Reading Websites

Module 2 | Chapter 1 | Notebook 2

The internet offers a lot of data that we can use and process. Unfortunately, the data is rarely available in a structured format. We often have to access the HTML code of the pages directly. HTML parsers can help us with this. We'll get to know one of them better in this exercise. By the end of this lesson:

- You will know how HTML documents are structured
- You will be able to access HTML code with the `beautifulsoup` module

HMTL Basics

Scenario: The Taiwanese investor from *Module 1, Chapter 1* gets in touch with you again. This time he's not interested in house prices. Instead, he wants to invest in DAX-listed companies. However, he doesn't yet have enough data on the companies to make an informed decision. So he asks you to collect publicly available data on the companies and to deliver it to him in a structured format.

To find out which DAX companies could be a lucrative investment, we first need to know which companies are actually listed on the DAX. This was why we looked at the relevant Wikipedia page *https://en.wikipedia.org/wiki/DAX*. The details of the DAX companies are listed in a table. We would like to isolate and store this data before we move on to more company data later. In the last lesson, you accessed and looked at the website's HTML code. We'll need the HTML code again in this lesson. So import `requests` and store the page address in the variable `website_url`.

In [1]:
```
import requests
website_url  = 'https://en.wikipedia.org/wiki/DAX'
```

Use `requests` to request the content of the website. Store the result in the variable `response`. Then check whether the query was successful.

In [2]:
```
response = requests.get(website_url)
response.status_code == requests.codes.ok
```

Out[2]:
```
True
```

If the query was successfully processed by the server, the HTML code of the web page is now stored in the `my_response.text` attribute. Print the first 125 characters of the text and have a look at it.

```
In [3]: response.text[:125]
```

```
Out[3]: '<!DOCTYPE html>\n<html class="client-nojs vector-feature-language-in-header-enabled vector-feature-language-in-main-page-heade'
```

You should receive the following text:

`'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n<meta charset="UTF-8"/>\n<title>DAX - Wikipedia</title>\n'`

You can already see the typical structure of an HTML document here. HTML stands for *Hypertext Markup Language* and is a markup language that was distributed by the European Organization for Nuclear Research (CERN) in 1992. HTML is an integral part of the *World Wide Web*, so knowing the basics will come in handy for *web scraping*.

Markup languages such as HTML are characterized by the fact that you can customize elements (e.g. paragraphs of text, images or animations) with properties, affiliations and representations. You generally do this by marking things with tags. This is the structural information we mentioned in the last lesson.

Every HTML document has a fixed structure, which looks like this:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    The header contains bits of information about the document. These are displayed as separate elements.
    <title>Title Goes Here</title>
    ...
  </head>
  <body>
    The website body goes here. This is what's displayed in the browser.
    ...
  </body>
</html>
```

In HTML you often see the combination of a word in angle brackets, then some text, followed by the same word in angle brackets with slash `/`, for example `'<title>DAX - Wikipedia</title>'`. These are the tags. These give the whole document its structure. The opening tag opens an element, for example, the title. The text that follows corresponds to the value of this element (e.g. `'DAX - Wikipedia'`). The closing tag with a slash (e.g. `'</title>'`) ends the element. You can also start new elements within an element, so they can be nested. This nesting is indicated by indentations in the example body.

HTML tags can contain attributes as well as text. These provide further information on the element. You can see an example of this in the `<html>` tag. For example, this has the attribute `lang='en'`, which specifies the language of the document as English.

HTML offers a vast number of tags and even more attributes, you generally don't have to keep them all in your head. This is why browsers such as Firefox and Chrome offer special tools to view this information on a website. This is called the *Page Inspector* in Firefox and Developer Tools in Chrome.

The following *tags* appear more frequently:

- `<h1>` : Describes headings. If there is a different number instead of 1, the tag describes a different level of heading.
- `<p>` : Describes a paragraph.
- `<a>` : Describes a hyperlink (*anchor tag*).
- `<div>` : Describes a section in the document, used to highlight text areas, for example.
- `<span>` : Like `<div>` , but for shorter sections.
- `<img>` : Describes a picture.
- `<table>` : Describes a table.
- `<tr>` : Describes a row in a table.
- `<td>` : Describes a cell in a table.

Attributes often help to identify the right elements. The `<div>` and `<span>` elements in particular are sometimes used excessively, which can make extracting information more difficult.

The data that we need is located in a table. How many tables does the website contain? Count the relevant tags.

Tip: Use the `my_string.count()` method. Remember that the tags can also contain attributes.

In [4]:
```
response.text.count('<table')
```

Out[4]:
```
10
```

We found 10 tables. The number may vary slightly for you if the website has changed slightly since we wrote this. Now we could use *string* methods to locate all the tables and print their contents to find out where the contents we need are. Fortunately, there is an easier way of doing this. We'll use a module that makes it easier for us to work with HTML.

Congratulations: You got an impression of how HTML documents are structured. This will help you understand how you can extract the information you need.

Reading HTML documents

As you've just seen, you have to look through the HTML tags if you want to isolate the company data from the website. Doing this using *string* methods alone is very tedious, especially since the tags can be nested within other tags. This is where what we call HTML parsers come in handy. These understand the structure provided by the tags and help us search for specific elements. One module we can do this with is `beautifulsoup` .

`beautifulsoup` is a very performant parser for HTML documents. You can find the parser in the `BeautifulSoup` class, which you can import as follows:

```
from bs4 import BeautifulSoup
```

You can find the most important parameter of this class here:

```
BeautifulSoup(markup=str #String containing website HTML code
             )
```

If we pass an HTML *string* to BeautifulSoup, we get an object back, which makes it easier to search for elements in the HTML code.

Import `BeautifulSoup`. Then pass the website's HTML code from `response.text` and save the result as a variable named `soup`.

In [5]:
```
from bs4 import BeautifulSoup
soup = BeautifulSoup(response.text)
```

Now that we have a `BeautifulSoup` object, we can make the website's content look a little more beautiful. The best way to do this is to use the `my_soup.prettify()` method and pass it to the `print()` function.

In [6]:
```
print(soup.prettify()[:1000])
```

```
<!DOCTYPE html>
<html class="client-nojs vector-feature-language-in-header-enabled vector-feature-lan
guage-in-main-page-header-disabled vector-feature-sticky-header-disabled vector-featu
re-page-tools-pinned-disabled vector-feature-toc-pinned-clientpref-1 vector-feature-m
ain-menu-pinned-disabled vector-feature-limited-width-clientpref-1 vector-feature-lim
ited-width-content-enabled vector-feature-custom-font-size-clientpref-0 vector-featur
e-client-preferences-disabled vector-feature-client-prefs-pinned-disabled vector-feat
ure-night-mode-disabled skin-theme-clientpref-day vector-toc-available" dir="ltr" lan
g="en">
 <head>
  <meta charset="utf-8"/>
  <title>
   DAX - Wikipedia
  </title>
  <script>
   (function(){var className="client-js vector-feature-language-in-header-enabled vec
tor-feature-language-in-main-page-header-disabled vector-feature-sticky-header-disabl
ed vector-feature-page-tools-pinned-disabled vector-feature-toc-pinned-clientpref-1 v
ector-feature-main-menu-pinned-disabled v
```

We're only interested in the `<table>` tags, because one of these tables contains the data we want to extract. To extract all the instances of a particular HTML tag, it's best to use the `my_soup.find_all()` method. You can pass it the type of tag you want as a *string* and it will return a `list` with the contents of the selected tag. Find all the tables on the Wikipedia page and store them in the variable `tables`.

In [7]:
```
tables = soup.find_all('table')
len(tables)
```

Out[7]:
```
10
```

You should have ended up with a `list` containing about 10 entries. Each table is therefore an element with the tag `table`. This element contains everything that comes between the `<table>` tag and the `</table>` tag. But how do we now find out which table we need?

This is where the *Page Inspector* or the developer tools come in handy. If we select the entire table within the Wikipedia page, it looks something like this:

Page Inspector Selection

You can see that the `<table>` tag here contains some attributes. It should look something like this `<table class="wikitable sortable jquery-tablesorter" style="text-align: center; font-size: 100%;" id="constituents" cellspacing="2" cellpadding="2">`. This means that we know the attributes of the table we want and can compare which of the tables has them.

Iterate through all the tables in `tables` to do this. Print the `my_element.attrs` attribute at each iteration. All elements in `soup` have this attribute. It's a *dictionary* containing the attributes of each tag.

In [8]:
```python
for table in tables:
    print(table.attrs)
```

```
{'class': ['infobox', 'vcard']}
{'class': ['wikitable']}
{'class': ['wikitable']}
{'class': ['wikitable', 'sortable', 'mw-collapsible', 'mw-collapsed'], 'style': 'text
-align:right;'}
{'class': ['wikitable', 'sortable'], 'style': 'text-align: center; font-size: 100%;',
'id': 'constituents', 'cellspacing': '2', 'cellpadding': '2'}
{'class': ['box-More_citations_needed', 'plainlinks', 'metadata', 'ambox', 'ambox-con
tent', 'ambox-Refimprove'], 'role': 'presentation'}
{'class': ['wikitable']}
{'class': ['nowraplinks', 'mw-collapsible', 'autocollapse', 'navbox-inner'], 'style':
'border-spacing:0;background:transparent;color:inherit'}
{'class': ['nowraplinks', 'mw-collapsible', 'autocollapse', 'navbox-inner'], 'style':
'border-spacing:0;background:transparent;color:inherit'}
{'class': ['nowraplinks', 'hlist', 'navbox-inner'], 'style': 'border-spacing:0;backgr
ound:transparent;color:inherit'}
```

You can see here that the attributes are sometimes very different. This helps us to select the right elements when web scraping, although this task becomes very difficult if a website is badly made. Then it becomes important to find the right combination of attributes. Good websites make it easy for us, for example by using the `id` attribute. With HTML, each `id` may only occur once, so that the elements can be clearly distinguished.

In our case we need the table with the `id` `constituents`.

`beautifulsoup` provides `soup` with the `my_soup.find()` method. This method allows us to select a very specific element just by using the `id`. To do this, you just have to give the `id` parameter a `str` with the ID of the element you want. Select the element with the `id` `'constituents'` and save it under the variable name `table`.

In [9]:
```python
table = soup.find(id='constituents')
```

Now we have the table stored in its own variable. Now we just need to extract the data from the table. It's useful here to know the typical structure of a table in HTML. We could get access to this with the Page Inspector or the developer tools. A table has up to three sections in HTML: Table header (`<thead>`), table body (`<tbody>`) and footer (`<tfoot>`). Each of these sections consists of rows (`<tr>`), which in turn consist of header cells (`<th>`) or data cells (`<td>`). You can imagine the whole thing like this:

```html
<table>
  <thead>
    <th>...</th>
    ...
  </thead>
  <tbody>
    <tr>
      <td>...</td>
      ...
    </tr>
    ...
  </tbody>
  <tfoot>
    <tr>
      <td>...</td>
      ...
    </tr>
  </tfoot>
</table>
```

If we compare it to a `DataFrame`, the `<th>` elements contain the column names and the `<td>` elements contain the data. So now we're interested in the contents of the individual cells. However, it's not so easy to access them without the right tools. Because each cell can contain additional tags, `beautifulsoup` helps us here as well.

First of all, the elements have the `my_element.text` attribute. It contains all the text between the opening and closing tags of the respective element and all the elements that it still contains otherwise. Try it out and print the text contents of all the elements in `table`.

In [10]:
```python
table.text
```

Out[10]:

```
'\n\nLogo\nCompany\nPrime Standard Sector\nTicker\nIndex weighting (%)1\nEmployees\nF
ounded\n\n\n\nAdidas\nApparel\nADS.DE\n2.0\n061,401 (2021)\n1924\n\n\n\nAirbus\nAeros
pace & Defence\nAIR.DE\n6.0\n126.495 (2021)\n1970\n\n\n\nAllianz\nFinancial Services
\nALV.DE\n7.1\n155,411 (2021)\n1890\n\n\n\nBASF\nChemicals\nBAS.DE\n3.5\n111,047 (202
1)\n1865\n\n\n\nBayer\nPharmaceuticals\nBAYN.DE\n4.8\n099,637 (2021)\n1863\n\n\n\nBei
ersdorf\nConsumer goods\nBEI.DE\n0.9\n020,567 (2021)\n1882\n\n\n\nBMW\nAutomotive\nBM
W.DE\n2.5\n118,909 (2021)\n1916\n\n\n\nBrenntag\nDistribution\nBNR.DE\n0.9\n017,200
(2021)\n1874\n\n\n\nCommerzbank\nFinancial Services\nCBK.DE\n0.8\n040,181 (2021)\n187
0\n\n\n\nContinental\nAutomotive\nCON.DE\n0.6\n190,875 (2021)\n1871\n\n\n\nCovestro\n
Chemicals\n1COV.DE\n0.6\n017,909 (2021)\n2015\n\n\n\nDaimler Truck\nAutomotive\nDTG.D
E\n1.1\n099,849 (2021)\n2021\n\n\n\nDeutsche Bank\nFinancial Services\nDBK.DE\n1.6\n0
82,969 (2021)\n1870\n\n\n\nDeutsche Börse\nFinancial Services\nDB1.DE\n2.7\n010,200
(2021)\n1992\n\n\n\nDeutsche Post\nLogistics\nDHL.DE\n3.4\n592,263 (2021)\n1995\n\n\n
\nDeutsche Telekom\nTelecommunication\nDTE.DE\n6.5\n216,528 (2021)\n1995\n\n\n\nE.ON
\nUtilities\nEOAN.DE\n1.9\n078,126 (2021)\n2000\n\n\n\nFresenius\nHealthcare\nFRE.DE
\n0.8\n316,078 (2021)\n1912\n\n\n\nHannover Re\nInsurance\nHNR1.DE\n0.8\n003,346 (202
1)\n1966\n\n\n\nHeidelberg Materials\nConstruction Materials\nHEI.DE\n0.7\n051,209 (2
021)\n1874\n\n\n\nHenkel\nConsumer Goods\nHEN3.DE\n0.9\n052,450 (2021)\n1876\n\n\n\nI
nfineon Technologies\nTechnology\nIFX.DE\n3.9\n050,280 (2021)\n1999\n\n\n\nMercedes-B
enz Group\nAutomotive\nMBG.DE\n4.8\n172,000 (2021)\n1926\n\n\n\nMerck\nPharmaceutical
s\nMRK.DE\n1.8\n008,081 (2021)\n1668\n\n\n\nMTU Aero Engines\nAerospace & Defence\nMT
X.DE\n1.0\n010,833 (2022)\n1934\n\n\n\nMunich Re\nFinancial Services\nMUV2.DE\n3.6\n0
40,177 (2022)\n1880\n\n\n\nPorsche\nAutomotive\nP911.DE\n1.1\n036,996 (2021)\n1931\n
\n\n\nPorsche SE\nAutomotive\nPAH3.DE\n0.6\n000,882 (2021)\n2007\n\n\n\nQiagen\nBiote
ch\nQIA.DE\n0.8\n005,900 (2021)\n1984\n\n\n\nRheinmetall\nAerospace & Defence\nRHM.DE
\n\n025.486 (2022)\n1889\n\n\n\nRWE\nUtilities\nRWE.DE\n2.2\n018,246 (2021)\n1898\n\n
\n\nSAP\nTechnology\nSAP.DE\n10.1\n107,415 (2021)\n1972\n\n\n\nSartorius\nMedical Tec
hnology\nSRT3.DE\n0.8\n018,832 (2021)\n1870\n\n\n\nSiemens\nIndustrials\nSIE.DE\n9.0
\n303,000 (2021)\n1847\n\n\n\nSiemens Energy\nEnergy technology\nENR.DE\n0.7\n092,000
(2021)\n2020\n\n\n\nSiemens Healthineers\nMedical Equipment\nSHL.DE\n1.2\n066,000 (20
21)\n2020\n\n\n\nSymrise\nChemicals\nSY1.DE\n1.1\n011,276 (2021)\n2003\n\n\n\nVolkswa
gen Group\nAutomotive\nVOW3.DE\n2.4\n672,800 (2021)\n1937\n\n\n\nVonovia\nReal Estate
\nVNA.DE\n1.1\n015,900 (2022)\n2001\n\n\n\nZalando\nE-Commerce\nZAL.DE\n0.7\n017,000
(2021)\n2008\n'
```

We get a `str`, which contains the characters `'\n'` very often, which represent line breaks. `my_element.text` separates the values of the individual tags by line breaks. `beautifulsoup` helps even further by allowing us to search through `table` just like `soup`. Each element represents a subtree of our structure and offers us the same methods again.

In our case, this means that we can output the contents of each row of a table, so we immediately get the contents of the corresponding cells. Since they are *strings*, we can use the `my_string.split()` method to separate the values.

Iterate through all the row elements (`<tr>`) in `table`. Use `my_string.split()` within the loop to split the text contents of the row elements at the line breaks and store them in lists. Store these lists in a parent list called `table_list`. Print `table_list` after the loop.

In [11]:

```
table_list = []
for row in table.find_all('tr'):
    table_list.append(row.text.split('\n'))
print(table_list)
```

```
[['', 'Logo', 'Company', 'Prime Standard Sector', 'Ticker', 'Index weighting (%)1',
'Employees', 'Founded', ''], ['', '', 'Adidas', 'Apparel', 'ADS.DE', '2.0', '061,401
(2021)', '1924', ''], ['', '', 'Airbus', 'Aerospace & Defence', 'AIR.DE', '6.0', '12
6.495 (2021)', '1970', ''], ['', '', 'Allianz', 'Financial Services', 'ALV.DE', '7.
1', '155,411 (2021)', '1890', ''], ['', '', 'BASF', 'Chemicals', 'BAS.DE', '3.5', '11
1,047 (2021)', '1865', ''], ['', '', 'Bayer', 'Pharmaceuticals', 'BAYN.DE', '4.8', '0
99,637 (2021)', '1863', ''], ['', '', 'Beiersdorf', 'Consumer goods', 'BEI.DE', '0.
9', '020,567 (2021)', '1882', ''], ['', '', 'BMW', 'Automotive', 'BMW.DE', '2.5', '11
8,909 (2021)', '1916', ''], ['', '', 'Brenntag', 'Distribution', 'BNR.DE', '0.9', '01
7,200 (2021)', '1874', ''], ['', '', 'Commerzbank', 'Financial Services', 'CBK.DE',
'0.8', '040,181 (2021)', '1870', ''], ['', '', 'Continental', 'Automotive', 'CON.DE',
'0.6', '190,875 (2021)', '1871', ''], ['', '', 'Covestro', 'Chemicals', '1COV.DE',
'0.6', '017,909 (2021)', '2015', ''], ['', '', 'Daimler Truck', 'Automotive', 'DTG.D
E', '1.1', '099,849 (2021)', '2021', ''], ['', '', 'Deutsche Bank', 'Financial Servic
es', 'DBK.DE', '1.6', '082,969 (2021)', '1870', ''], ['', '', 'Deutsche Börse', 'Fina
ncial Services', 'DB1.DE', '2.7', '010,200 (2021)', '1992', ''], ['', '', 'Deutsche P
ost', 'Logistics', 'DHL.DE', '3.4', '592,263 (2021)', '1995', ''], ['', '', 'Deutsche
Telekom', 'Telecommunication', 'DTE.DE', '6.5', '216,528 (2021)', '1995', ''], ['',
'', 'E.ON', 'Utilities', 'EOAN.DE', '1.9', '078,126 (2021)', '2000', ''], ['', '', 'F
resenius', 'Healthcare', 'FRE.DE', '0.8', '316,078 (2021)', '1912', ''], ['', '', 'Ha
nnover Re', 'Insurance', 'HNR1.DE', '0.8', '003,346 (2021)', '1966', ''], ['', '', 'H
eidelberg Materials', 'Construction Materials', 'HEI.DE', '0.7', '051,209 (2021)', '1
874', ''], ['', '', 'Henkel', 'Consumer Goods', 'HEN3.DE', '0.9', '052,450 (2021)',
'1876', ''], ['', '', 'Infineon Technologies', 'Technology', 'IFX.DE', '3.9', '050,28
0 (2021)', '1999', ''], ['', '', 'Mercedes-Benz Group', 'Automotive', 'MBG.DE', '4.
8', '172,000 (2021)', '1926', ''], ['', '', 'Merck', 'Pharmaceuticals', 'MRK.DE', '1.
8', '008,081 (2021)', '1668', ''], ['', '', 'MTU Aero Engines', 'Aerospace & Defenc
e', 'MTX.DE', '1.0', '010,833 (2022)', '1934', ''], ['', '', 'Munich Re', 'Financial
Services', 'MUV2.DE', '3.6', '040,177 (2022)', '1880', ''], ['', '', 'Porsche', 'Auto
motive', 'P911.DE', '1.1', '036,996 (2021)', '1931', ''], ['', '', 'Porsche SE', 'Aut
omotive', 'PAH3.DE', '0.6', '000,882 (2021)', '2007', ''], ['', '', 'Qiagen', 'Biotec
h', 'QIA.DE', '0.8', '005,900 (2021)', '1984', ''], ['', '', 'Rheinmetall', 'Aerospac
e & Defence', 'RHM.DE', '', '025.486 (2022)', '1889', ''], ['', '', 'RWE', 'Utilitie
s', 'RWE.DE', '2.2', '018,246 (2021)', '1898', ''], ['', '', 'SAP', 'Technology', 'SA
P.DE', '10.1', '107,415 (2021)', '1972', ''], ['', '', 'Sartorius', 'Medical Technolo
gy', 'SRT3.DE', '0.8', '018,832 (2021)', '1870', ''], ['', '', 'Siemens', 'Industrial
s', 'SIE.DE', '9.0', '303,000 (2021)', '1847', ''], ['', '', 'Siemens Energy', 'Energ
y technology', 'ENR.DE', '0.7', '092,000 (2021)', '2020', ''], ['', '', 'Siemens Heal
thineers', 'Medical Equipment', 'SHL.DE', '1.2', '066,000 (2021)', '2020', ''], ['',
'', 'Symrise', 'Chemicals', 'SY1.DE', '1.1', '011,276 (2021)', '2003', ''], ['', '',
'Volkswagen Group', 'Automotive', 'VOW3.DE', '2.4', '672,800 (2021)', '1937', ''],
['', '', 'Vonovia', 'Real Estate', 'VNA.DE', '1.1', '015,900 (2022)', '2001', ''],
['', '', 'Zalando', 'E-Commerce', 'ZAL.DE', '0.7', '017,000 (2021)', '2008', '']]
```

Now check the number of entries in each sub-list in `table_list`.

In [12]:

```python
for row in table_list:
    print(len(row))
```

```
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
9
```

In our case, the sub-lists of `table_list` each have 9 entries. So they're all the same length. This makes it easy for us to transfer them into a table which we can then save. Our table will have 9 columns accordingly. Create a `DataFrame` from `table_list` and name it `df_dax`. You should use the first list in `table_list` for the column names. Then print the number of rows and columns as well as the first five rows of `df_dax`.

In [13]:

```
import pandas as pd
df_dax = pd.DataFrame(table_list[1:], columns=table_list[0])
print(df_dax.shape)
df_dax.head()
```

```
(40, 9)
```

Out[13]:

	Logo	Company	Prime Standard Sector	Ticker	Index weighting (%)1	Employees	Founded
0		Adidas	Apparel	ADS.DE	2.0	061,401 (2021)	1924
1		Airbus	Aerospace & Defence	AIR.DE	6.0	126.495 (2021)	1970
2		Allianz	Financial Services	ALV.DE	7.1	155,411 (2021)	1890
3		BASF	Chemicals	BAS.DE	3.5	111,047 (2021)	1865
4		Bayer	Pharmaceuticals	BAYN.DE	4.8	099,637 (2021)	1863

`df_dax` should look something like this:

First rows of df_dax

Your `DataFrame might look slightly different as the website contents may have changed.

When looking at `df_dax` you might notice that only 6 of the 9 columns are displayed. So there are 3 completely empty columns. This is mainly because links and images without text content have led to line breaks without any content. For example, we don't have the logos of the companies which consist of a link and an image. Delete the columns that have only an empty `str`` (` ''`) as the column name. If your table does *not* contain any empty columns you can ignore the next code cell.

In [14]:

```
df_dax = df_dax.drop('', axis=1)
df_dax.head()
```

Out[14]:

	Logo	Company	Prime Standard Sector	Ticker	Index weighting (%)1	Employees	Founded
0		Adidas	Apparel	ADS.DE	2.0	061,401 (2021)	1924
1		Airbus	Aerospace & Defence	AIR.DE	6.0	126.495 (2021)	1970
2		Allianz	Financial Services	ALV.DE	7.1	155,411 (2021)	1890
3		BASF	Chemicals	BAS.DE	3.5	111,047 (2021)	1865
4		Bayer	Pharmaceuticals	BAYN.DE	4.8	099,637 (2021)	1863

The data is now available in a structured form. One observation corresponds to one company. Every company has its own row Each variable that describes the company in more detail has its own column. But each value doesn't yet have its own cell. The `'Employees'` column also shows the year in which the data was collected. So the data doesn't yet follow the *tidy data principles*. So we could structure the data even better by adding more columns and separating the values that are stored together. But for now let's move on. You'll see how you can do this elegantly later in the chapter with another example.

It's also a good idea to adjust the data types in `df_dax`. This ensures that the data is structured in such a way that we can use it easily in the future. What types of data do we have at the moment?

In [17]:
```
df_dax.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 40 entries, 0 to 39
Data columns (total 7 columns):
 #   Column                 Non-Null Count  Dtype 
---  ------                 --------------  ----- 
 0   Logo                   40 non-null     object
 1   Company                40 non-null     object
 2   Prime Standard Sector  40 non-null     object
 3   Ticker                 40 non-null     object
 4   Index weighting (%)1   40 non-null     object
 5   Employees              40 non-null     object
 6   Founded                40 non-null     object
dtypes: object(7)
memory usage: 2.3+ KB
```

All our columns have the `object` type. This was to be expected, since we used lists of strings.

The column with the number of employees still contains the year. Because we don't need this, we'll remove them quickly. For this we'll use *regular expressions*, which you will get to know later in this chapter.

In [18]:
```
df_dax.loc[:, 'Employees'] = df_dax.loc[:, 'Employees'].str.replace(r'\(\d\d\d\d\)', '
```

Convert the columns that do not contain text to numeric data types. Then check the data types.

Tip: Use the `pd.to_numeric()` function. If individual values are not numeric (e.g. because they represent missing values) you can use the `errors='coerce'` parameter. If columns use commas as the thousands separator, you should remove them using a *string* method.

In [19]:
```
df_dax.loc[:, 'Index weighting (%)1'] = pd.to_numeric(df_dax.loc[:, 'Index weighting (
df_dax.loc[:, 'Employees'] = pd.to_numeric(df_dax.loc[:, 'Employees'].str.replace(',',
df_dax.loc[:, 'Founded'] = pd.to_numeric(df_dax.loc[:, 'Founded'])

df_dax.dtypes
```

Out[19]:
```
Logo                     object
Company                  object
Prime Standard Sector    object
Ticker                   object
Index weighting (%)1    float64
Employees               float64
Founded                   int64
dtype: object
```

Save `df_dax` as a *pickle* `dax_data.p` so you don't have to do all this work again.

In [20]:
```
df_dax.to_pickle('dax_data.p')
```

Now that you've gone through the tedious but universal process of reading and processing HTML tags, here's a simple trick to quickly read table data from web pages. Pandas has a method that performs the steps we just went through line by line in the code.

You can simply use the `pd.read_html()` function and pass a web address and optional table attributes to automatically transfer table data into a `DataFrame`. You will get a `list` containing all the tables on the website that contain the attributes you specify.

Here are the most important parameters:

```
pd.read_html(io=str,      #url or HTML text
             attrs=dict) #(optional) Attributes to select specific table
like {attr: value}
```

Now try to create an (uncleaned) `DataFrame` with the information about the DAX companies using `pd.read_html()`.

In [21]:
```
dfs = pd.read_html(website_url, attrs={'id': 'constituents'})
dfs[0].head()
```

Out[21]:

	Logo	Company	Prime Standard Sector	Ticker	Index weighting (%)1	Employees	Founded
0	NaN	Adidas	Apparel	ADS.DE	2.0	061,401 (2021)	1924
1	NaN	Airbus	Aerospace & Defence	AIR.DE	6.0	126.495 (2021)	1970
2	NaN	Allianz	Financial Services	ALV.DE	7.1	155,411 (2021)	1890
3	NaN	BASF	Chemicals	BAS.DE	3.5	111,047 (2021)	1865
4	NaN	Bayer	Pharmaceuticals	BAYN.DE	4.8	099,637 (2021)	1863

Congratulations: You have successfully read the HTML code with `beautifulsoup` and transferred some data from a webpage into a `DataFrame`. The Taiwanese investor is pleased that you are making good progress in gathering data for him! Until now, you could have also just collected the data manually. It was only a table with 30 entries. However, the approach we've developed will also work well for larger web scraping projects that would take a lot of effort to carry out by hand. Next, we'll access many web pages one after the other and prepare the data in them.

Remember:

- Transform HTML code with `beautifulsoup`

- Iterate through elements with a certain tag with `my_soup.find_all('tagname')`
- Find specific elements with `id` attribute with `my_soup.find(id='my_id')`
- Output all text content within an element and its sub-elements with `my_element.text`
- Read web pages with the help of `pd.read_html()`

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
