Communicating with Websites

Module 2 | Chapter 1 | Notebook 1

Data scientists prefer to work with well-structured data that is clean and already available. But it doesn't always work like that. Sometimes this data isn't available anywhere in your company. In these situations, the internet can help you. There's a lot of interesting data available there. However, this is rarely available as a structured data set. Instead, it's scattered over various websites. Extracting data automatically from web pages is called *web scraping*. In this chapter, we'll look at how you can extract data from web pages. In this lesson you will learn about:

- The `requests` module
- Website status codes

Reading a website

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.

So this time you'll have to find the data ourselves. When you're looking for data, the internet is the first place most people look. You can find quite a lot there. But the data is rarely nicely organized. It's often hidden somewhere in a web page's HTML code. There are various modules for Python to access this kind of data. A very basic module is `requests`. `requests` makes it easy for us to access websites with Python. Usually you don't use an alias for this module. Import `requests`.

In [2]:
```
import requests
```

First of all, we want to find out which companies are currently listed on the DAX. We can find this out by looking at the relevant Wikipedia page: *https://en.wikipedia.org/wiki/DAX*, for example. If you want to pull data from a web page, it's often a good idea to have a look at how the page is structured first. Click on the link and take a look at the page.

The current companies are listed in a table under the **Components** heading. We'll access these later. First we'll connect to the website. Data on the internet is usually transmitted using HTTPS (Hypertext Transfer Protocol Secure). This is an encrypted version of HTTP (Hyptertext Transfer Protocol). The browser sends a request to the server and the server sends a response back.

With `requests`, we can even send the request to the web server without using a browser. The module offers the `requests.get()` function for this purpose. Store the address of the Wikipedia page as a *string* in the `website_url` variable. Pass this variable to `requests.get()` and store the response as `response`. Then print it.

In [7]:
```
website_url = 'https://en.wikipedia.org/wiki/DAX'
response = requests.get(website_url)
response
```

Out[7]:
```
<Response [200]>
```

You should have ended up with something like `<Response [200]>`. So what exactly does that mean? We've received a reply to our request to send us all the information we need to reproduce the Wikipedia page. This has the status code 200. There are some standard codes for status codes. They can be divided into 5 categories, indicated by the first digit:

number	meaning
1XX	The request was received (Information)
2XX	The request was received and accepted (successful request)
3XX	Further action has to be taken to fulfill the request (redirection)
4XX	The request cannot be carried out, which is probably due to the client (client error)
4XX	The request cannot be carried out, which is probably due to the server(server error)

The next two digits then specify the answer. You can find an extensive list of response codes here.

We received the code 200. This code indicates that everything went smoothly. Here are some other important codes:

- 401 (access to the information you requested has been denied, possibly due to missing/incorrect authentication)
- 404 (the information you requested could not be found)
- 429 (You made too many requests in a specific time frame)

Now let's take a closer look at the response. Which data type does it have?

In [8]:
```
type(response)
```

Out[8]:
```
requests.models.Response
```

`response` has the `Response` type. A `response` contains some information that is contained in different attributes. The status code is stored as an `int` in the `my_response.status_code` attribute. When we make automated requests, it makes sense to check the status before we continue working with the response. `requests.codes` offers us all codes that are readable for humans. This allows people with no or little experience with websites to read the code.

We'll only need the code `requests.codes.ok`. It tells us that everything is okay (code 200). Compare this with `response.status_code`. Are they the same?

In [15]:
```
response.status_code == requests.codes.ok
```

Out[15]:
```
True
```

The status code should be the same as `requests.codes.ok`. If you want to be doubly sure, you can issue an error message if the status code does not match. The method `my_response.raise_for_status()` will do this for you. Let's try it out. Use `requests` to get the website data from https://httpbin.org/status/404. Save the response as `response_404` and use the `my_response.raise_for_status()` method. What happens?

In [22]:
```
#response_404 = response.raise_for_status('https://httpbin.org/status/404')
response_404 = requests.get('https://httpbin.org/status/404')
response_404.raise_for_status()
```

```
---------------------------------------------------------------------------
HTTPError                                 Traceback (most recent call last)
Input In [22], in <module>
      1 #response_404 = response.raise_for_status('https://httpbin.org/status/404')
      2 response_404 = requests.get('https://httpbin.org/status/404')
----> 3 response_404.raise_for_status()

File ~/.virtualenvs/training_env/lib/python3.8/site-packages/requests/models.py:960,
in Response.raise_for_status(self)
    957     http_error_msg = u'%s Server Error: %s for url: %s' % (self.status_code,
reason, self.url)
    959 if http_error_msg:
--> 960     raise HTTPError(http_error_msg, response=self)

HTTPError: 404 Client Error: NOT FOUND for url: https://httpbin.org/status/404
```

We get the error message `HTTPError: 404 Client Error: NOT FOUND for url: https://httpbin.org/status/404`. You might have seen status code 404 in your browser, whenever a website cannot be found. What happens when you try to generate a status message for the Wikipedia page, i.e. with `response`?

In [28]:
```
response.raise_for_status()
```

Nothing happened. This is a good sign! If `my_response.raise_for_status()` doesn't generate an error message, then the request has been processed by the server and you have received a response that you can continue working with.

In addition to the status code, you can find more information in our response. Print the `my_response.headers` attribute.

In [29]:
```
response.headers
```

Out[29]:

```
{'date': 'Fri, 12 Apr 2024 06:55:25 GMT', 'vary': 'Accept-Encoding,Cookie,Authorizati
on', 'server': 'ATS/9.1.4', 'x-content-type-options': 'nosniff', 'content-language':
'en', 'origin-trial': 'AonOP4SwCrqpb0nhZbg554z9iJimP3DxUDB8V4yu9fyyepauGKD0NXqTknWi4g
nuDfMG6hNb7TDUDTsl0mDw9gIAAABmeyJvcmlnaW4iOiJodHRwczovL3dpa2lwZWRpYS5vcmc6NDQzIiwiZmV
hdHVyZSI6IlRvcExldmVsVHBjZCIsImV4cGlyeSI6MTczNTM0Mzk5OSwiaXNTdWJkb21haW4iOnRydWV9',
'accept-ch': '', 'last-modified': 'Thu, 11 Apr 2024 07:19:52 GMT', 'content-type': 't
ext/html; charset=UTF-8', 'content-encoding': 'gzip', 'age': '8845', 'x-cache': 'cp30
71 hit, cp3071 hit/18', 'x-cache-status': 'hit-front', 'server-timing': 'cache;desc
="hit-front", host;desc="cp3071"', 'strict-transport-security': 'max-age=106384710; i
ncludeSubDomains; preload', 'report-to': '{ "group": "wm_nel", "max_age": 604800, "en
dpoints": [{ "url": "https://intake-logging.wikimedia.org/v1/events?stream=w3c.report
ingapi.network_error&schema_uri=/w3c/reportingapi/network_error/1.0.0" }] }', 'nel':
'{ "report_to": "wm_nel", "max_age": 604800, "failure_fraction": 0.05, "success_fract
ion": 0.0}', 'set-cookie': 'WMF-Last-Access=12-Apr-2024;Path=/;HttpOnly;secure;Expire
s=Tue, 14 May 2024 00:00:00 GMT, WMF-Last-Access-Global=12-Apr-2024;Path=/;Domain=.wi
kipedia.org;HttpOnly;secure;Expires=Tue, 14 May 2024 00:00:00 GMT, WMF-DP=fc4;Path=/;
HttpOnly;secure;Expires=Fri, 12 Apr 2024 00:00:00 GMT, GeoIP=DE:BE:Berlin:52.41:13.3
7:v4; Path=/; secure; Domain=.wikipedia.org, NetworkProbeLimit=0.001;Path=/;Secure;Ma
x-Age=3600', 'x-client-ip': '167.235.119.111', 'cache-control': 'private, s-maxage=0,
max-age=0, must-revalidate', 'accept-ranges': 'bytes', 'content-length': '35425'}
```


The output is a bit confusing at first sight. But we can tell from the curly brackets that it's probably a *dictionary*. That's true but there's more to it than that. Since the header data for communication via HTTP has been defined as *case insensitive*, this also applies to `my_response.headers`. So it doesn't matter if the letters in the *keys* are upper or lower case. `my_response.headers` contains metadata about the response we received. What values are paired with the *keys* `'content-type'` and `'date'`? Output them and see.

Tip: As we mentioned just now, the *keys* here are not case-sensitive, unlike a normal `dict`. So it does not matter whether you write `'content-type'` or `'CONTENT-type'`.

In [31]:
```
print(response.headers['content-type'])
print(response.headers['date'])
```

```
text/html; charset=UTF-8
Fri, 12 Apr 2024 06:55:25 GMT
```

`'Date'` returns the date and time of the server response, as you may have guessed. `'content-type'` indicates what kind of content we received. In our case this is a text encoded with `'UTF-8'` and containing HTML code. Take a look at the text by outputting the first 500 characters of the `my_response.text` attribute.

In [32]:
```
response.text[:500]
```

Out[32]:

```
'<!DOCTYPE html>\n<html class="client-nojs vector-feature-language-in-header-enabled
vector-feature-language-in-main-page-header-disabled vector-feature-sticky-header-dis
abled vector-feature-page-tools-pinned-disabled vector-feature-toc-pinned-clientpref-
1 vector-feature-main-menu-pinned-disabled vector-feature-limited-width-clientpref-1
vector-feature-limited-width-content-enabled vector-feature-custom-font-size-clientpr
ef-0 vector-feature-client-preferences-disabled vector-feature-client-prefs-p'
```


It contains all of the website's content as HTML code. At first glance, this looks pretty messy. However, data in HTML form is not completely unstructured. We call this semi-structured data. Semi-structured data carries part of the structure information with it, instead of conforming to a general structure already like a table. You may have come across other semi-structured data

formats. These include JSON and XML. In the next lesson you'll get a better understanding of what we mean when we say that HTML carries part of the structural information with it.

In addition to semi-structured data, there's also unstructured data and structured data. Most of the data companies have access to is unstructured. This is mainly texts in natural language. But audio and video recordings as well as images also come under unstructured data. These are characterized by the fact that it is particularly difficult to extract information from them and make it usable for analyses and models.

It's best to work with structured data where possible. This is data that's available in tabular form, such as SQL databases, for example. You can put this directly into a machine learning model without much effort. However, since most data is not available in this form, data scientists often have to structure the data first. This takes up a lot of your work time. When structuring data, you should pay attention to 3 principles to ensure that the data can be used as easily as possible for automated evaluations and models. These principles are often called *tidy data principles*. They are as follows:

- Each observation has its own row
- Each variable has its own column
- Each value has its own cell

In this chapter, we'll extract publicly available data from websites and then put it into a structured form according to these principles. In other words, we'll structure it the way we'd like to receive data ourselves.

Congratulations: You've got to know the `requests` module. You used this to query a website and look at it in text form. You also learned what the status codes stand for and how to create an error message to safeguard against a bad status code.

Remember:

- Access a web page with `requests.get('my_website_url')`
- Check the status of the server response with `my_response.status_code == requests.codes.ok` or generate an error with `my_response.raise_for_status()` if the response is unusable.
- Output HTML content of the web page with `my_response.text`

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.
