies for **content dissemination** (HTTP, HTML/XML, JSON, plain text, etc.);
- technologies for **information extraction** (R, Python, XPath, JSON parsers, Beautiful Soup, Selenium, regexps, etc.), and
- technologies for **data storage** (R, Python, SQL, binary formats, plain text formats, etc.).

16.2.1 Content Dissemination

The information that web scrapers look for on webpages appears in one of the following formats:

HTML – Hypertext Markup Language is used to display information on the web; it is not a dedicated data storage format, but it typically contains the information of interest; HTML is interpreted and transformed into "pretty" output by browsers (using CSS);

Figure 16.5: Comparison between HTML and XML (left, e-cartouche.ch ⧉), and between JSON and XML (right, activeVOS.com ⧉).

XML – Extensible Markup Language is a popular format for exchanging data over the web; its main purpose is to store data; XML is data wrapped in user-defined tags and as such is more flexible for storing data than HTML is;

JSON – JavaScript Object Notation is another data storage and exchange format; it is compatible with many programming languages and software; it is easier to parse than HTML or XML, and there is no need to use a specific query language (high level R is usually sufficient);

AJAX – Asynchronous JavaScript and XML is a group of technologies that enables websites to request data in the background of the browser session and update its visual appearance in a dynamic fashion, while allowing navigation to proceed when waiting for server reply (this can be a nuisance for web scrapers).

16.2.2 Hyper Text Transfer Protocol

Hypertext Transfer Protocol (HTTP) is a message language used between web browsers and web servers; Hypertext Transfer Protocol **Secure** (HTTPS) combines HTTP with SSL (encryption) and TLS (authentication) protocols.

In a nutshell, when we type in a URL in a browser to access a web page, the browser sends an HTTP **request** to the underlying **server**.

A request is made up of a verb, a path, a list of headers, and possibly some parameters. Common **verbs** include: GET (click on a link) and PUT (fill-out a form and submit).

For instance, if we type `http://www.yahoo.com/search` into the browser, a GET request is sent by the browswer to the `yahoo.com` server, together with the path `/search`.

NORMAL HTTP REQUEST

AJAX XMLHTTP REQUEST
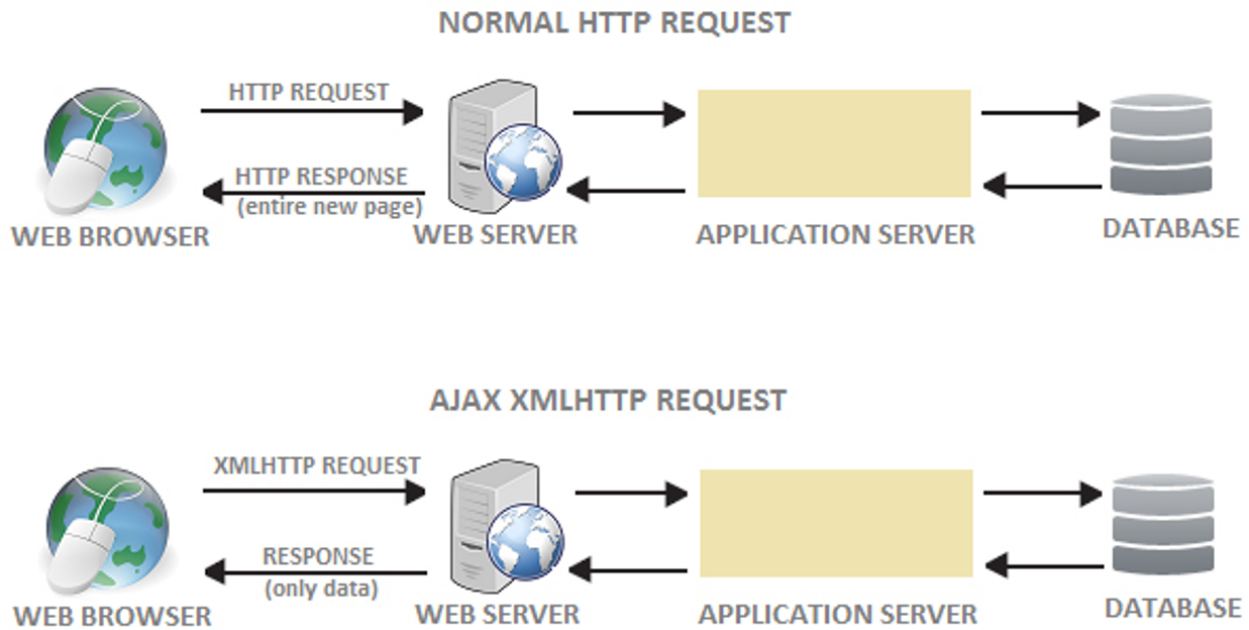
Figure 16.6: Schematics of HTTP (top) and AJAX (bottom) requests; a new HTTP request refreshes the entire page, a new AJAX request only refreshes the data javabelazy.blogspot.ca ⬀ .

The web server then sends a **response** to the browser, containing a code (404, 200, etc.), as well as headers and content.

The 200 response, say, means that the request was successful: the browser reads the content and uses it (and CSS files) to display the page.

16.2.3 Web Content

Webpage content itself comes into three main types:

- **Hypertext Markup Language** and variants (HTML/XML) is used for web content and code;
- **Cascading Style Sheets** (CSS) is used to define the webpage style, and
- **JavaScript** (JS) is used to provide webpage interactivity.

HTML is, in some sense, the most **fundamental** (the other two are optional); HTML is a **document language**, like LATEXor markdown (on which this book is based). A fresh HTTP/HTTPS request for a page usually returns an HTML file, which may contain references to additional server files (CSS, JavaScript, images, etc.) – the browser makes additional requests for these when the webpage is rendered.

Understanding the **tree structure** of HTML documents goes a long way towards helping analysts make full use of the **scraping toolbox** (see Section 16.3).

CSS defines the colour schemes, the fonts, spacing, and so on. It operates basically as a PowerPoint template would. In the absence of a CSS file, the browser uses a default style to render the webpage.

JS, on the other hand, is a programming language. After the browser parses and displays the HTML file, it executes any JS files referenced

in the HTML. JS can be used to manipulate most things on the page (delete/add/change content, change CSS, fetch more files from server, go to new page, etc.), and it can set up actions that run as a result of page events (clicking a button, typing in a text box, etc.)

16.2.4 HTML/XML

HTML syntax is fairly straightforward. HTML is a document language based on **tags**. Tags either come in **pairs**:

- `<title>...</title>` (self-explanatory),
- `<b>...</b>` (bold face text), etc.,

or as stand alone **singletons**:

- `<br>` (linebreak),
- `<hr>` (horizontal rule), etc.

Paired tags are **nested**:

- `<em><strong>...</strong></em>` is acceptable, whereas
- `<em><strong>...</em></strong>` is not.

An HTML file is a **tree of tags**, also known as **elements** or **nodes**.

Tags consist of a **name/type** (mandatory) and **attributes** (optional): the tag `<p lang="en">...</p>`, for instance, is of type p (paragraph), and it has a single attribute: `lang="en"`.

Plain text is allowed inside tags: `<span>Hello World!</span>`.

Beyond this, the only other thing left to learn is the set of possible tags, and the set of possible attributes. The list is extensive; information can be found at [2].

Two attributes are particularly important for web scraping:

- id uniquely identifies an element: `<a id="product">...</a>`, `<p id="saleInfo">...</p>`, etc.;
- class can contain multiple values, separated by spaces and is not unique, but it identifies a set of elements: `<h1 class="lightBackground oddPage">...</h1>`, etc.

16.2.5 Cookies and Other Headers

We discuss briefly three common headers:

- a **cookie** is a string that is sent and received with HTTP/HTTPS; it allows servers to keep track of user sessions. Upon logging on to a website, users receive a cookie. If the cookie is included in future requests, the user (and its preferences and choices) is recognized by the server; otherwise, the website acts as though the user has logged out.
- **user agent** contain the name and the version of the user's browser.
- **referrer** sends the page URL from which the request was initiated; if the user is on Page A and clicks a link to Page B, the server for Page B will see that the user came from Page A.
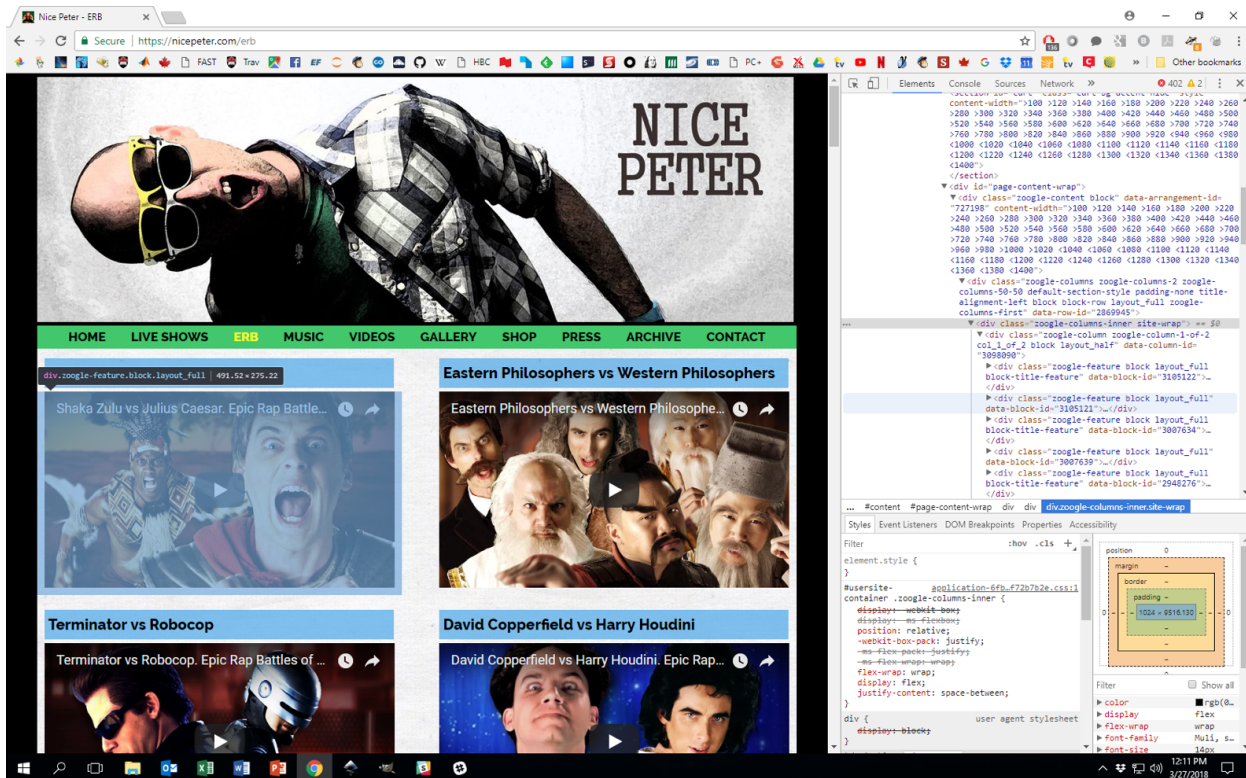
Figure 16.7: Inspecting Nice Peter ↗'s website's elements using Developer Tools in Chrome.

16.3 Scraping Toolbox

From experience, we know that a number of tools can facilitate the automated data extraction process, including:

- Developer Tools,
- XPath,
- regular expressions,
- Beautiful Soup, and
- Selenium.

We will briefly introduce each of them in this section.

16.3.1 Developer Tools

Developer Tools allow us to see the correspondence between the HTML code for a page and the rendered version seen in the browser, as illustrated in Figure 16.7.

Unlike "View Source", Developer Tools show the *dynamic* version of the HTML content.[7] Inspecting a page's various elements and discovering where they reside in the HTML file is **crucial** to efficient web scraping:

- **Firefox** – right click page → Inspect Element
- **Safari** – Safari → Preferences → Advanced → Show Develop Menu in Menu Bar, then Develop → Show Web Inspector
- **Chrome** – right click page → Inspect

7: That is, the HTML is shown with any changes made by JavaScript since the page was first received.

16.3.2 XPath

XPath is a query (domain-specific) language which is used to select specific pieces of information from marked-up documents.[8] Before this can be done, the information stored in a marked-up document needs to be converted (or **parsed**) into a format suitable for processing and statistical analysis; this is implemented in the R package XML, for instance.

8: Such as HTML, XML, or variants such as SVG, RSS.

The process is simple; it involves

1. **specifying** the data of interest;
2. **locating it** in a specific document, and
3. tailoring a query to the document to **extract** the desired info.

HTML/XML tags have **attributes** and **values**. HTML files must be parsed before they can be queried by XPath. XPath queries require both a **path** and a **document** to search; paths consist of hierarchical addressing mechanism (succession of nodes, separated by forward slashes ("/"), while a query takes the form xpathSApply(doc,path).[9]

9: xpathSApply(parsed_doc,"/html/body/div/p/i"), for instance, would find all <i> tags under a <p> tag, itself under a <div> tag in the body of the html file of parsed_doc. A substantially heftier treatment can be found in [6].

We will illustrate Xpath's functionality with the following webpage:

Laws of the *Internet*
Osmo Antero Wiio

Communication usually fails, except by accident.

Source: Wiion lait - ja vähän muidenkin

Melvin Kranzberg

Technology is neither good nor bad; nor is it neutral.
(Kranzberg's 1st Law)

Source: Technology and Culture. 27 (3): 544–560.

Theodore Sturgeon

90% of everything is crap.
(Sturgeon's Revelation)

Source: "Books: On Hand". Venture Science Fiction. Vol. 2, no. 2. p. 66.

Others:

- The 1% Rule: "Only 1% of the users of a website actively create new content, while the other 99% of the participants only lurk."
- D!@kwad Theory: "Normal Person + Anonymity + Audience = Total D!@kwad"
- Godwin's Law: "As an online discussion grows longer, the probability of a comparison involving Nazis or Hitler approaches one."
- Poe's Law: "Without a clear indicator of the author's intent, parodies of extreme views will be mistaken by some readers or viewers as sincere expressions of the parodied views."
- Skitt's Law: "Any post correcting an error in another post will contain at least one error itself."
- Law of Exclamation: "The more exclamation points used in an email (or other posting), the more likely it is a complete lie."
- Cunningham's Law: "The best way to get the right answer on the Internet is not to ask a question, it's to post the wrong answer."
- The Wiki Rule: "There's a wiki for that."
- Danth's Law: "If you have to insist that you've won an Internet argument, you've probably lost badly."
- Law of the Echo Chamber: "If you feel comfortable enough to post an opinion of any importance on any given Internet site, you are most likely delivering that opinion to people who already agree with you."
- Munroe's Law: "You will never change anyone's opinion on anything by making a post on the Internet. This will not stop you from trying."

15 Fundamental Laws of the Internet, by Matthew Jones

Figure 16.8: A simple HTML document, rendered in a browser, based on [4].

The underlying HTML code is in the file laws.html; we parse the document using XML's htmlParse().

```
parsed_doc <- XML::htmlParse(file = "Data/laws.html")
print(parsed_doc)
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head><title>Laws of the Internet</title></head>
<body>
<h1>Laws of the <i>Internet</i>
</h1>
<div id="wiio" lang="english" date="1978">
  <h2>Osmo Antero Wiio</h2>
  <p><i>Communication usually fails, except by accident.</i></p>
  <p><b>Source: </b>Wiion lait - ja vähän muidenkin</p>
</div>

<div lang="english" date="1986">
  <h2>Melvin Kranzberg</h2>
  <p><i>Technology is neither good nor bad; nor is it neutral.</i> <br><emph>(Kranzberg's 1st Law)</emph></p>
  <p><b>Source: </b><a href="https://www.jstor.org/stable/3105385">Technology and Culture. 27 (3): 544-560.</a></p>
</div>

<div lang="english" date="1958">
  <h2>Theodore Sturgeon</h2>
  <p><i>90% of everything is crap.</i> <br><emph>(Sturgeon's Revelation)</emph></p>
  <p><b>Source: </b>"Books: On Hand". Venture Science Fiction. Vol. 2, no. 2. p. 66.</p>
</div>

<div id="other">
<h2>Others:</h2>
<ul>
<li>The 1% Rule: "Only 1% of the users of a website actively create new content, while the other 99% of
    the participants only lurk."</li>
<li>D!@kwad Theory: "Normal Person + Anonymity + Audience = Total D!@kwad"</li>
<li>Godwin's Law: "As an online discussion grows longer, the probability of a comparison involving Nazis
    or Hitler approaches one."</li>
<li>Poe's Law: "Without a clear indicator of the author's intent, parodies of extreme views will be mistaken
    by some readers or viewers as sincere expressions of the parodied views."</li>
<li>Skitt's Law: "Any post correcting an error in another post will contain at least one error itself."</li>
<li>Law of Exclamation: "The more exclamation points used in an email (or other posting), the more likely
    it is a complete lie."</li>
<li>Cunningham's Law: "The best way to get the right answer on the Internet is not to ask a question, it's to
    post the wrong answer."</li>
<li>The Wiki Rule: "There's a wiki for that."</li>
<li>Danth's Law: "If you have to insist that you've won an Internet argument, you've probably lost badly."</li>
<li>Law of the Echo Chamber: "If you feel comfortable enough to post an opinion of any importance on any
    given Internet site, you are most likely delivering that opinion to people who already agree with you."</li>
<li>Munroe's Law: "You will never change anyone's opinion on anything by making a post on the Internet.
    This will not stop you from trying."</li>
</ul>
</div>

<address>
<a href="https://exceptionnotfound.net/15-fundamental-laws-of-the-internet/"><i>15 Fundamental Laws
    of the Internet</i></a>, by Matthew Jones<a></a>
</address>

</body>
</html>
```

Figure 16.9: The HTML document tree for the R built-in `fortunes.html` file [6].

Basic Structural Queries

XPath queries are called using `xpathSApply()`, which requires a parsed document `doc` and a query path `path`.

It is much easier to determine the required query paths if we have some idea of the structure of the underlying **HTML document tree**.[10]

Absolute paths are represented by single forward slashes [/]; **relative paths** by double forward slashes [//]. The next three calls will all return the same output.

```
XML::xpathSApply(doc = parsed_doc, path = "/html/body/div/p/i")
XML::xpathSApply(parsed_doc, "//body//p/i")
XML::xpathSApply(parsed_doc, "//p/i")
```

```
[[1]]
<i>Communication usually fails, except by accident.</i>

[[2]]
<i>Technology is neither good nor bad; nor is it neutral.</i>

[[3]]
<i>90% of everything is crap.</i>
```

Wildcards are represented by an asterisk [∗]: the code below once again has the same output as the one above.

```
XML::xpathSApply(parsed_doc, "/html/body/div/*/i")
```

Going up one level in the parsed tree is represented by a double dot [..].

```
XML::xpathSApply(parsed_doc, "//title/..")
```

```
[[1]]
<head>
  <title>Laws of the Internet</title>
</head>
```

The **disjunction** (OR) of two paths is represented by the operator [|].

```
XML::xpathSApply(parsed_doc, "//address | //title")
```

```
[[1]]
<title>Laws of the Internet</title>

[[2]]
<address>
<a href="https://exceptionnotfound.net/15-fundamental-laws-of-the-internet/">
    <i>15 Fundamental Laws of the Internet</i></a>, by Matthew Jones<a/>
</address>
```

We can also **concatenate** multiple queries (which, in this case, would produce the same output as the immediate call above).

```
twoQueries <- c(address = "//address", title = "//title")
XML::xpathSApply(parsed_doc, twoQueries)
```

Note, however, that absolute (or even relative) paths cannot always succinctly select nodes in large or complicated files.

Node Relations

A query's path can also exploit a node's relation to other nodes. By analogy with a **family tree**, a node's placement in the parsed tree often mimics the relations in extended families.

Relations are denoted according to `node1/relation::node2`. For instance:

- `"//a/ancestor::div"` returns all `<div>` nodes that are an ancestor to an `<a>` node;
- `"//a/ancestor::div//i"` returns all `<i>` nodes contained in a `<div>` node that is an ancestor to an `<a>` node, etc.[11]

11: See Figure 16.10 for a complete list of node relations.

Axis name	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects all nodes that appear before the current node in the document except ancestors, attribute nodes, and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

Figure 16.10: Generic node relations [6].

The following XPath query looks for <a> tags in the document, and produces their **ancestors** <div> tag.[12]

12: There is only one of each in this example, but that is an accident of the file with which we are working; there could be more in general.

```
XML::xpathSApply(parsed_doc, "//a/ancestor::div")
```

```
[[1]]
<div lang="english" date="1986">
  <h2>Melvin Kranzberg</h2>
  <p><i>Technology is neither good nor bad; nor is it neutral.</i>
    <br/><emph>(Kranzberg's 1st Law)</emph></p>
  <p><b>Source: </b><a href="https://www.jstor.org/stable/3105385">Technology and Culture.
    27 (3): 544-560.</a></p>
</div>
```

The following XPath query looks for <a> tags in the document, and produces all <i> tags of their ancestors <div> tag (there is only one in this example).

```
XML::xpathSApply(parsed_doc, "//a/ancestor::div//i")
```

```
[[1]]
<i>Technology is neither good nor bad; nor is it neutral.</i>
```

The following XPath query looks for <p> tags in the document, and produces the <h2> tags of all their preceding-sibling nodes (there are three in this example).

```
XML::xpathSApply(parsed_doc, "//p/preceding-sibling::h2")
```

```
[[1]]
<h2>Osmo Antero Wiio</h2>

[[2]]
<h2>Melvin Kranzberg</h2>

[[3]]
<h2>Theodore Sturgeon</h2>
```

What do you think this query will do?

```
XML::xpathSApply(parsed_doc, "//title/parent::*")
```

XPath Predicates

A **predicate** is a function that applies to a node's **name**, **value**, or **attributes** and that returns a logical TRUE or FALSE.

Predicates modify the path input of an XPath query: the query selects the nodes for which the relation holds.

Predicates are denoted by square brackets, placed after a node.

For instance:

- "//p[position()=1]" returns the first <p> node relative to its parent node;
- "//p[last()]" returns the last <p> node relative to its parent node, and
- "//div[count(./@*)>2]" returns all <div> nodes with 2+ attributes.

This XPath query finds the first <p> node in each <div> node.

```
XML::xpathSApply(parsed_doc, "//div/p[position()=1]")
```

```
[[1]]
<p>
  <i>Communication usually fails, except by accident.</i>
</p>

[[2]]
<p><i>Technology is neither good nor bad; nor is it neutral.</i>
<br/><emph>(Kranzberg's 1st Law)</emph></p>

[[3]]
<p><i>90% of everything is crap.</i>
<br/><emph>(Sturgeon's Revelation)</emph></p>
```

This XPath query finds the last <p> node in each <div> node.

```
XML::xpathSApply(parsed_doc, "//div/p[last()]")
```

```
[[1]]
<p><b>Source: </b>Wiion lait - ja vähän muidenkin</p>

[[2]]
<p>
  <b>Source: </b>
  <a href="https://www.jstor.org/stable/3105385">Technology and Culture. 27 (3): 544-560.</a>
</p>

[[3]]
<p><b>Source: </b>"Books: On Hand". Venture Science Fiction. Vol. 2, no. 2. p. 66.</p>
```

This next XPath query finds the second last <p> node in each <div> node.

```
XML::xpathSApply(parsed_doc, "//div/p[last()-1]")
```

```
[[1]]
<p>
  <i>Communication usually fails, except by accident.</i>
</p>

[[2]]
<p><i>Technology is neither good nor bad; nor is it neutral.</i>
<br/><emph>(Kranzberg's 1st Law)</emph></p>

[[3]]
<p><i>90% of everything is crap.</i> <br/><emph>(Sturgeon's Revelation)</emph></p>
```

This XPath query finds the <div> nodes that have at least one <a> node among their children.

```
XML::xpathSApply(parsed_doc, "//div[count(.//a)>0]")
```

```
[[1]]
<div lang="english" date="1986">
  <h2>Melvin Kranzberg</h2>
  <p><i>Technology is neither good nor bad; nor is it neutral.</i>
  <br/><emph>(Kranzberg's 1st Law)</emph></p>
  <p><b>Source: </b><a href="https://www.jstor.org/stable/3105385">Technology and Culture.
  27 (3): 544-560.</a></p>
</div>
```

A number of commonly-used XPath functions are shown in Table 16.1.

For instance, the following XPath query finds the <div> nodes that have more than 2 attributes.

```
XML::xpathSApply(parsed_doc, "//div[count(./@*)>2]")
```

```
[[1]]
<div id="wiio" lang="english" date="1978">
  <h2>Osmo Antero Wiio</h2>
  <p><i>Communication usually fails, except by accident.</i></p>
  <p><b>Source: </b>Wiion lait - ja vähän muidenkin</p>
</div>
```

This XPath query finds the nodes for which the text component has more than 50 characters.

```
XML::xpathSApply(parsed_doc, "//*[string-length(text())>50]")
```

```
[[1]]
<i>Technology is neither good nor bad; nor is it neutral.</i>

[[2]]
<p><b>Source: </b>"Books: On Hand". Venture Science Fiction. Vol. 2, no. 2. p. 66.</p>

[[3]]
<li>The 1% Rule: "Only 1% of the users of a website actively create new content, while ...

[[4]]
<li>D!@kwad Theory: "Normal Person + Anonymity + Audience = Total D!@kwad"</li>

[[5]]
<li>Godwin's Law: "As an online discussion grows longer, the probability of a comparison ...

[[6]]
<li>Poe's Law: "Without a clear indicator of the author's intent, parodies of extreme views ...

[[7]]
<li>Skitt's Law: "Any post correcting an error in another post will contain at least ...

[[8]]
<li>Law of Exclamation: "The more exclamation points used in an email (or other posting), the ...

[[9]]
<li>Cunningham's Law: "The best way to get the right answer on the Internet is not to ask ...

[[10]]
<li>Danth's Law: "If you have to insist that you've won an Internet argument, you've ...

[[11]]
<li>Law of the Echo Chamber: "If you feel comfortable enough to post an opinion of ...

[[12]]
<li>Munroe's Law: "You will never change anyone's opinion on anything by making a post ...
```

This XPath query finds all `<div>` nodes with 2 or fewer attributes.

```
XML::xpathSApply(parsed_doc, "//div[not(count(./@*)>2)]")
```

```
[[1]]
<div lang="english" date="1986">
  <h2>Melvin Kranzberg</h2>
  <p><i>Technology is neither good nor bad; nor is it neutral.</i> <br/><emph>(Kranzberg's ...
  <p><b>Source: </b><a href="https://www.jstor.org/stable/3105385">Technology and Culture...
</div>

[[2]]
<div lang="english" date="1958">
  <h2>Theodore Sturgeon</h2>
  <p><i>90% of everything is crap.</i> <br/><emph>(Sturgeon's Revelation)</emph></p>
  <p><b>Source: </b>"Books: On Hand". Venture Science Fiction. Vol. 2, no. 2. p. 66.</p>
</div>

[[3]]
<div id="other">
<h2>Others:</h2>
<ul><li>The 1% Rule: "Only 1% of the users of a website actively create new content...
<li>D!@kwad Theory: "Normal Person + Anonymity + Audience = Total D!@kwad"</li>
<li>Godwin's Law: "As an online discussion grows longer, the probability of a comparison ...
<li>Poe's Law: "Without a clear indicator of the author's intent, parodies of extreme ...
<li>Skitt's Law: "Any post correcting an error in another post will contain at least ...
<li>Law of Exclamation: "The more exclamation points used in an email (or other posting), the ...
<li>Cunningham's Law: "The best way to get the right answer on the Internet is not to ...
<li>The Wiki Rule: "There's a wiki for that."</li>
<li>Danth's Law: "If you have to insist that you've won an Internet argument, you've ...
<li>Law of the Echo Chamber: "If you feel comfortable enough to post an opinion of any ...
<li>Munroe's Law: "You will never change anyone's opinion on anything by making a post ...
</ul></div>
```

Can you predict what the following queries do? What they will return?

```
XML::xpathSApply(parsed_doc, "//div[@date='1958']")
XML::xpathSApply(parsed_doc, "//*[contains(text(), '%')]")
XML::xpathSApply(parsed_doc, "//div[starts-with(./@id, 'wiio')]")
```

Extracting Node Elements

XPath queries can also extract specific elements, using the `fun` option
(`xmlValue`, `xmlAttrs`, `xmlGetAttr`, `xmlName`, `xmlChildren`, `xmlSize`).

For instance, `xmlValue` returns a node's value:

```
XML::xpathSApply(parsed_doc, "//title", fun = XML::xmlValue)
```

```
[1] "Laws of the Internet"
```

Function	Description	Example
`name(<node>)`	Returns the name of `<node>` or the first node in a node set	`//*[name()='title']`; Returns: `<title>`
`text(<node>)`	Returns the value of `<node>` or the first node in a node set	`//*[text()='The book homepage']`; Returns: `<i>` with value *The book homepage*
`@attribute`	Returns the value of a node's *attribute* or of the first node in a node set	`//div[@id='R Inventor']`; Returns: `<div>` with attribute *id* value *R Inventor*
`string-length(str1)`	Returns the length of `str1`. If there is no string argument, it returns the length of the string value of the current node	`//h1[string-length()>11]`; Returns: `<h1>` with value *Robert Gentleman*
`translate(str1, str2, str3)`	Converts `str1` by replacing the characters in `str2` with the characters in `str3`	`//div[translate(./@date, '2003', '2005')='June/2005']`; Returns: first `<div>` node with date attribute value *June/2003*
`contains(str1,str2)`	Returns `TRUE` if `str1` contains `str2`, otherwise `FALSE`	`//div[contains(@id, 'Inventor')]`; Returns: first `<div>` node with id attribute value *R Inventor*
`starts-with(str1,str2)`	Returns `TRUE` if `str1` starts with `str2`, otherwise `FALSE`	`//i[starts-with(text(), 'The')]`; Returns: `<i>` with value *The book homepage*
`substring-before(str1,str2)`	Returns the start of `str1` before `str2` occurs in it	`//div[substring-before(@date, '/')='June']`; Returns: `<div>` with date attribute value *June/2003*
`substring-after(str1,str2)`	Returns the remainder of `str1` after `str2` occurs in it	`//div[substring-after(@date, '/')=2003]`; Returns: `<div>` with date attribute value *June/2003*
`not(arg)`	Returns `TRUE` if the boolean value is `FALSE`, and `FALSE` if the boolean value is `TRUE`	`//div[not(contains(@id, 'Inventor'))]`; Returns: the `<div>` node that does not contain the string *Inventor* in its id attribute value
`local-name(<node>)`	Returns the name of the current `<node>` or the first node in a node set—without the namespace prefix	`//*[local-name()='address']`; Returns: `<address>`
`count(<node>)`	Returns the count of a nodeset `<node>`	`//div[count(.//a)=0]`; Result: The second `<div>` with one `<a>` child
`position(<node>)`	Returns the index position of `<node>` that is currently being processed	`//div/p[position()=1]`; Result: The first `<p>` node in each `<div>` node
`last()`	Returns the number of items in the processed node list `<node>`	`//div/p[last()]`; Result: The last `<p>` node in each `<div>` node

Table 16.1: Commonly-used XPath functions [6].

On the other hand, `xmlAttrs` returns a node's attributes. In the first call, the first component returns 4 nodes; the second component returns each of these nodes' attributes.

```
XML::xpathSApply(parsed_doc, "//div", XML::xmlAttrs)
```

```
[[1]]
      id       lang       date
   "wiio"  "english"     "1978"

[[2]]
      lang       date
   "english"    "1986"

[[3]]                    [[4]]
      lang       date          id
   "english"    "1958"     "other"
```

Finally, `xmlGetAttr` can be used to return a specific attribute:

```
XML::xpathSApply(parsed_doc, "//div", XML::xmlGetAttr, "lang")
```

```
[[1]]          [[2]]          [[3]]          [[4]]
[1] "english"  [1] "english"  [1] "english"  NULL
```

16.3.3 Regular Expressions

Regular expressions can be used to achieve the main web scraping objective, which is to extract relevant information from reams of data. Among this mostly unstructured data lurk **systematic elements**, which can be used to help the automation process, especially if quantitative methods are eventually going to be applied to the scraped data.

Systematic structures include numbers, names (countries, etc.), addresses (mailing, e-mailing, URLs, etc.), specific character strings, etc. Regular expressions (**regexps**) are abstract sequences of strings that match concrete recurring patterns in text; they allow for the systematic extraction of the information components from plain text, HTML, and XML.

The examples in this section are based on [3].

Initializing the Environment

The Python module for regular expressions is re.

```
import re
```

Let us take a quick look at some basics, through the re method match(). We can try to match a pattern from the beginning of a string, as below:

```
re.match('super','supercalifragilisticexpialidocious')
```

```
<re.Match object; span=(0, 5), match='super'>
```

No such match occurs in the following chunk of code, however.

```
re.match('super','Supercalifragilisticexpialidocious')
```

The regular expression pattern (more on this in a moment) for "word" is \w+. The following bit of code would match the first word in a string:

```
w_regex = '\w+'
re.match(w_regex,'Hello World!')
```

```
<re.Match object; span=(0, 5), match='Hello'>
```

Common Regular Expression Patterns

A **regular expression pattern** is a short form used to indicate a type of (sub)string:

- \w+: word
- \d: digit
- \s: space
- .: wildcard

- + or ∗: greedy match
- \W: not word
- \D: not digit
- \S: not space
- [a-z]: lower case group
- [A-Z]: upper case group

There are a few re functions which, combined with regexps, can make it easier to extract information from large, unstructured text documents:

- split(): splits a string on a regexp;
- findall(): finds all substrings matching a regexp in a string;
- search(): searches for a regexp in a string, and
- match(): matches an entire string based on a regexp

Each of these functions takes two arguments: a **regexp** (first) and a **string** (second). For instance, we can split a string on the spaces (and remove them):

```
re.split('\s+','Can  you do the split?')
```

```
['Can', 'you', 'do', 'the', 'split?']
```

The \ in the regexp above is crucial. The following code splits the sentence on the s (and removes them):

```
re.split('s+','Can  you do the split?')
```

```
['Can  you do the ', 'plit?']
```

We can also split on single spaces and remove them:

```
re.split('\s','Can  you do the split?')
```

```
['Can', '', 'you', 'do', 'the', 'split?']
```

Alternatively, we can also split on the words and remove them:

```
re.split('\w+','Can  you do the split?')
```

```
['', ' ', ' ', ' ', ' ', '?']
```

Or better yet, split on the non-words and remove them:

```
re.split('\W+','Can you do the split?')
```

```
['Can', 'you', 'do', 'the', 'split', '']
```

Let us take some time to study a silly sentence, saved as a string.

```
test_string = 'Oh they built the built the ship Titanic.
  It was a mistake. It cost more than 1.5 million dollars.
  Never again!'
test_string
```

'Oh they built the built the ship Titanic. It was a mistake. It cost more than 1.5 million dollars.
Never again!'

In English, only three characters can end a sentence: ., ?, ![13] We create
a regexp group (more on those in a moment) as follows:[14]

```
sent_ends = r"[.?!]"
```

We could then split the string into its constituent sentences:

```
print(re.split(sent_ends,test_string))
```

['Oh they built the built the ship Titanic', ' It was a mistake',
 ' It cost more than 1', '5 million dollars', ' Never again', '']

If we wanted to know how many such sentences there were, we simply
use the len() function:

```
print(len(re.split(sent_ends,test_string)))
```

6

The regexp range consisting of words with an uppercase initial letter is
easy to build:

```
cap_words = r"[A-Z]\w+" # Upper case characters
```

We can find all such words (and how many there are in the string)
through:

```
print(re.findall(cap_words,test_string))
print(len(re.findall(cap_words,test_string)))
```

['Oh', 'Titanic', 'It', 'It', 'Never']
5

The regexp for spaces is:

```
spaces = r"\s+" # spaces
```

We can then split the string on spaces, and count the number of **tokens**
(see Chapter 27, *Text Analysis and Text Mining*):

13: Apparently, nobody's heard of the interrobang...

14: In Python, regular expression patterns must be prefixed with an r to differentiate between the **raw string** and the **string's interpretation**.

```
print(re.split(spaces,test_string))
print(len(re.split(spaces,test_string)))
```

```
['Oh', 'they', 'built', 'the', 'built', 'the', 'ship', 'Titanic.',
 'It', 'was', 'a', 'mistake.', 'It', 'cost', 'more', 'than', '1.5',
 'million', 'dollars.', 'Never', 'again!']
21
```

The regexp for numbers (contiguous strings of digits) is:

```
numbers = r"\d+"
```

We can find all the numeric characters using:

```
print(re.findall(numbers,test_string))
print(len(re.findall(numbers,test_string)))
```

```
['1', '5']
2
```

The main difference between `search()` and `match()` is that `match()` tries to match from the beginning of a string, whereas `search()` looks for a match anywhere in the string.

Regular Expressions Groups '( )' and Ranges '[ ]' With OR '|'

We can create more complicated regexps using **groups**, **ranges**, and/or "or" statements:

- `[a-zA-Z]+`: an unlimited number of lower and upper case English/French (unaccented) letters;
- `[0-9]`: the digits from 0 to 9;
- `[a-zA-Z'\.\-]+`: any combination of lower and upper case English/French (unaccented) letters, ', ., and -;
- `(a-z)`: the characters a, -, and z;
- `(\s+|,)`: any number of spaces, or a comma;
- `(\d+|\w+)`: words or numerics

For instance, consider the following text string and regexps groups:

```
text = 'On the 1st day of xmas, my boat sank.'
numbers_or_words = r"(\d+|\w+)"
spaces_or_commas = r"(\s+|,)"
```

This next chunk of code does exactly what one would expect:

```
print(re.findall(numbers_or_words,text))
```

```
['On', 'the', '1', 'st', 'day', 'of', 'xmas', 'my', 'boat', 'sank']
```

What about this one?

```
print(re.findall(spaces_or_commas,text))
```

```
[' ', ' ', ' ', ' ', ' ', ',', ' ', ' ', ' ']
```

Now, consider a different string:

```
text = "will something happen after the semi-colon; I don't think so"
```

What might happen in each of the following cases?

```
print(re.match(r"[a-z -]+",text))
print(re.match(r"[a-z ]+",text))
print(re.match(r"[a-z]+",text))
print(re.match(r"(a-z-)+",text))
```

16.3.4 BeautifulSoup

Simple web requests require some networking code to fetch a page and return the HTML contents.

Browsers do a lot of work to intelligently parse improper HTML syntax,[15] so that something like

```
<a href="data-action-lab.com> <b>link text<a> </b>,
```

say, would be correctly interpreted as

```
<a href="data-action-lab.com><b>link text</b></a>.
```

BeautifulSoup (BS) is a Python library that helps extract data out of HTML and XML files; it parses HTML files, even if they are broken. But BS does not simply convert bad HTML to good X/HTML; it allows a user to fully inspect the (proper) HTML structure it produces, in a programmatical fashion.[16]

Typical HTML elements to be extracted/read come in various formats, such as:

- text
- tables
- form field values
- images
- videos
- etc.

15: Only up to a certain point, of course.

16: The R equivalent is rvest; we will not describe how to use it, but you are **strongly encouraged** to read up on this versatile tool and to use it in the Exercises.

When BS has finished its work on an HTML file, the resulting *soup* is an API for **traversing**, **searching**, and **reading** the document's elements. In essence, it provides **idiomatic** ways of navigating, searching, and modifying the parse tree of the HTML file, which can save a fair amount of time.

For instance, soup.find_all('a') would find and output all <a ...> ... tag pairs (with attributes and content) in the soup, whereas the following chink of code would output the URLs found in the same tag pairs.

```python
for link in soup.find_all('a'):
    print(link.get('href')'
```

The *BeautifulSoup* documentation is quite explicit and provides numerous examples [1]. We use the lyrics to Meet the Elements ⬈ , a song by *They Might Be Giants*, to illustrate BeautifulSoup's functionality.

```
html_doc = """
<html>
<head><title>Meet the Elements</title> <meta name="author" content="They Might Be Giants"></head>
<body><p class="title"><b>Meet the Elements</b></p>

<p class="author"><i>They Might Be Giants</i></p>

<div class="lyrics"><p class="verse" id="verse1">
<a href="https://en.wikipedia.org/wiki/Iron" class="element" id="link1">Iron</a> is a metal, you
    see it every day<br>
<a href="https://en.wikipedia.org/wiki/Oxygen" class="element" id="link2">Oxygen</a>, eventually,
    will make it rust away<br>
<a href="https://en.wikipedia.org/wiki/Carbon" class="element" id="link3">Carbon</a> in its
    ordinary form is coal<br>
Crush it together, and diamonds are born</p>

<p class="chorus" id="chorus1">
Come on, come on, and meet the elements <br>
May I introduce you to our friends, the elements? <br>
Like a box of paints that are mixed to make every shade <br>
They either combine to make a chemical compound or stand alone as they are</p>

<p class="verse" id="verse2">
<a href="https://en.wikipedia.org/wiki/Neon" class="element" id="link4">Neon</a>'s a gas that
    lights up the sign for a pizza place <br>
The coins that you pay with are <a href="https://en.wikipedia.org/wiki/Copper" class="element"
    id="link5">copper</a>, <a href="https://en.wikipedia.org/wiki/Nickel" class="element"
    id="link6">nickel</a>, and <a href="https://en.wikipedia.org/wiki/Zinc" class="element"
    id="link7">zinc</a> <br>
<a href="https://en.wikipedia.org/wiki/Silicon" class="element" id="link8">Silicon</a> and oxygen
    make concrete bricks and glass <br>
Now add some <a href="https://en.wikipedia.org/wiki/Gold" class="element" id="link9">gold</a> and
    <a href="https://en.wikipedia.org/wiki/Silver" class="element" id="link10">silver</a> for some
        pizza place class</p>

<p class="chorus" id="chorus2">
Come on, come on, and meet the elements <br>
I think you should check out the ones they call the elements <br>
```

```
Like a box of paints that are mixed to make every shade <br>
They either combine to make a chemical compound or stand alone as they are <br>
Team up with other elements making compounds when they combine <br>
Or make up a simple element formed out of atoms of the one kind </p>

<p class="verse" id="verse3">
Balloons are full of <a href="https://en.wikipedia.org/wiki/Helium" class="element"
    id="link11">helium</a>, and so is every star <br>
Stars are mostly <a href="https://en.wikipedia.org/wiki/Hydrogen" class="element"
    id="link12">hydrogen</a>, which may someday fill your car <br>
Hey, who let in all these elephants? <br>
Did you know that elephants are made of elements? <br>
Elephants are mostly made of four elements <br>
And every living thing is mostly made of four elements <br>
Plants, bugs, birds, fish, bacteria and men <br>
Are mostly carbon, hydrogen, <a href="https://en.wikipedia.org/wiki/Nitrogen" class="element"
    id="link13">nitrogen</a>, and oxygen</p>

<p class="chorus" id="chorus3">
Come on, come on, and meet the elements <br>
You and I are complicated, but we're made of elements <br>
Like a box of paints that are mixed to make every shade <br>
They either combine to make a chemical compound or stand alone as they are <br>
Team up with other elements making compounds when they combine <br>
Or make up a simple element formed out of atoms of the one kind <br>
Come on come on and meet the elements <br>
Check out the ones they call the elements <br>
Like a box of paints that are mixed to make every shade <br>
They either combine to make a chemical compound or stand alone as they are</p>

</div>
"""
```

Note that the HTML file contains neither a </body> nor a </html> tag.
We import the BeautifulSoup module, and parse the file into a soup
using the html.parser.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
print(soup.prettify())
```

```
<html>
 <head>
  <title>
   Meet the Elements
  </title>
  <meta content="They Might Be Giants" name="author"/>
 </head>
 <body>
  <p class="title">
   <b>
    Meet the Elements
   </b>
  </p>
```

```
  <p class="author">
   <i>
    They Might Be Giants
   </i>
  </p>
  <div class="lyrics">
   <p class="verse" id="verse1">
    <a class="element" href="https://en.wikipedia.org/wiki/Iron" id="link1">
     Iron
    </a>
    is a metal, you see it every day
    <br/>

    ...

   <p class="chorus" id="chorus3">
    Come on, come on, and meet the elements
    <br/>
    You and I are complicated, but we're made of elements
    <br/>
    Like a box of paints that are mixed to make every shade
    <br/>
    They either combine to make a chemical compound or stand alone as they are
    <br/>
    Team up with other elements making compounds when they combine
    <br/>
    Or make up a simple element formed out of atoms of the one kind
    <br/>
    Come on come on and meet the elements
    <br/>
    Check out the ones they call the elements
    <br/>
    Like a box of paints that are mixed to make every shade
    <br/>
    They either combine to make a chemical compound or stand alone as they are
   </p>
  </div>
 </body>
</html>
```

The parser has "fixed" the file by appending the missing tags; it also indents the tags to make it easier to spot the document's hierarchic (tree) structure.

BeautifulSoup Functionality

Is the functionality of BS clear from the following examples?

```
print(soup.title)
```

```
<title>Meet the Elements</title>
```

```
print(soup.title.name)
```

title

```
print(soup.title.string)
```

Meet the Elements

```
print(soup.title.parent.name)
```

head

```
print(soup.p)
```

<p class="title">Meet the Elements</p>

```
soup.p['class']
```

['title']

```
print(soup.a)
```

Iron

```
soup.find_all('a')
```

[Iron,
 Oxygen,
 Carbon,
 Neon,
 copper,
 nickel,
 zinc,
 Silicon,
 gold,
 silver,
 helium,
 hydrogen,
 nitrogen]
```

```
print(soup.find(id="link5"))
```

```
<a class="element" href="https://en.wikipedia.org/wiki/Copper"
    id="link5">copper</a>
```

```
for link in soup.find_all('a'):
    print(link.get('href'))
```

```
https://en.wikipedia.org/wiki/Iron
https://en.wikipedia.org/wiki/Oxygen
https://en.wikipedia.org/wiki/Carbon
https://en.wikipedia.org/wiki/Neon
https://en.wikipedia.org/wiki/Copper
https://en.wikipedia.org/wiki/Nickel
https://en.wikipedia.org/wiki/Zinc
https://en.wikipedia.org/wiki/Silicon
https://en.wikipedia.org/wiki/Gold
https://en.wikipedia.org/wiki/Silver
https://en.wikipedia.org/wiki/Helium
https://en.wikipedia.org/wiki/Hydrogen
https://en.wikipedia.org/wiki/Nitrogen
```

```
print(soup.get_text())
```

```
Meet the Elements
Meet the Elements
They Might Be Giants
Iron is a metal, you see it every day
Oxygen, eventually, will make it rust away
Carbon in its ordinary form is coal
Crush it together, and diamonds are born
Come on, come on, and meet the elements
May I introduce you to our friends, the elements?
Like a box of paints that are mixed to make every shade
They either combine to make a chemical compound or stand alone as they are
Neon's a gas that lights up the sign for a pizza place
The coins that you pay with are copper, nickel, and zinc
Silicon and oxygen make concrete bricks and glass
Now add some gold and silver for some pizza place class
Come on, come on, and meet the elements
I think you should check out the ones they call the elements
Like a box of paints that are mixed to make every shade
They either combine to make a chemical compound or stand alone as they are
Team up with other elements making compounds when they combine
Or make up a simple element formed out of atoms of the one kind
Balloons are full of helium, and so is every star
Stars are mostly hydrogen, which may someday fill your car
Hey, who let in all these elephants?
Did you know that elephants are made of elements?
```

```
Elephants are mostly made of four elements
And every living thing is mostly made of four elements
Plants, bugs, birds, fish, bacteria and men
Are mostly carbon, hydrogen, nitrogen, and oxygen
Come on, come on, and meet the elements
You and I are complicated, but we're made of elements
Like a box of paints that are mixed to make every shade
They either combine to make a chemical compound or stand alone as they are
Team up with other elements making compounds when they combine
Or make up a simple element formed out of atoms of the one kind
Come on come on and meet the elements
Check out the ones they call the elements
Like a box of paints that are mixed to make every shade
They either combine to make a chemical compound or stand alone as they are
```

16.3.5 Selenium

Selenium is a Python tool used to automate web browser interactions. It is used primarily for testing purposes, but it has data extraction uses as well. Mainly, it allows the user to open a browser and to act as a human being would:

- clicking buttons;
- entering information in forms;
- searching for specific information on a page, etc.

Selenium requires a driver to interface with the chosen browser. Firefox, for example, uses `geckodriver`.[17]

17: Here are the driver URL for supported browsers:

- Chrome ☐
- Edge ☐
- Firefox ☐
- Safari ☐

Selenium automatically controls a complete browser, including **rendering** the web documents and **running JavaScript**. This is useful for pages with a lot of dynamic content that is not in the base HTML. Selenium can program actions like "click on this button", or "type this text", to provide access to the dynamic HTML of the current state of the page, not unlike what happens in *Developer Tools* (but now the process can be fully automated). More information can be found in [9, 7, 8].

16.3.6 APIs

An **application programming interface** (API) is a website's way of giving programs access to their data, without the need for scraping. APIs provide **structured access** to **structured data**: not every bit of information will necessarily be made available to analysts.

For example, a finance site might offer an API with financial aggregate data, the *New York Times* might offer an API for news articles from a specific time period, Twitter might offer an API to collect tweets by users or hashtags, etc. In all cases, however, the data will be available in a **pre-defined**, **structured** format (often JSON).

In the examples we consider in Section 16.4, the APIs we consider have R/Python libraries that encapsulate all required networking and encoding. This means that users only need to read the library documentation to get a sense for what needs to be done to get the data.[18]

18: A full list of R API libraries can be found here ☐ .

16.3.7 Specialized Uses and Applications

Although we will not be discussing them in these notes, it could prove useful for web scrapers to learn how to handle:

HTML Forms Sometimes we do not just want to receive data from the server, we also want to **send** data, such as a **username/password** combination to log in to a site. Other input types include: check boxes, radio buttons, hidden inputs, etc. Real users accomplish this by filling out forms and submitting them to the server. When this happens the browser looks at the form HTML and sends a request with the user inputs as *parameters*. The server can use those parameters to send back different data.

Encoding What if we wanted to write <br /> as **text** in an HTML file? If we just type it in as-is, it would be interpreted as an HTML tag, not as text. The solution is to use HTML **encoding**. In order to type <br />, we have to encode it in a special form of text that the browser understands. An HTML decoder/encoder can be found here ⧉ .

Combination HTML forms can specify a method for GET as well as for PUT. In that case the parameters are appended to the URL after a "?", like so:

http://search.yahoo.com/search/?p=data+analysis&lang=en.

In that example, the parameter names are p and lang. The parameter value data+analysis actually represents the string "data analysis", but spaces get encoded in URLs. Other characters (such as "/") often are as well; use the urlencoder.org ⧉ to get the correct strings.

16.4 Examples

In this section, we provide web scraping examples (in R and Python) that highlight some of the notions we discussed in the chapter.[19]

16.4.1 Wikipedia

This example is inspired by a task found in [6]. We analyze the list of largest cities on the planet, found on Wikipedia ⧉ .[20]

Preamble

We will be using the following R libraries:

- stringr, stringi, and strex, for string manipulation;
- XML, for reading and creating XML documents;
- maps, to display maps, and
- rvest, which provides a wrapper for HTTP requests in R.

19: These examples all worked as of Dec 2022; but it is possible that the websites that are being scraped have changed their structure or been deleted, or that the tools used have been updated/upgraded/made obsolete in the intervening time.

20: Wikipedia is a commonly-used source of data on various topics (in a first pass, at the very least), but it should probably not be your ONLY source of information.

Loading and Parsing the Data

We read the material from the Wikipedia website using rvest's read_-
html() command, and we store it to the object html.

```
html <- rvest::read_html("https://en.Wikipedia.org/wiki/List_of_largest_cities")
```

A call to the object shows the entire structure of the page under the
hood.

```
html
```

```
{html_document}
<html class="client-nojs" lang="en" dir="ltr">
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
[2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject  ...
```

Now that we have the information from the webpage, we parse it to
create a string of words.

```
cities_parsed <- XML::htmlParse(html, encoding="UTF-8")
```

Note that this new output contains the same information as the original
object html, but that if it was displayed, it would be so in a format
resembling what a human programer would expect to see (at least, to
some extent). We opt not to display it due to its excessive length.

Now that the information from the webpage is parsed, we create tables
to hold the words, using XML's readHTMLTable().

```
tables <- XML::readHTMLTable(cities_parsed, stringsAsFactors = FALSE)
```

Essentially, readHTMLTable() hunts for <table>...</table> tag pairs
in the file; it finds 4 here. We get some structural information by calling
str on the resulting object tables.

```
str(tables)
```

```
List of 4
 $ NULL:'data.frame':   5 obs. of  1 variable:
  ..$ V1: chr [1:5] "Ekistics" "" "List of largest cities\nList of cities proper by ...
 $ NULL:'data.frame':   84 obs. of  13 variables:
  ..$ V1 : chr [1:84] "City[a]" "Definition" " " "Tokyo" ...
  ..$ V2 : chr [1:84] "Country" "Population" "" " Japan" ...
  ..$ V3 : chr [1:84] "UN 2018 population estimates[b]" "Area.mw-parser-output  ...
  ..$ V4 : chr [1:84] "City proper[c]" "Density(/km2)" "" "Metropolis prefecture" ...
  ..$ V5 : chr [1:84] "Urban area[8]" "Population" "" "13,515,271" ...
  ..$ V6 : chr [1:84] "Metropolitan area[d]" "Area(km2)" "" "2,191" ...
  ..$ V7 : chr [1:84] NA "Density(/km2)" "" "6,169[13]" ...
  ..$ V8 : chr [1:84] NA "Population" "" "39,105,000" ...
```

```
 ..$ V9 : chr [1:84] NA "Area(km2)" "" "8,231" ...
 ..$ V10: chr [1:84] NA "Density(/km2)" "" "4,751[e]" ...
 ..$ V11: chr [1:84] NA NA "" "37,274,000" ...
 ..$ V12: chr [1:84] NA NA "" "13,452" ...
 ..$ V13: chr [1:84] NA NA "" "2,771[14]" ...
$ NULL:'data.frame':   7 obs. of  2 variables:
 ..$ V1: chr [1:7] "v\nt\ne\n\nWorld's largest cities" "City proper" "Metropolitan area" ...
 ..$ V2: chr [1:7] NA "Capitals\nAfrica\n\nAmericas (North\n\nLatin\nCentral\n\nSouth)\n\nAsia ...
$ NULL:'data.frame':   9 obs. of  2 variables:
 ..$ V1: chr [1:9] "v\nt\ne\n\nCities" "Urban geography" "Urban government" "Urban economics" ...
 ..$ V2: chr [1:9] NA "Urban area\n\nCity centre\nDowntown\nSuburb\nExurb\nCore city\nTwin ...
```

Data Processing and Data Cleaning

We extract the table containing the information of interest.

```
cities_table <- tables[[2]]
```

The column headers are not as we might want them:

```
colnames(cities_table)
```

```
 [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"
[13] "V8"  "V9"  "V10" "V11" "V12" "V13"
```

Compare with the second row of `cities_table`:

```
cities_table[2,]
```

```
        V1          V2                                                                V3
2 Definition Population Area.mw-parser-output .nobold{font-weight:normal}(km2)
            V4          V5         V6          V7          V8          V9
2 Density(/km2) Population Area(km2) Density(/km2) Population Area(km2)
            V10  V11  V12  V13
2 Density(/km2) <NA> <NA> <NA>
```

This is still not ideal: the first and second rows of the table contain variable information, and the data itself starts with row 3. We need to manually input the column names, and delete the non-data rows.

```
colnames(cities_table) <- c("city", "country", "un.2018.pop","city.def","city.pop","city.area",
   "city.den","metro.pop","metro.area","metro.den","urban.pop","urban.area","urban.den")
cities_table <- data.frame(cities_table[4:nrow(cities_table),])
```

We only select a sample of the columns of the table:

- `city` [1];
- `country` [2];
- `urban.pop` [11];
- `urban.area` [12], and
- `urban.den` [13].

```
cities_table <- cities_table[,c(1,2,11,12,13)]
```

It is never a bad idea to **validate** our work as we build the scraper: are we getting what we would expect along the way? Let us take a look at the structure of the data and compare the first 6 entries of the table to the information we can see on the Wikipedia page.

```
str(cities_table)
```

```
'data.frame':   81 obs. of  5 variables:
 $ city      : chr  "Tokyo" "Delhi" "Shanghai" "São Paulo" ...
 $ country   : chr  " Japan" " India" " China" " Brazil" ...
 $ urban.pop : chr  "37,274,000" "29,000,000" "--" "21,734,682" ...
 $ urban.area: chr  "13,452" "3,483" "--" "7,947" ...
 $ urban.den : chr  "2,771[14]" "8,326[16]" "--" "2,735[20]" ...
```

```
head(cities_table)
```

	city	country	urban.pop	urban.area	urban.den
4	Tokyo	Japan	37,274,000	13,452	2,771[14]
5	Delhi	India	29,000,000	3,483	8,326[16]
6	Shanghai	China	—	—	—
7	São Paulo	Brazil	21,734,682	7,947	2,735[20]
8	Mexico City	Mexico	21,804,515	7,866	2,772[22]
9	Cairo	Egypt	—	—	—

We see that all variables appear as **character strings**, and that there are oddities with some of the numerical values (square brackets, missing values, comma separators, etc.).

We obtain the numerical values using stringr's str_extract() and regexps(), or strex's str_extract_numbers() and str_first_number().

The urban populations are all above 5M, and they are all displayed using comma separators, thus they all have values that look like ddd,ddd,ddd, where d ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

We extract only the portion of the strings that follow this pattern from the population column using str_extract(),[21] removing the commas after the fact using gsub(), and coercing the outcome to a numerical format using as.numeric().

21: Which does not retain the footnote markers.

```
cities_table$urban.pop <- as.numeric(
  gsub(",", "",
      stringr::str_extract(
        cities_table$urban.pop,
        stringr::regex("\\d+,\\d+,\\d+")
        )
      )
  )
```

The area column contains no footnote, so we can directly extract the comma-separated values (using `str_extract_numbers()`) and coerce to a vector using `as.numeric()`.[22]

```
cities_table$urban.area <- as.numeric(
  strex::str_extract_numbers(
    cities_table$urban.area,
    commas=TRUE
    )
  )
```

Finally, we extract the first number that appears in each density value, removing the footnotes,[23] using `str_first_number()`; the result is then coerced to a numeric vector using `as.numeric()`.

```
cities_table$urban.den <-
  as.numeric(strex::str_first_number(
    cities_table$urban.den,
    commas=TRUE
    )
  )
```

The first six entries are shown below.

```
rownames(cities_table) = NULL
head(cities_table)
```

city	country	urban.pop	urban.area	urban.den
Tokyo	Japan	37274000	13452	2771
Delhi	India	29000000	3483	8326
Shanghai	China	NA	NA	NA
São Paulo	Brazil	21734682	7947	2735
Mexico City	Mexico	21804515	7866	2772
Cairo	Egypt	NA	NA	NA

We can download latitude and longitude details ⬀ for ≈ 41K cities.

```
world_cities = read.csv("worldcities.csv",
    stringsAsFactors = TRUE, nrow=200)
str(world_cities)
```

```
'data.frame':   200 obs. of  3 variables:
 $ city_ascii: Factor w/ 198 levels "Abidjan","Ahmedabad", ...
 $ lat       : num  35.69 -6.21 28.66 18.97 14.6 ...
 $ lng       : num  139.7 106.8 77.2 72.8 121 ...
```

We extract a 5-digit code for each city, in the hope of being able to match them in both datasets.

We remove accents using `stringi`'s `stri_trans_general()`, which will convert every character to its nearest equivalent in the Latin ASCII character list.

```
world_cities$code = stringi::stri_trans_general(
  tolower(
    substr(
      world_cities$city_ascii,1,5)
    ),
  "Latin-ASCII"
  )

cities_table$code = stringi::stri_trans_general(
  tolower(
    substr(cities_table$city,1,5)
    ),
  "Latin-ASCII"
  )
```

We merge the data frames:

```
(complete = merge(cities_table, world_cities, all.x=TRUE))
```

code	city	country	urban.pop	urban.area	urban.den	city_ascii	lat	lng
ahmed	Ahmedabad	India	6300000	NA	NA	Ahmedabad	23.0300	72.5800
alexa	Alexandria	Egypt	NA	NA	NA	Alexandria	31.2000	29.9167
atlan	Atlanta	United States	5949951	21690	274	Atlanta	33.7627	-84.4224
baghd	Baghdad	Iraq	NA	NA	NA	Baghdad	33.3500	44.4167
banga	Bangalore	India	NA	NA	NA	Bangalore	12.9699	77.5980
...	...	...	...	...	...	...	...	...
seoul	Seoul	South Korea	25514000	11704	2180	Seoul	37.5600	126.9900
shang	Shanghai	China	NA	NA	NA	Shangrao	28.4419	117.9633
shang	Shanghai	China	NA	NA	NA	Shanghai	31.1667	121.4667
shang	Shanghai	China	NA	NA	NA	Shangqiu	34.4259	115.6467
...	...	...	...	...	...	...	...	...
suzho	Suzhou	China	NA	NA	NA	Suzhou	31.3040	120.6164
suzho	Suzhou	China	NA	NA	NA	Suzhou	33.6333	116.9683
...	...	...	...	...	...	...	...	...
toron	Toronto	Canada	5928040	5906	1004	Toronto	43.7417	-79.3733
washi	Washington	United States	6263245	17009	368	Washington	38.9047	-77.0163
wuhan	Wuhan	China	NA	NA	NA	Wuhan	30.5872	114.2881
xi'an	Xi'an	China	NA	NA	NA	Xi'an	34.2667	108.9000
yango	Yangon	Myanmar	NA	NA	NA	NA	NA	NA

There are still some issues with the data:

- Suzhou shows up twice, with two different sets of coordinates, but the appropriate coordinates are found online to be (31.299999, 120.599998);
- Neither Yangon nor Fukuoka appear in the world_cities dataset, but their coordinates are found online to be (16.871311,96.199379) and (33.583332,130.399994), respectively;
- Shanghai has been associated to three cities: Shanghai, Shangrao, and Shangqiu, each with its own coordinates. As neither Shangrao nor Shangqiu appears in the original list, they may be removed with impunity,

- there is no population data for Foshan, but the Wikipedia page informs us that Foshan is included in the Guangzhou urban area, so we will remove the former from the dataset, and
- there are missing population, area, and density values for a number of cities, but these were missing in the original dataset, so we will leave it be for now.[24]

24: Were they really, though? The problem arises because variables V11, V12, and V13 were poor choices in the first place. We will ask you to revisit this in the exercises.

```
# remove duplicate entries
complete = complete[-c(23,68,70,76),c(7,3,8,9,4,5,6)]
rownames(complete) = NULL

# add Fukuoka coordinates
complete[23,3] = 33.5833
complete[23,4] = 130.3999

# add Yangon coordinates
complete[80,3] = 16.8713
complete[80,4] = 96.1994

# add new factor levels for missing city names
complete$city_ascii = factor(complete$city_ascii,
  levels=c(levels(complete$city_ascii),"Yangon","Fukuoka"))
complete[23,1] = "Fukuoka"
complete[80,1] = "Yangon"

# rename city_ascii to city
colnames(complete)[1] <- "city"
```

Visualization

All the work we have done has brought the data in a format that is amenable to analysis. As an illustration, we plot the cities on a map of the world. We can display a Mercator projection by using maps's map().

```
par(oma=c(0,0,0,0)); par(mar=c(0,0,0,0))
maps::map("world", col = "darkgrey", lwd = .5, mar = c(0.1,0.1,0.1,0.1))
points(complete$lng, complete$lat, col = "black", cex = .8)
title("Locations of the 80 most populuous urban areas", line=1)
box()
```

Locations of the 80 most populuous urban areas

We could also provide a bubble chart of the urban regions (note that not all 80 observations actually appear):

```
library(ggplot2)
ggplot(complete,aes(x=urban.pop, y=urban.area, size=urban.den, colour=urban.den,
    label=city)) +
  geom_point() + theme_bw() + xlab("urban population") + scale_y_log10() +
  ylab("urban area (log scaled)") + guides(colour = guide_legend(reverse = TRUE)) +
  geom_text(aes(label=ifelse(urban.pop>23000000, as.character(city),'')),
            size=3, hjust=1, vjust=2) +
  labs(colour = "urban density", size = "urban density") +
  ggtitle("Relationship between population, area, and density",
          subtitle="80 most populous urban areas")
```


16.4.2 Weather Data

Simple web requests sometimes require some networking code to fetch a page and return the HTML contents.

When a user is extracting data from HTML, they typically search for certain HTML elements and read the data they contain (data could be text, tables, form field values, images, videos, etc.).

In this example, we build a function that extracts the 7-day forecast for the main Canadian cities location.

Preamble

The first step is to load the various required Python modules.

```
from bs4 import BeautifulSoup      # to parse and process the data
from urllib.request import urlopen # to open URLs
import numpy as np                 # to do some numerics eventually
import pandas                      # to do some analysis eventually
```

Loading and Parsing the Data

We get a handle on the website structure by studying the page for a single location, say Ottawa, Ontario ⧉ .[25]

```
ottawaURL = "https://weather.gc.ca/city/pages/on-118_metric_e.html"
```

The page looks something like the image in Figure 16.11.

Ottawa (Kanata - Orléans), ON

Figure 16.11: 7-day forecast for Ottawa, ON, on Wednesday April 6, 2022. [weather.gc.ca ⧉]

26: Other parsers can also be used, depending on the type of files with which we work.

We download the HTML and load it into *BeautifulSoup*, using `html.parser`.[26]

```
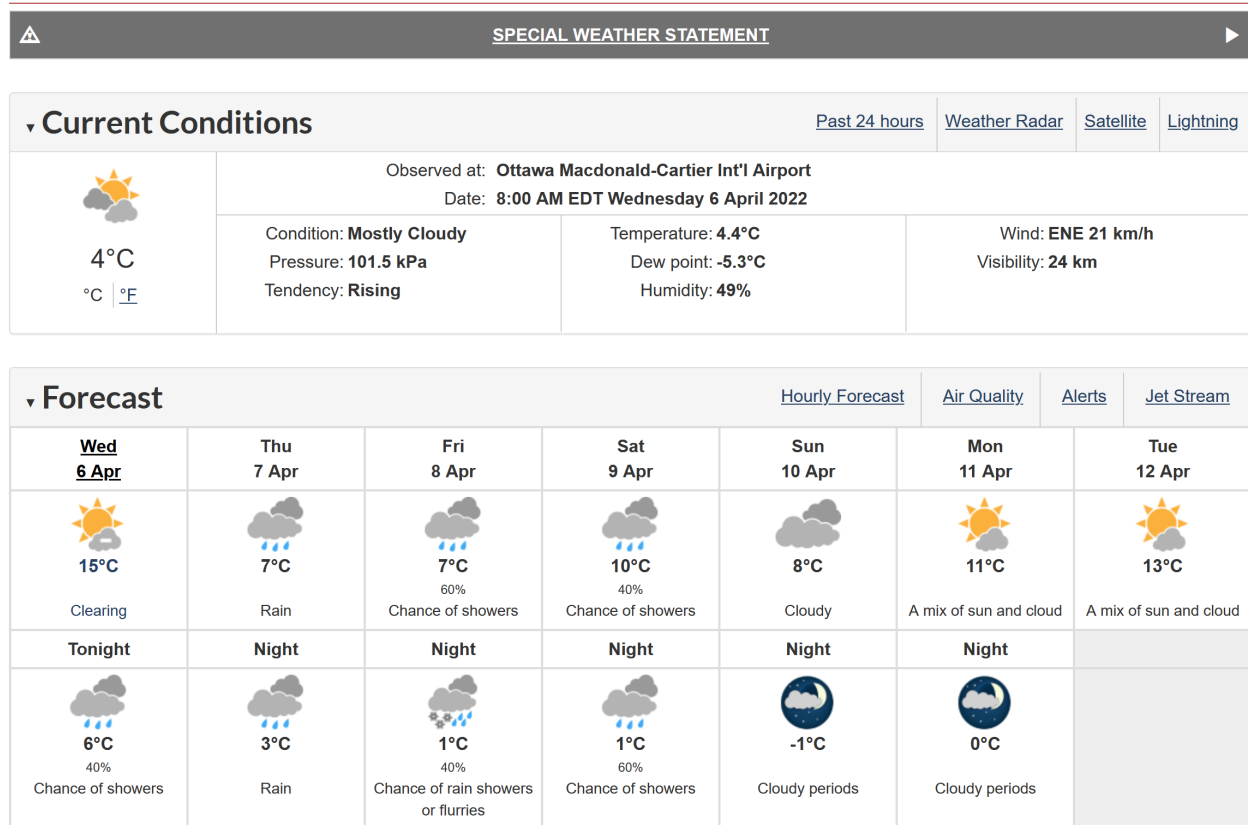ottawaHTML = urlopen(ottawaURL)
ottawaBS = BeautifulSoup(ottawaHTML, 'html.parser')
```

The soup (parsed content) is now available in `ottawaBS`. The data of interest is in there, we just need to pick it out of the document.

If we open developer tools pane in our browser, we can examine the specific HTML elements that contain the numbers we want. The table with the 7 day forecast appears to correspond to `div` element with `class=div-table` (see Figure 16.12); the weather information is contained in 7 columns, each of which is a `div` element with `class=div-column` (see Figure 16.13).

Figure 16.12: 7-day forecast for Ottawa, ON, on Wednesday April 6, 2022; the 'div' element with 'class=div-table' is highlighted in the Firefox Inspector.

Figure 16.13: 7-day forecast for Ottawa, ON, on Wednesday April 6, 2022; the 'div' element with 'class=div-column' is highlighted in the Firefox Inspector.

We can find it in the soup `ottawaBS` as follows:

```python
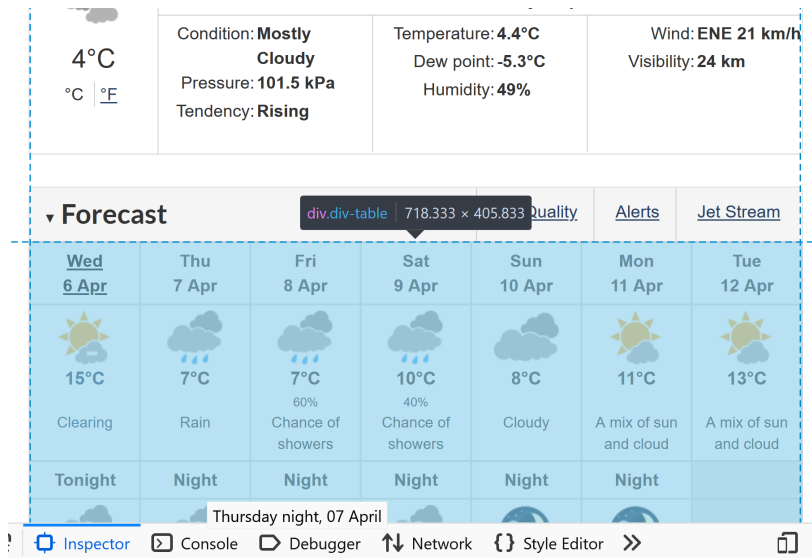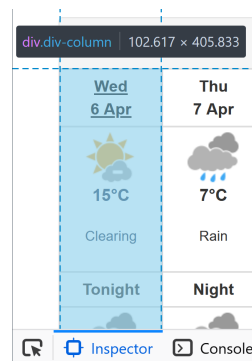sevenDaysBS = ottawaBS.find_all('div', attrs={"class" : "div-column"})
```

We display the HTML for the first of those columns below.

```python
print(sevenDaysBS[0].prettify())
```

```html
<div class="div-column">
 <div class="div-row div-row1 div-row-head">
  <a href="/forecast/hourly/on-118_metric_e.html">
   <strong title="Friday">
    Fri
   </strong>
   <br/>
   7
   <abbr title="October">
    Oct
   </abbr>
  </a>
 </div>
</div>
```

```
<a class="linkdate" href="/forecast/hourly/on-118_metric_e.html">
<div class="div-row div-row2 div-row-data">
 <img alt="Chance of showers" class="center-block" height="51" src="/weathericons/12.gif" width="60"/>
 <p class="mrgn-bttm-0">
  <span class="high wxo-metric-hide" title="max">
   10°
   <abbr title="Celsius">
    C
   </abbr>
   <span class="abnTrend">
    *
   </span>
  </span>
  <span class="high wxo-imperial-hide wxo-city-hidden" title="max">
   50°
   <abbr title="Fahrenheit">
    F
   </abbr>
   <span class="abnTrend">
    *
   </span>
  </span>
 </p>
 <p class="mrgn-bttm-0 pop text-center" title="Chance of Precipitation">
  <small>
   60%
  </small>
 </p>
 <p class="mrgn-bttm-0">
  Chance of showers
 </p>
</div>
</a>
<div class="div-row div-row3 div-row-head">
 Tonight
</div>
<div class="div-row div-row4 div-row-data">
 <img alt="Mainly cloudy" class="center-block" height="51" src="/weathericons/33.gif" width="60"/>
 <p class="mrgn-bttm-0">
  <span class="low wxo-metric-hide" title="min">
   0°
   <abbr title="Celsius">
    C
   </abbr>
  </span>
  <span class="low wxo-imperial-hide wxo-city-hidden" title="min">
   32°
   <abbr title="Fahrenheit">
    F
   </abbr>
  </span>
 </p>
 <p class="mrgn-bttm-0 pop text-center">
 </p>
 <p class="mrgn-bttm-0">
  Mainly cloudy
 </p>
</div>
```

In each of the columns, the first row contains the date (see Figure 16.14), and the second the maximum forecast temperature during the day (see Figure 16.14).[27]

Figure 16.14: 7-day forecast for Ottawa, ON, on Wednesday April 6, 2022; the 'div' element with 'class=div-row1' is highlighted in the Firefox Inspector.

Figure 16.15: 7-day forecast for Ottawa, ON, on Wednesday April 6, 2022; the 'div' element with 'class=div-row2' is highlighted in the Firefox Inspector.

We can extract the strings in each of the first two cells of the first column using the `.strings` method, as below:

```
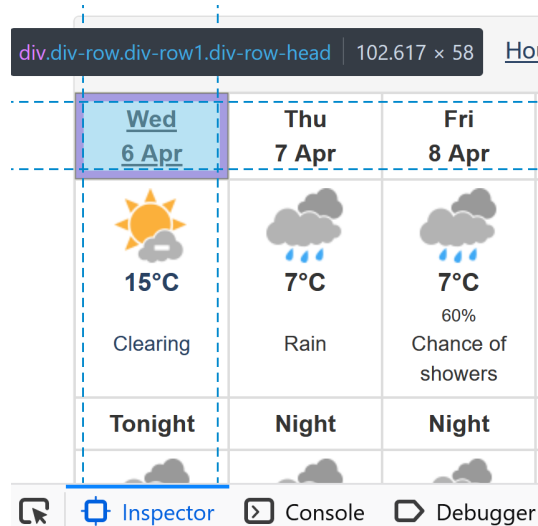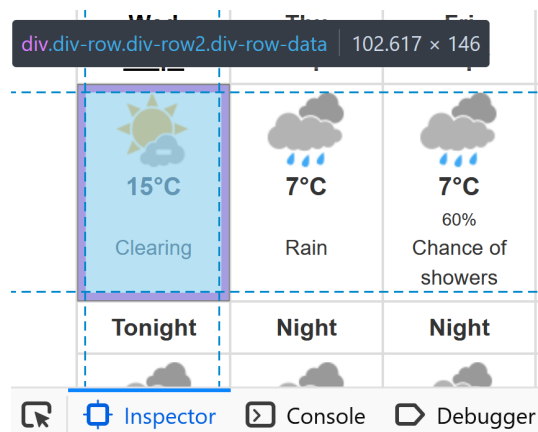# date
list(sevenDaysBS[0].find(class_="div-row div-row1 div-row-head").strings)
# temp
list(sevenDaysBS[0].find(class_="high wxo-metric-hide").strings)
```

```
['Fri', '7\xa0', 'Oct']
['10°', 'C', '*']
```

The lists contains the information of interest, together with additional characters; for both variables, we join the list elements into a single string and remove the odd characters (°C, \xa0,*), using the `.replace()` method.

```
' '.join(list(sevenDaysBS[0].find(class_="div-row div-row1 div-row-head").strings))
    .replace("\xa0","").replace("*","")
''.join(list(sevenDaysBS[0].find(class_="high wxo-metric-hide").strings))
    .replace("°C","").replace("*","")
```

```
'Fri 7 Oct'
'10'
```

Based on this work, we now write functions that extract a 7-day forecast, the corresponding dates, the city name, and the province code given a URL of the right format.[28]

```
def sevenDayForecast(url):
    html = urlopen(url)
    htmlBS = BeautifulSoup(html, 'html.parser')
    sevenDaysBS = htmlBS.find_all('div', attrs={"class" : "div-column"})
    temp_degree = []
    for day in sevenDaysBS:
        temp_de = int(
            ''.join(list(day.find(class_="high wxo-metric-hide").strings)).replace("°C","")
              .replace("*","")
                    )
        temp_degree.append(temp_de)
    return temp_degree

def sevenDayForecastDates(url):
    html = urlopen(url)
    htmlBS = BeautifulSoup(html, 'html.parser')
    sevenDaysBS = htmlBS.find_all('div', attrs={"class" : "div-column"})
    temp_date = []
    for day in sevenDaysBS:
        temp_da = ' '.join(list(day.find(class_="div-row div-row1 div-row-head").strings))
            .replace("\xa0","").replace("\n ","").replace(" \n","").replace("*","")
        temp_date.append(temp_da)
    return temp_date

def cityName(url):
    html = urlopen(url)
    htmlBS = BeautifulSoup(html, 'html.parser')
    nameBS = htmlBS.find('h1', attrs={"property" : "name"})
    city_name = list(nameBS.strings)[0].replace(" \n","").replace(", ","")
    return city_name

def provinceCode(url):
    html = urlopen(url)
    htmlBS = BeautifulSoup(html, 'html.parser')
    nameBS = htmlBS.find('h1', attrs={"property" : "name"})
    province_code = list(nameBS.strings)[1]
    return province_code
```

We validated the functions on the Ottawa URL, on Oct 7, 2022.[29]

```
sevenDayForecast(ottawaURL)
sevenDayForecastDates(ottawaURL)
cityName(ottawaURL)
provinceCode(ottawaURL)
```

```
[10, 11, 13, 12, 14, 17, 15]
['Fri 7 Oct', 'Sat 8 Oct', 'Sun 9 Oct', 'Mon 10 Oct', 'Tue 11 Oct', 'Wed 12 Oct', 'Thu 13 Oct']
'Ottawa (Kanata - Orléans)'
'ON'
```

Data Processing

We now prepare the data for analysis. We select the 20 Canadian cities that appear on the website's main page ⬚ .

For each of these cities, we extract the 7-day forecast, and display "today's" temperature, "tomorrow's" prediction, the weekly change 1 week from "today", and the mean prediction over the 7-day forecast.

This could be done manually by feeding the URL to each of the 4 functions defined above (in the example for Ottawa), but we will use *BeautifulSoup* to scrape the information automatically (and cleanly).[30] We start by finding the URL for each of the cities on the main page.

```
wURL = "https://weather.gc.ca/canada_e.html"
wHTML = urlopen(wURL)
wBS = BeautifulSoup(wHTML, 'html.parser')
tableBS = wBS.find('table', attrs={"class" : "table
    table-hover table-striped table-condensed"})
citiesBS = tableBS.find_all('a', href=True)

citiesFURLs = []
for a in citiesBS:
    temp = a['href']
    citiesFURLs.append(temp)

citiesURLs = ["https://weather.gc.ca" + citiesFURLs[index]
    for index in range(len(citiesFURLs))]
```

Next we build a dictionary containing the desired data, for each city:[31]

```
today_date = sevenDayForecastDates(citiesURLs[0])[0]

row_dict = []
for row in range(len(citiesURLs)):
    d = dict()
    tmp = sevenDayForecast(citiesURLs[row])
    d['city'] = cityName(citiesURLs[row])
    d['province'] = provinceCode(citiesURLs[row])
    d['date'] = today_date
    d['today'] = tmp[0]
```

```
d['tomorrow'] = tmp[1]
d['1 week change'] = np.subtract(tmp[6],tmp[0])
d['weekly mean'] = np.mean(tmp)
row_dict.append(d)
```

Finally, we covert the dictionary into a pandas data frame:

```
wDF = pandas.DataFrame(row_dict)
wDF
```

```
                        city province  ... 1 week change   weekly mean
0                       Calgary        AB  ...              6     20.000000
1                   Charlottetown    PE  ...             -3     14.142857
2                       Edmonton       AB  ...              0     20.285714
3                      Fredericton     NB  ...             -4     15.571429
4                        Halifax        NS  ...             -2     15.285714
5                        Iqaluit        NU  ...              2     -1.142857
6                       Montréal        QC  ...              0     14.571429
7     Ottawa (Kanata - Orléans)    ON  ...              5     13.142857
8                   Prince George    BC  ...             -2     16.142857
9                         Québec       QC  ...             -4     12.714286
10                        Regina       SK  ...              3     18.428571
11                     Saskatoon      SK  ...              0     19.000000
12                    St. John's     NL  ...             -2     12.714286
13                   Thunder Bay     ON  ...              2     11.857143
14                       Toronto       ON  ...              4     14.857143
15                     Vancouver      BC  ...             -2     18.571429
16                      Victoria       BC  ...             -2     20.285714
17                    Whitehorse     YT  ...            -13     10.571429
18                      Winnipeg       MB  ...              3     14.285714
19                   Yellowknife     NT  ...             -9      7.571429

[20 rows x 7 columns]
```

Visualization

As a last exercise, we provide a basic visualization for the collected dataset.

```
import seaborn as sns
sns.set(style='whitegrid')
```

We use seaborn's pairplot() to produce the scatterplot matrix of the data, and matplotlib's plt() to display it.

```
cols = ['today', 'tomorrow', '1 week change',
    'weekly mean']
sns.pairplot(data=wDF[cols], size=2.5)
```
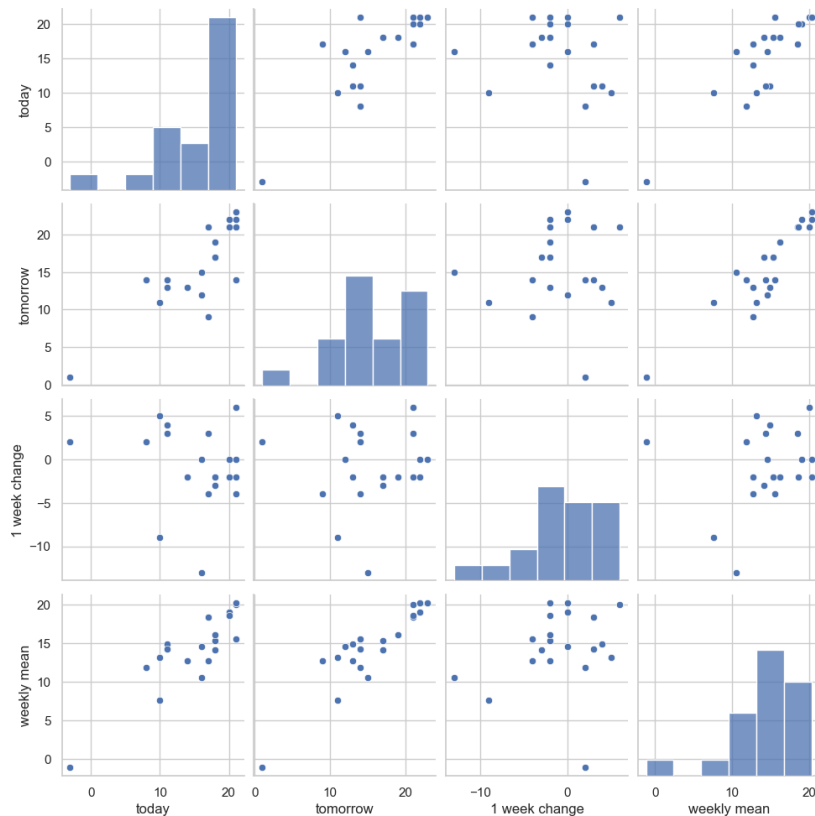
Perhaps unsurprisingly, there does not seem to be much insight available in the dataset. If there is an association between tomorrow's prediction and the prediction one week from now, we require more information to explore it; data collected on a daily basis, perhaps?[32]

That is an important point to keep in mind: the process is sometimes long and complicated, but that **does not always translate into insight at the end of the day**.[33]

16.4.3 CFL Play-by-Play

In this example, we obtain **structured play-by-play** data for past CFL.[34] games. We could use this information to ask questions such as:

- how often do teams convert on 3rd and X?
- do teams come back from 7+ pt deficits in the 4th quarter?
- etc.

Preamble

Before you start, make sure that *BeautifulSoup*, Selenium, Pandas, Firefox, and Geckodriver are installed in your Python environment. You can use the code below to install the Python modules.

- `pip3 install beautifulsoup4`
- `pip3 install pandas`
- `pip3 install selenium`

32: In which case, it would be useful to save the data; how would this be accomplished?

33: Unless the absence of an apparent link is insight... which it could very well be, in certain cases.

34: Canadian Football League.

You can get download information for Firefox and Geckodriver here:

- Firefox ⬀
- Geckodriver ⬀

Of course, other browsers have their own installation information. We will use the following Python modules for pulling data out of HTML and XML files (BeautifulSoup), for dealing with potentially dynamic websites (Selenium), to open URLs (urllib.request), and other regular tasks.

```python
from bs4 import BeautifulSoup
from pyvirtualdisplay import Display
from selenium import webdriver
from urllib.request import urlopen
import csv
import pandas
import time
import warnings; warnings.filterwarnings('ignore')
```

Game Schedule

Let us start by getting a list of all games in a season; we will switch to processing data on a game-by-game basis at a later stage. All games in a season (2016, say) are listed at a single URL in the following format.

```python
year = 2016
scheduleURL = 'https://www.cfl.ca/schedule/?season={}'
    .format(year)
```

This produces the following URL:

```python
scheduleURL
```

```
'https://www.cfl.ca/schedule/?season=2016'
```

Now we open the schedule page and parse it with *BeautifulSoup*:

```python
scheduleHTML = urlopen(scheduleURL)
scheduleBS = BeautifulSoup(scheduleHTML, 'html.parser')
```

We could display the HTML code with:

```python
scheduleBS
```

Warning: the HTML file contains a lot of information, so the display has been suppressed. For completness' sake, when rendered in a browser, the page looks like the image in Figure 16.16.

Figure 16.16: Extract of the 2016 CFL schedule and results cfl.ca .

We could sift through the HTML to try to find what each piece of code corresponds to on , but that is not the most efficient approach to use.

Instead, we use the Developer Tools to get a better idea. In the example below, let's say we are interested in the second pre-season game.[35]

35: A 25-16 victory by the Toronto Argonauts over the Hamilton Tiger-Cats.

Right-click on the box containing the game information, and select "Inspect Element (Q)" from the menu that appears. In the Developer Tool, you will be taken to the section of HTML code corresponding to the element you selected.[36]

36: We might need to try right-clicking over a few locations as there sub-elements in the game box.

Each game is represented by a row. According to developer tools, these rows are `div` elements with the class `heading collapsible-header`.[37]

37: See Figure 16.17.

Figure 16.17: CFL 2016 schedule and results; the 'div' element with 'heading collapsible-header' is highlighted in the Firefox Inspector.

Scrolling down on the schedule page, it appears as though every game is presented in the same format, so it is worth a shot to ask *Beautiful Soup* to find all rows that contain the class `heading collapsible-header`.[38]

38: Note the _ after `class`, and the single quotes.

```
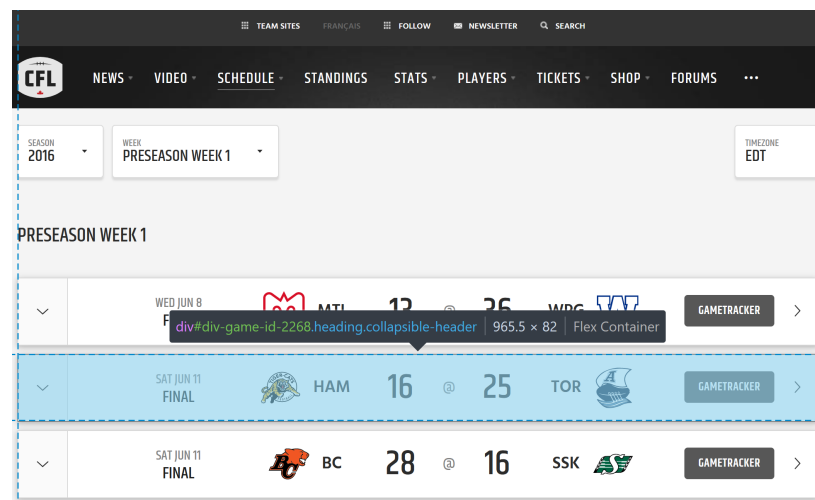scheduleRows = scheduleBS.findAll(class_='heading
    collapsible-header')
```

Here's a better view of a single row, with some parts omitted:

```
<div id="div-game-id-2268" class="heading collapsible-header">
  <div class="controls">
    ...
  </div>
  <div class="sponsored">
    ...
  </div>
  <div class="date-time">
    ...
  </div>
  <div class="matchup">
    ...
  </div>

  <div class="action">
    <a
      href="javascript:void(0);"
      data-url="https://www.cfl.ca/games/2268/hamilton-tiger-cats-vs-toronto-argonauts/"
      class="gametracker">
      <span class="btn">Gametracker</span>
    </a>
  </div>
</div>
```

We want the URL that the "GAMETRACKER" button links to – this is
the game page that contains the play-by-play info. The link is found in
the `data-url` attribute, rather than in the `href` attribute. We can get the
link for the 2nd pre-season game by querying `scheduleRows[1]`.[39]

39: Recall that list indexing starts with 0
in `Python`.

```
row = scheduleRows[1]
button = row.find(class_='gametracker')
button['data-url']
```

`'https://www.cfl.ca/games/2268/hamilton-tiger-cats-vs-toronto-argonauts/'`

These are all the steps we need to get the list of game page URLs for an
entire season.

We might also want to store each of these game pages in a Python array.
This can be done as follows.

```
urls = []

for row in scheduleRows:
    button = row.find(class_='gametracker')
    url = button['data-url']
    urls.append(url)

# uncomment to display the URLs
# print(urls)

df = pandas.DataFrame(urls)
df.to_csv(path_or_buf='Data/CFL_Schedule_2016.csv',
    header=False)
```

Incidentally, how many games were played in total in 2016, including the
pre-season and the playoffs?

That is easy to answer:

```
len(urls)
```

95

Scraping Game Data

Here is a URL for one particular game.

```
gameURL = 'https://www.cfl.ca/games/2391/ottawa-redblacks-vs-toronto-argonauts'
```

The screenshot of Figure 16.18 shows the page as it is rendered in the
browser **after** clicking the "PLAY BY PLAY" button.

Figure 16.18: Play-by-play data for the July 24, 2016 CFL game between the Ottawa Redblacks and the Toronto Argonauts cfl.ca ⬀ .

The page actually only loads the play-by-play data once the "PLAY BY PLAY'' button is pressed. If we download the HTML before pressing the button, the data just **isn't there**.

```
gameBS = BeautifulSoup(urlopen(gameURL))
gameBS.text.count('Kickoff')
```

```
0
```

The page does contain JavaScript code that tells the browser to fetch more data when the button is clicked and add it to the page. The most straightforward way to get this data is to run a browser but control it automatically. All we need is a way to identify the button to press.
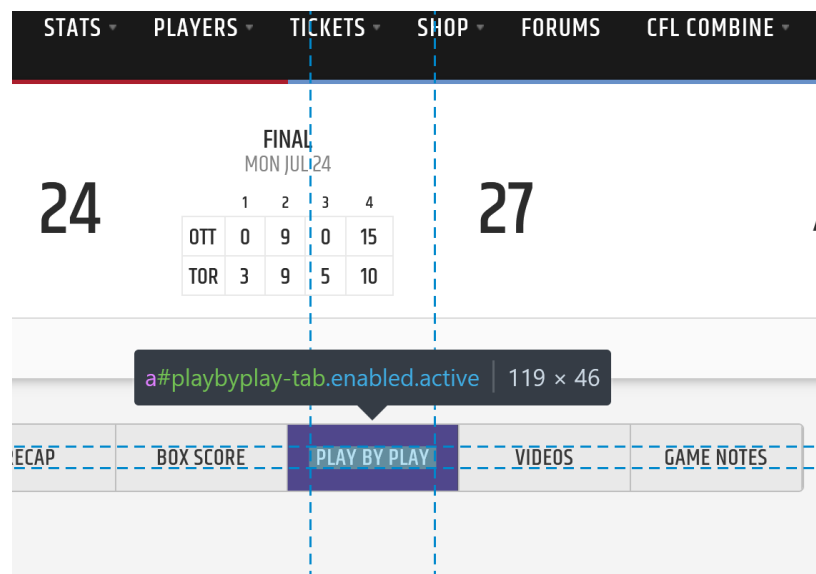
Figure 16.19: Play-by-play data for the July 24, 2016 CFL game between the Ottawa Redblacks and the Toronto Argonauts; the 'div' element with 'playbyplay-tab' is highlighted in the Firefox Inspector.

Luckily the button has a (unique) id (see Figure 16.19), so we can use that. We define an XPATH string for that id.

```
pbp_btn_xpath = '//*[@id="playbyplay-tab"]'
```

For browser automation, we use Firefox with Selenium – it is important to ensure that `geckodriver` is installed (or whatever the appropriate driver is for the browser in use).

In the next block, we run code for the `driver` object (in this case, Selenium controlling Firefox), telling it to load the page, click the button, and then get the HTML. Depending on the system, the variable `executable_path` will vary.

```
display = Display(visible=0, size=(1440, 1080))
display.start()
driver = webdriver.Firefox(executable_path='/usr/local/bin/geckodriver')

# Open the page
driver.get(gameURL)

# Wait for loading
time.sleep(5)
# less about robots.txt but more about content "physically" being there

# Click button to get play-by-play data
playbyplay_btn = driver.find_element_by_xpath(pbp_btn_xpath)
playbyplay_btn.click()

# Wait again for loading
time.sleep(5)

# Take HTML and save in BS object
soup = BeautifulSoup(driver.page_source)
driver.close()
```

The URL of the loaded play-by-play page can be loaded and parsed into a into a soup.

```
pbpURL = 'https://www.cfl.ca/games/2391/ottawa-redblacks-vs-toronto-argonauts#playbyplay'
pbpHTML = urlopen(pbpURL)
soup = BeautifulSoup(pbpHTML, 'html.parser')
```

Now that we have the HTML of the loaded page, we can extract data as usual with Beautiful Soup, such as finding the home team and the away teams, and so on.

```
# away, home
[soup.find(class_='js-data-team_2_location').text,
 soup.find(class_='js-data-team_1_location').text]
```

```
['Ottawa', 'Toronto']
```

16.4.4 Bad HTML

When we write a R/Python program with incorrect syntax, we get an error and our program does not work. If we write an HTML page with incorrect syntax, there's a good chance that browsers will be able to make sense of it anyway – browsers **try to guess** ways to correct each error.

We can check whether a webpage on the internet uses correct syntax or not by entering the URL at validator.w3.org ⧉ .

If what we see in our browsers is a fixed-up version of the HTML, then when we parse HTML with Python we'd like to be able to get a similarly fixed-up version. We look at some simple examples of how *Beautiful Soup* handles bad HTML.

```
from bs4 import BeautifulSoup
```

First we pass *Beautiful Soup* a proper (incomplete) HTML document:

```
goodBS = BeautifulSoup(
    '<html><head><title>blah</title></head><body></body>
    </html>', 'html.parser')
```

As expected, we can operate with the parsed document, such as finding elements and getting their data.

```
goodBS.find('title').text
```

```
'blah'
```

Now what if we omit the closing `</title>` tag? We print the corrected version that *BeautifulSoup* builds.

```
badBS = BeautifulSoup(
    '<html><head><title>blah</head><body></body></html>',
    'html.parser')
print(badBS)
```

```
<html><head><title>blah</title></head><body></body></html>
```

You see that the closing tag has been returned. Similar behaviour is seen in the following examples where tags are misplaced or omitted. Note that although `<li>` (list item) tags are supposed to be put inside a list tag such as `<ul>` (unordered list) or `<ol>` (ordered list), *Beautiful Soup* doesn't add those tags.

```
badBS = BeautifulSoup(
    '<html><head></head><body><li><em>hi</body></em></html>',
    'html.parser')
print(badBS.prettify())
```

```
<html>
<html>
 <head>
 </head>
 <body>
  <li>
   <em>
    hi
   </em>
  </li>
 </body>
</html>
```

```
badBS = BeautifulSoup(
   '<html><head></head><body><li><em>hi<li></body></em>
   </html>','html.parser')
print(badBS.prettify())
```

```
<html>
 <head>
 </head>
 <body>
  <li>
   <em>
    hi
    <li>
    </li>
   </em>
  </li>
 </body>
</html>
```

In general, if the browser can do a good enough job to render an HTML page as intended, we can trust *BeautifulSoup* to fix things up **logically**. But when we automate the data collection process, we do not usually visit each page before it is scraped; there might be surprises in store!

16.4.5 Extracting Text from a PDF File

Apache Tika ⬈ can be used to convert PDF files to TXT files, but a few R libraries can also do so.[40] We use the `pdftools` library to extract text from the DAL Data Visualization Learning Map ⬈ .

40: And are potentially easier to use, depending on the document's structure.

```
library(pdftools)
DAL <- pdf_text("https://www.data-action-lab.com/wp-content/uploads/2020/01/
    Learning-Map-Data-Visualization-ACFO.pdf")
length(DAL)
N <- 1:length(DAL)
```

```
[1] 7
```

What does the first page look like, say?

```
DAL[1]
```

```
[1] "
DATA\n      DATA EXPLORATION AND DATA VISUALIZATION\n
ACTION\n
LAB\n\n\n\n\n                                    Data
Analytics\n                               Conference -
ACFO\n                          Data Exploration
and Data\n
Visualization\n\n
Learning Map\n\n\nab\n\n\n\n\n
Provided by the Data Action Lab\n    Canada School of Public
Service   Digital Academy\n"
```

It us quite a mess – the text contains a fair amount of "noise" (we will revisit text cleaning in detail in Chapter 27, *Text Analysis and Text Mining*).

```
library(stringr)
library(dplyr)

removeDiacritics <- function(string) {
  chartr(
    "SZszYÀÁÂÃÄÅÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖÙÚÛÜÝýÿ",
    "SZszYAAAAAACEEEEIIIIDNOOOOOUUUUYaaaaaaceeeeiiiidnooooouuuuyy",
    string
  )
}

DAL.clean=c()

for(i in 1:length(DAL)){
  DAL.clean[i] <- DAL[i] %>%
    { gsub("\n", " ", .) } %>%
    { gsub("\\d", " ", .) } %>%
    { gsub("\\s+", " ", .) } %>%
    trimws()
}
```

We might keep only those pages with at least 1000 characters, say:

```
index <- nchar(DAL.clean)>1000
DAL.clean <- DAL.clean[index]
length(DAL.clean)
```

```
[1] 6
```

Only 6 of the pages contain at least 1000 characters. Let us take a look at one of them, the last one, say.

```
DAL.clean[6]
```

[1] "DAL instructors have consulted for DATA EXPLORATION (and taught to participants from) a
AND DATA VISUALIZATION variety of groups, a selection of which is shown below: § Canada Revenue
Agency § Canada School of Public Service's Digital Academy § Canadian Air Transport Security
Authority § Canadian Coast Guard § Canadian Food Inspection Agency § Canadian Institute for
Health Information § The Children's Hospital of Eastern Ontario § Communications Research
Centre Canada § Department of National Defence § Environment and Climate Change Canada
§ Fisheries and Ocean Canada § Health Canada § Immigration, Refugees and Citizenship
Canada § Indigenous and Northern Affairs Canada § Natural Resources Canada § Nuclear Waste
Management Organization § Office of the Privacy Commissioner of Canada § Privy Council Office
§ Public Services and Procurement Canada § Royal Canadian Mounted Police § Transport Canada
§ Treasury Board Secretariat Consult our Data Training Catalogues for a list of practical data
analysis and data leadership courses. Visit data-action-lab.com or contact info@data-action-lab.com
for more information. DATA ACTION LAB | info@data-action-lab.com"

Not too shabby, eh? It's almost readable, even!

16.4.6 YouTube Video Titles

In this example, we will see how to use the YouTube API to scrape the
titles of YouTube videos.

```python
from apiclient.discovery import build
from apiclient.errors import HttpError
from oauth2client.tools import argparser

# some of these will only be useful for the exerices
import pandas
from functional import seq
import codecs
import glob
import html
import os
import re
import unidecode
import urllib
import urllib.request
import warnings; warnings.filterwarnings('ignore')
```

Authentication

The task is to build or add to a corpus of text by fetching video transcripts
from YouTube. To use the YouTube API, we need to authenticate ourselves.
Create a config.json file in the main directory, whose only content looks
like this:

```
{  "DEVELOPER_KEY":    "your_key_here"  }
```

Instructions on how to obtain a key are provided here ⬁

Once that done, we create an object youtube, through which we can access YouTube API methods (more information is available at Wikipedia ⬁ , YouTube I ⬁ , YouTube II ⬁).

```
config = seq.json('config.json').dict()


DEVELOPER_KEY = config['DEVELOPER_KEY']
YOUTUBE_API_SERVICE_NAME = "youtube"
YOUTUBE_API_VERSION = "v3"


youtube = build(YOUTUBE_API_SERVICE_NAME,
                YOUTUBE_API_VERSION,
                developerKey=DEVELOPER_KEY)
```

YouTube API

We could hand-pick videos to read, but we will take a shortcut by getting all the transcripts in a whole playlist of videos.

The first task is taking a playlist ID and using the API to get the video IDs of each entry in the playlist. The API for getting entries in a playlist is **paginated**. This means that we have to make one request for the first chunk of entries, then make another request to get some more entries, and so on until we have got the entire playlist.

It's designed this way so that we don't download more than we need; for example if we were building an infinite scrolling menu, we wouldn't want to load everything up front.

41: In this example, 10 videos at a time.

After we have obtained the first chunk,[41] we need to tell the API where to start the next chunk. This is done using a **page token**.

We take the nextPageToken of the response we get, and pass it to the API for the next request, until the API returns no nextPageToken value.

```
def fetch_playlist_videos(playlistId):
    '''
    get all videos in a playlist.
    Returns: list of dictionaries representing
        playlistItem resources,
    see https://developers.google.com/youtube/v3/docs
        /playlistItems#resource-representation
    for the structure of this resource
    '''

    # API method: https://developers.google.com/youtube/
    #    v3/docs/playlistItems/list
    res = youtube.playlistItems().list(
        part="snippet",
        playlistId=playlistId,
        maxResults="10").execute()
```

```
    nextPageToken = res.get('nextPageToken')
    while ('nextPageToken' in res):
        nextPage = youtube.playlistItems().list(
            part="snippet",
            playlistId=playlistId,
            maxResults="10",
            pageToken=nextPageToken).execute()
        res['items'] = res['items'] + nextPage['items']

        if 'nextPageToken' not in nextPage:
            res.pop('nextPageToken', None)
        else:
            nextPageToken = nextPage['nextPageToken']


    return res['items']
```

Playlist Extraction

The playlist entries come in the form of `playlistItem` **resource dictionaries**.[42] In the `Python` API, the object is a **nested dictionary**. We want to get to the video ID, which we will do for all playlist items.

First, take the time to explore the following YouTube playlist: Introduction to Quantitative Consulting ⧉ .

42: A data format defined in the API documentation that contains fields for all the information associated with an item in a playlist.

Figure 16.20: Introduction to Quantiative Consulting YouTube playlist.

Next, we build the list of videos.

```
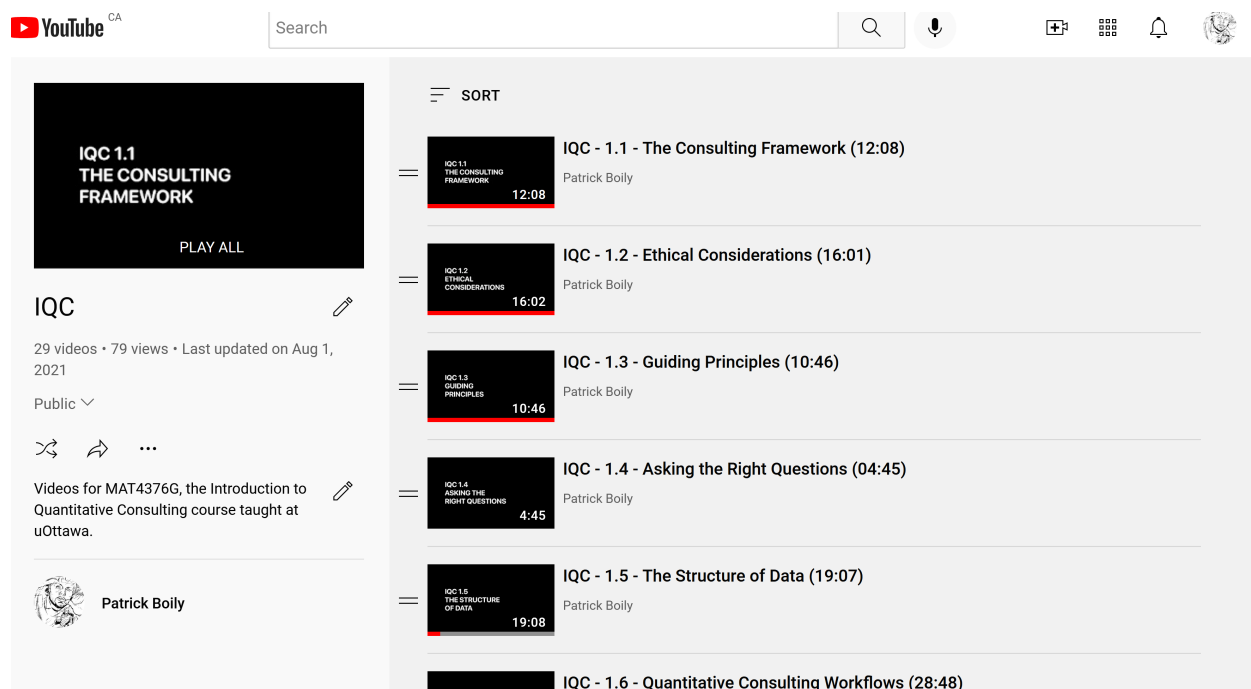# some playlists with English transcripts available
IQCPlaylist = ['PLbVTnkp2K536WxfoqSvoY08aJ3sLBg9mI']
```

```
videos = []
for playlistID in IQCPlaylist:
    videos += fetch_playlist_videos(playlistID)
```

We can explore the list by looking at the 3rd video in the playlist, say.

```
print(videos[2])
```

```
{'kind': 'youtube#playlistItem',
 'etag': 'LYF0Osi6rdPvtzVAqAY-TobYZnE',
 'id': 'UExiVlRua3AySzUzNld4Zm9xU3ZvWTA4YUozc0xCZzltSS4xMkVGQjNCMUM1N0RFNEUx',
 'snippet': {'publishedAt': '2020-06-20T21:18:23Z',
 'channelId': 'UCIi6fq-A7sTT4iBDQUKekyg',
 'title': 'IQC - 1.3 - Guiding Principles (10:46)',
 'description': '1.3.1 Best Practices\n1.3.2 The Good, the Bad, and the Ugly',
 'thumbnails':
    {'default':
        {'url': 'https://i.ytimg.com/vi/eodNQzJFJpg/default.jpg', 'width': 120, 'height': 90},
    'medium':
        {'url': 'https://i.ytimg.com/vi/eodNQzJFJpg/mqdefault.jpg', 'width': 320, 'height': 180},
    'high':
        {'url': 'https://i.ytimg.com/vi/eodNQzJFJpg/hqdefault.jpg', 'width': 480, 'height': 360},
    'standard':
        {'url': 'https://i.ytimg.com/vi/eodNQzJFJpg/sddefault.jpg', 'width': 640, 'height': 480},
    'maxres':
        {'url': 'https://i.ytimg.com/vi/eodNQzJFJpg/maxresdefault.jpg', 'width': 1280, 'height': 720}},
 'channelTitle': 'Patrick Boily',
 'playlistId': 'PLbVTnkp2K536WxfoqSvoY08aJ3sLBg9mI',
 'position': 2,
 'resourceId': {'kind': 'youtube#video', 'videoId': 'eodNQzJFJpg'},
 'videoOwnerChannelTitle': 'Patrick Boily',
 'videoOwnerChannelId': 'UCIi6fq-A7sTT4iBDQUKekyg'}}
```

We get list of all video IDs and their titles as follows.

```
videoIDs = [ video['snippet']['resourceId']['videoId'] for video in videos ]
videotitles = [ video['snippet']['title'] for video in videos ]
print(videoIDs)
```

```
['-dZImvCSPKI', '0vBXkgiJIP8', 'eodNQzJFJpg', 'IiQJ1G4QJWg', 'ycBovk3EtfQ', 'RErsLHdKFSM',
 '5eu_FoJu7uo', 'HUzosM19QCs', 'n4Z3SgEJ4bg', 'P-jkx_XdJlw', 'LFS6RbpzLSw', 'OhrtH6sGbtA',
 'bIO4JmGVf_k', 'LUU_UKk2YyQ', 'dgtapT4n484', '1cRmNcT1pvo', 'Ga6VEPk_HfY', 'Q2o8bIV6328',
 '-ZLuiE0j8Ts', 'WqTH3OvPKxQ', '_9eUuc_-z9s', 'ITGBjuOwY4w', 'cQvCq1_Eoms', 'erP8Xc0hO0U',
 'mb7p4B2spP0', 'KG4SBzXccEk', 'A9Wh4L7ZJr0', 'CL_cVCZ5l7Q', 'yxP4Nz09rSE']
```

We put this information into a dictionary:

```
yt_dict = []
for row in range(len(videoIDs)):
    d = dict()
    d['youtubeURL'] = 'https://youtu.be/{}'.format(videoIDs[row])
    d['title'] = videotitles[row]
    yt_dict.append(d)
```

It is now child's play to convert the dictionary to a Pandas dataframe:

```
ytDF = pandas.DataFrame(yt_dict)
ytDF
```

```
                    youtubeURL                                        title
0    https://youtu.be/-dZImvCSPKI      IQC - 1.1 - The Consulting Framework (12:08)
1    https://youtu.be/0vBXkgiJIP8      IQC - 1.2 - Ethical Considerations (16:01)
2    https://youtu.be/eodNQzJFJpg        IQC - 1.3 - Guiding Principles (10:46)
3    https://youtu.be/IiQJ1G4QJWg    IQC - 1.4 - Asking the Right Questions (04:45)
4    https://youtu.be/ycBovk3EtfQ       IQC - 1.5 - The Structure of Data (19:07)
5    https://youtu.be/RErsLHdKFSM  IQC - 1.6 - Quantitative Consulting Workflows ...
6    https://youtu.be/5eu_FoJu7uo     IQC - 1.7 - Roles and Responsibilities (14:24)

...

21   https://youtu.be/ITGBjuOwY4w              IQC - 2.11 - Invoicing (07:25)
22   https://youtu.be/cQvCq1_Eoms          IQC - 2.12 - Closing the File (03:43)
23   https://youtu.be/erP8Xc0hO0U  IQC - 3.1 - Lessons Learned: About Clients (19...
24   https://youtu.be/mb7p4B2spP0  IQC - 3.2 - Lessons Learned: About Consultants...
25   https://youtu.be/KG4SBzXccEk  IQC - 4.1 - The Basics of Business Development...
26   https://youtu.be/A9Wh4L7ZJr0         IQC - 4.2 - Clients and Choices (04:00)
27   https://youtu.be/CL_cVCZ5l7Q             IQC - 4.3 - Building Trust (10:33)
28   https://youtu.be/yxP4Nz09rSE           IQC - 4.4 - Improving Trust (09:04)
```

16.5 Exercises

In these exercises, use R's `rvest`, Python's *Beautiful Soup*, or any other tool (whether we discussed it or not in the main text) that will allow you to complete the task. You may need to look up various tutorials and examples, and consult documentation, Stack Overflow, and so on.

1. Complete the unanswered questions in Sections 28.3.2 (XPath) and 28.3.3 (regexp).
2. Recreate the web scraping example of Section 28.4.1, this time selecting (or creating) variables that will provide population and area values for all entries in the table (not necessarily variables `V11`, `V12`, `V13`). What changes? What stays the same? Why is that the case?
3. Web data is available from a variety of sources, in a variety of formats and languages. Your job is to build a collection of 5 text corpora, each one consisting of documents written in a different language (English, French, Spanish, Italian, and Other). The text documents will be collected from the New Zealand Government's press releases ⬀ , from Wikipedia, from twitter, from a PDF document, and from other sources. Your final dataset will consist of all of the observations (text) placed in rows, each row associated with a specific language code ("Eng", "Fra", "Esp", "Ita", "Oth").

   a) English: the text of all Canadian government press releases published in 2020.
   b) French: the text from the (French) Wikipedia entries of all French actresses whose last name starts with "L".
   c) Spanish: 700 tweets (total) from `@realmadrid`, `@PaulinaRubio`, `@Armada_esp` + 2 other tweeters of your choice.
   d) Italian: the text from Giovannino Guareschi's Tutto don Camillo (I racconti del Mondo piccolo) – Volume 1 di 5 (PDF), 1 page per row.
   e) Other: 500 other text documents, in other languages that use a Latin-based alphabet.

4. Use Zomato ⬀ to find which Canadian city has the best sushi restaurants.

5. Build a scraper that automatically collects a multiple-day forecast for all Canadian cities in the database (not only those found on the landing page ⬀), independently of the time at which the scraping takes place.

6. Consider the `parsed_doc` object from the XPath section. What do you think the following blocks of code do?

```
lowerCaseFun <- function(x) {
  x <- tolower(xmlValue(x))
  return(x)
}


XML::xpathSApply(parsed_doc, "//div//i",
    fun = lowerCaseFun)
```

```
dateFun <- function(x) {
  require(stringr)
  date <- xmlGetAttr(node = x, name = "date")
  year <- str_extract(date, "[0-9]{4}")
  return(year)
  }


XML::xpathSApply(parsed_doc, "//div", dateFun)
```

7. In the CFL example, the play-by-play data is in separate tables for each quarter. Write a routine that grabs the information and produces a Pandas dataframe for each quarter, with the following headers: `ID`, `away`, `details`, `down`, `home`, `quarter`, `time`, `type`, and `yard`.

8. Modify the YouTube example in order to extract the videos' captions. Clean them using *BeautifulSoup*.

9. Use `twitteR` (or other packages) to build a data frame of tweets related to the *Marvel Cinematic Universe*. Do your tweets mostly originate from Android or iPhone devices? Plot the frequency of tweets against time. Do the same for retweets. Do any patterns emerge?

10. Collect **all** Canadian government press releases ⬀ for the 2021 calendar year. Identify the date, emanating Department(s), and the number of characters in each release. Are there Departments who release news more frequently than others? Are there Departments whose releases are typically longer than average? What other insights can you draw from your data frame? Repeat this process with French-language press releases ⬀ .

11. Produce a data frame listing all new products available at David's Tea ⬀ , the page number where the product was listed, and its price. Remember the scraping do's and don't's!

Chapter References

[1]   *Beautiful Soup Documentation* ⬀ .

[2]   *HTML Cheat Sheet* ⬀ .

[3]   K. Jarmul. *Natural Language Processing Fundamentals in Python.*

[4]   M. Jones. *15 Fundamental Laws of the Internet* ⬀ .

[5]   R. Mitchell. *Web Scraping with Python: Collecting Data From the Modern Web.* 2nd. O'Reilly Media, 2018.

[6]   S. Munzert et al. *Automated Data Collection with R: A Practical Guide to Web Scraping and Text Mining.* 2nd. Wiley Publishing, 2015.

[7]   *Selenium Documentation* ⬀ .

[8]   *The Selenium Browser Automation Project* ⬀ .

[9]   R. Taracha. *Introduction to Web Scraping Using Selenium* ⬀ . 2017.