# Web Scraping

Prof. Dr. Seppe vanden Broucke

seppe.vandenbroucke@kuleuven.be

# Outline

- **Introduction**
- **Web Scraping with Python**
  - Why Python?
  - The three core pillars: HTTP, HTML, and CSS
  - The three core libraries: Requests, BeautifulSoup, Selenium
  - Hands-on examples
  - Other noteworthy libraries
- **Other Tools and Cross-overs**
  - Commercial products
  - What without Python?
  - Web scraping versus web crawling
  - Web scraping versus AI and ML
  - Web scraping versus RPA
  - From web scraping to <blank> scraping
- **Managerial Aspects**
  - Legal concerns
  - Web scraping as part of your data science pipeline
  - Buy or build?
  - When not to opt for web scraping?
  - How to manage a web scraping project?
- **Conclusions**

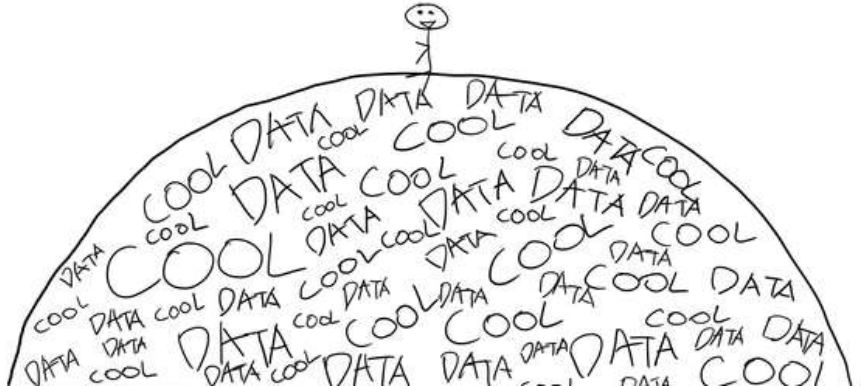
# Introduction

# Overview

- What is web scraping?
- Which use cases does web scraping enable?
- Some practical examples
- Workshop outline

# What is web scraping?

- Web scraping = web harvesting = web data extraction = web data mining
  - The construction of a software agent to download, parse, and organize data from the web in an automated manner
  - Instead of a human end-user clicking away in a web browser and extracting interesting parts into, web scraping offloads this task to a computer program which can execute it much faster, and more correctly, than a human can
  - From unstructured data on the web to structured data

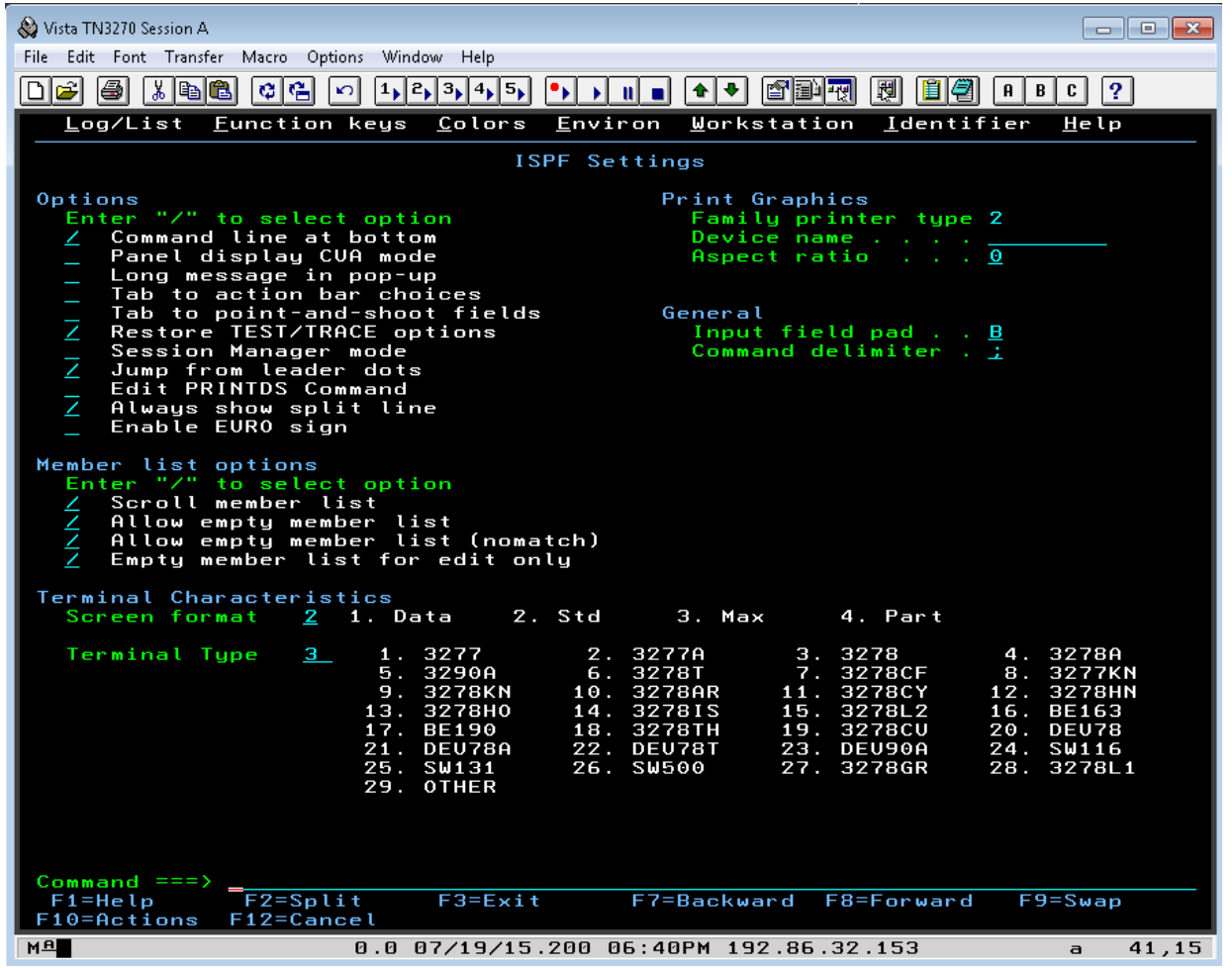


# What is web scraping?

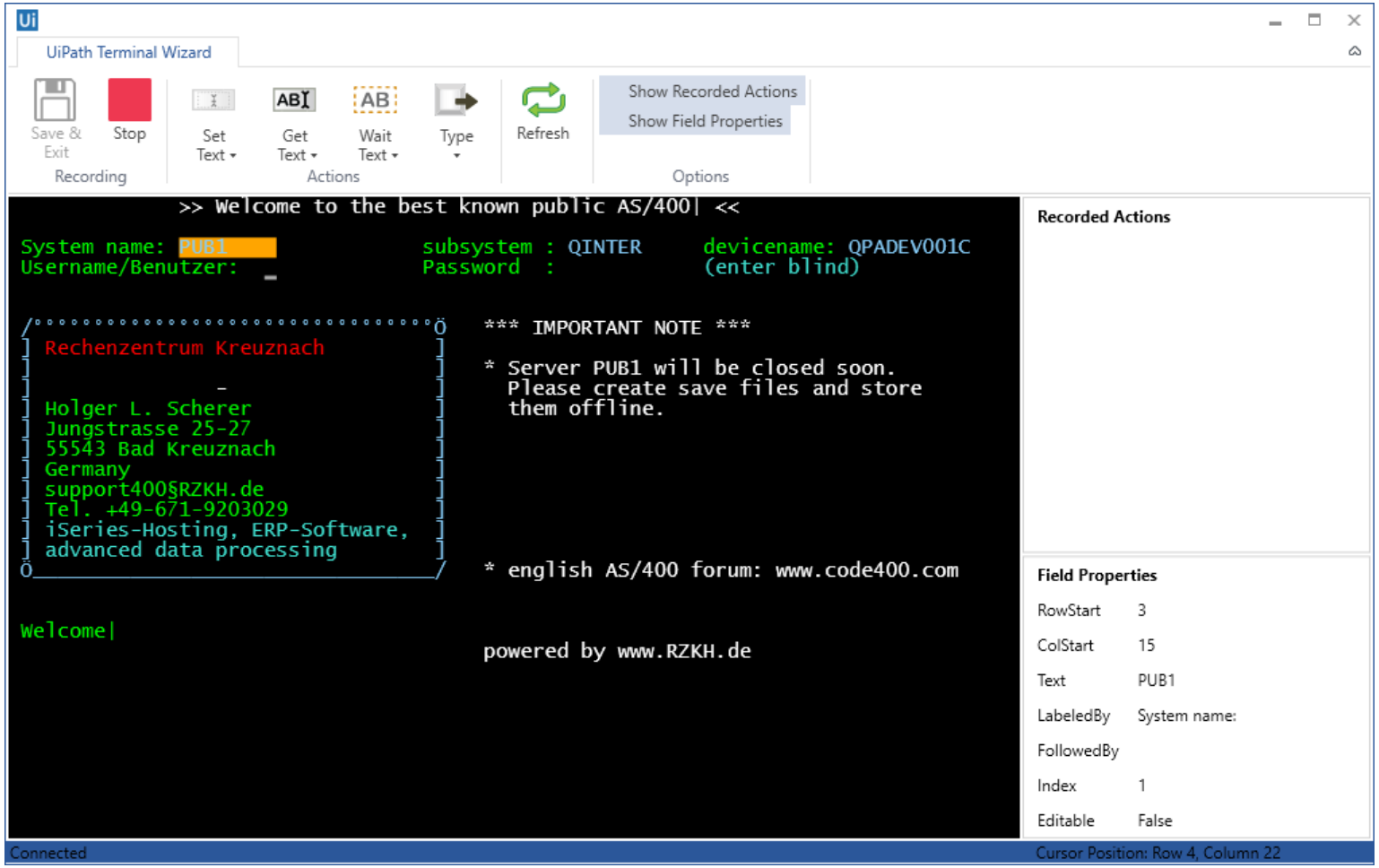
- The automated gathering of data from the Internet is as old as the Internet itself

- The term “scraping” has been around for much longer than the web
  - Before “web scraping” became popularized, a practice known as “screen scraping” was already well-established as a way to extract data from a visual representation, which in the early days of computing boiled down to simple, text based “terminals”

# What is web scraping?

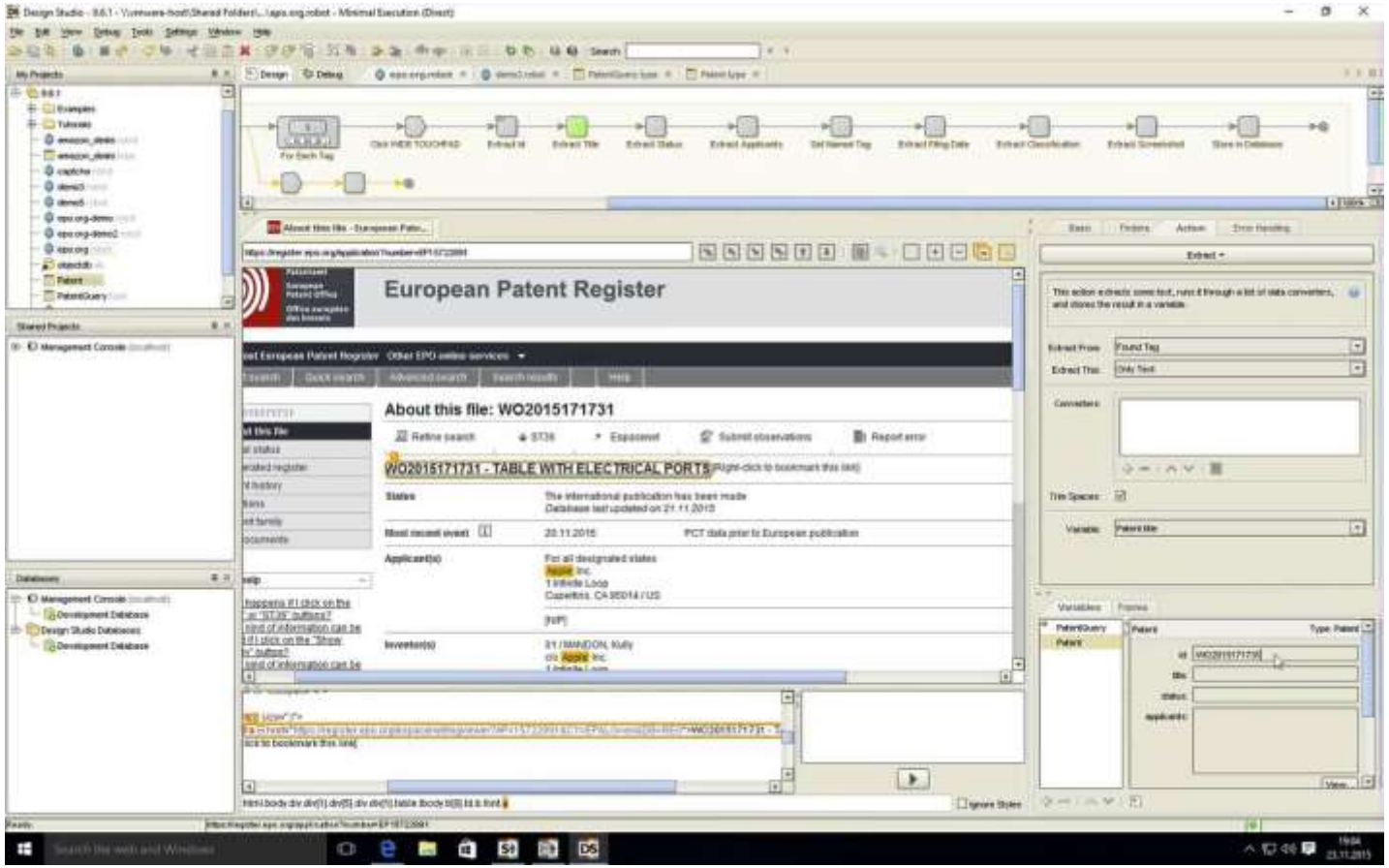


# What is web scraping?



# What is web scraping?

# What is web scraping?



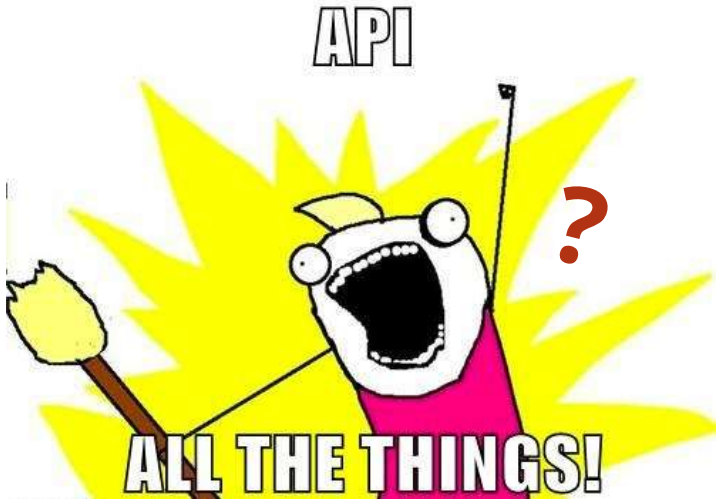
# Why?

- The web contains a treasure trove of data
  - Rich input source for big data and analytics
  - But: very unstructured in nature: every page has different layout, semantics, navigation…
  - Does not always make it easy to gather or export this data in an easy manner
  - Web browsers are very good at showing images, displaying animations, and laying out websites in a way that is visually appealing to humans, but they do not expose a simple way to export their data (in most cases)

- **Web scraping can aid to extract and enrich data sources, automate processes, capture open data…**

# Why not?

- “Isn’t this what an API is for?”
  - E.g. as offered by Twitter, Google, Facebook, YourDataVendor™…
  - Generally suggested to explore this option before you embark on a web scraping quest!
  - But…
    - The website might not offer an API
    - The API does not expose all information or not at the same level of detail
    - The API is expensive (cost or setup time)
    - The API is rate-limited

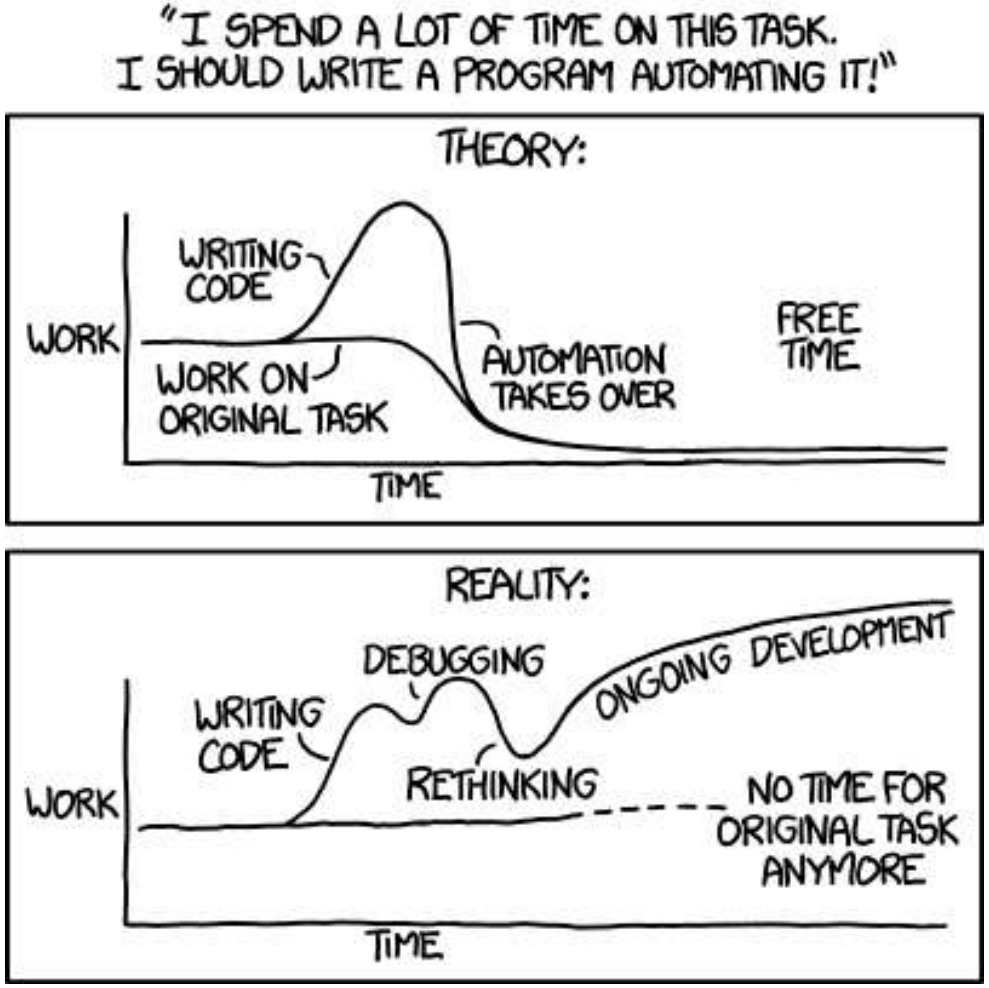


# Why not?

- For one-off projects dealing with relatively structured data


| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | Nr. | Regering | Premier | Partijen | Van | Tot | Dagen |
| 2 | - | Voorlopig Bewind | Louis de Potter | Unionistisch | 24 september 1830 | 25 februari 1831 | 154 |
| 3 | 1 | De Gerlache | Etienne de Gerlache | Unionistisch | 26 februari 1831 | 23 maart 1831 | 25 |
| 4 | 2 | Lebeau I | Joseph Lebeau | Liberalen | 23 maart 1831 | 21 juli 1831 | 120 |
| 5 | 3 | De Muelenaere | Felix de Muelenaere | Unionistisch | 26 juli 1831 | 17 september 1832 | 419 |
| 6 | 4 | Goblet d'Alviella | Albert Goblet d'Alviella | Unionistisch | 20 oktober 1832 | 1 augustus 1834 | 650 |
| 7 | 5 | De Theux de Meylandt I | Barthélémy de Theux de Meylandt | Unionistisch | 4 augustus 1834 | 6 april 1840 | 2072 |

# Why not?




https://xkcd.com/1319/

# Examples

- Lyst
  - http://talks.lystit.com/dsl-scraping-presentation/
  - “A London based online fashion marketplace, scraped the web for semi-structured information about fashion products and then applied machine learning to present this information cleanly and elegantly for consumers from one central website”

# Examples

- Finance
  - https://www.ft.com/content/08a22da8-b587-11e6-ba85-95d1533d9a62
  - “Scouring emails for such nuggets of information is just the tip of the “alternative data” iceberg. Almost everything we do leaves a digital fingerprint, which can be scraped, aggregated and sold to investment firms looking for tradable signals, or to use the jargon — market-beating “alpha””

# Examples

- “We Feel Fine”, Jonathan Harris and Sep Kamvar
  - Web scraper and crawler to search sentences starting with “I feel…”

# Examples

- “Using big data to predict suicide risk among Canadian youth”, SAS
  - https://www.sas.com/en_us/insights/articles/analytics/using-big-data-to-predict-suicide-risk-canada.html
  - Web scraper for blogs, Twitter, other social media
  - “Team members [...] collected 2.3 million tweets and used text mining software to identify 1.1 million of them as likely to have been authored by 13 to 17 year olds in Canada by building a machine learning model to predict age, based on the open source PAN author profiling dataset. Their analysis made use of natural language processing, predictive modelling, text mining, and data visualization”

# Examples

- “The Billion Prices Project: Using Online Prices for Measurement and Research”
  - http://www.nber.org/papers/w22111
  - “Web scraping was used to collect a data set of online price information which was used to construct a robust daily price index for multiple countries”

# Examples

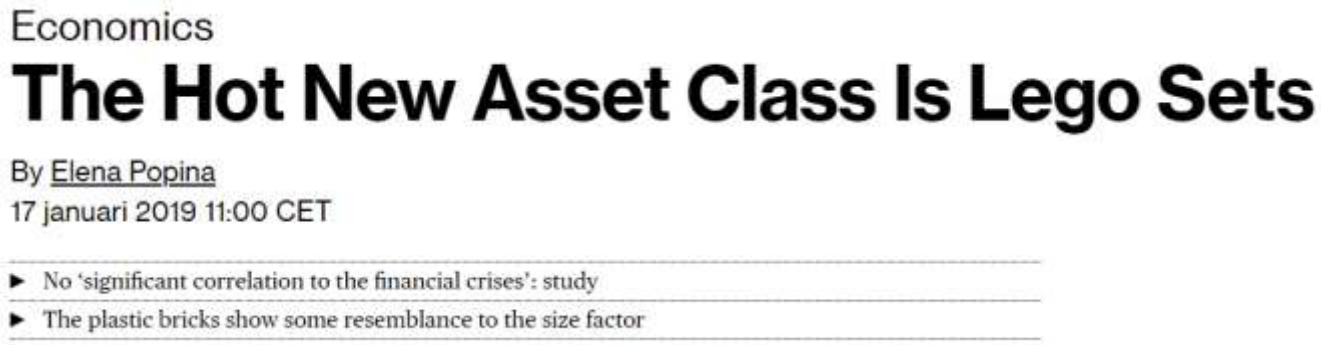
- “Predicting selfie success”
  - http://karpathy.github.io/2015/10/25/selfie/
  - “Train a deep learning model based on scraped images from Tinder and Instagram together with their “likes” to predict whether an image would be deemed “attractive””

# Examples

- Fintech
  - https://nordicapis.com/fintechs-want-save-screen-scraping/
  - “This is why screen scraping is so important to the FinTech community. The protagonists have taken advantage of what is available to them, and want to continue to do so. If FinTechs had waited for the banking industry in general to create APIs, there would be no FinTech scene. Organizations like Yodlee, Sofort, Trustly and Figo would either not exist or have much less compelling offers. Consumers would be importing transactions from online banking into Xero or Quickbooks by doing CSV exports and then massaging the data into their chosen accounting package”

# Examples

- “LEGO - The Toy of Smart Investors”, Victoria Dobrynskaya and Julia Kishilova
  - https://www.bloomberg.com/news/articles/2019-01-17/lego-collecting-delivers-huge-and-uncorrelated-market-returns
  - “We collect price data for LEGO sets from the website Brickpicker.com and the book "The Ultimate Guide to Collectible LEGO Sets" (subsequently referred to as “price guide”) written by the founders of Brickpicker.com Ed and Jeff Maciorowski”



Economics

**The Hot New Asset Class Is Lego Sets**

By Elena Popina

17 januari 2019 11:00 CET

- No ‘significant correlation to the financial crises’: study
- The plastic bricks show some resemblance to the size factor

# Examples

- HR analytics
  - https://www.bloomberg.com/news/features/2017-11-15/the-brutal-fight-to-mine-your-data-and-sell-it-to-your-boss
  - “The San Francisco based hiQ startup specializes in selling employee analyses by collecting and examining public profile information, for instance from LinkedIn (who was not happy about this but was so far unable to prevent this practice following a court case)”

# Examples

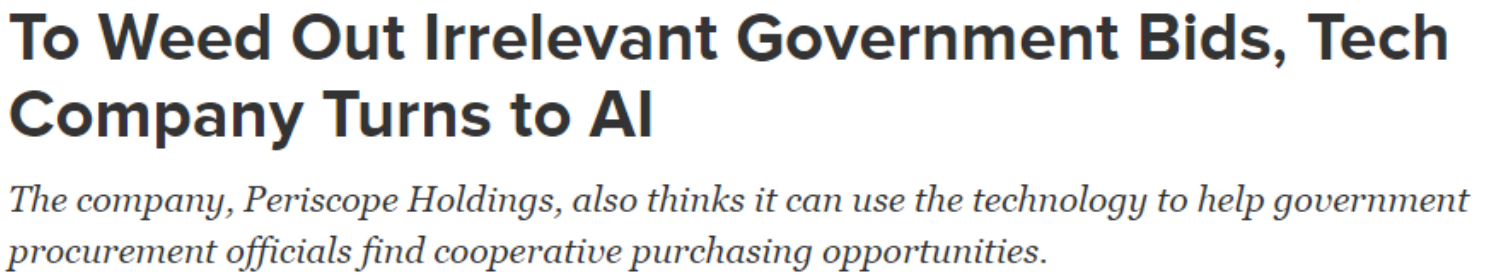
- Real estate
  - https://blog.datahut.co/web-scraping-in-real-estate-the-ultimate-tool/
  - “Scraping the web provides parameters which the realtor can further study to determine sales and prospective buyers”

# Examples

- Fraud analytics
  - https://securityboulevard.com/2018/10/scraping-social-security-numbers-on-the-web/
  - “The most important steps to take once it is believed that a Social Security Number has been compromised is to first file an identity theft report with the local police. Secondly, place a fraud alert on your credit file indicating that a potential identity theft has occurred”

# Examples

- AI data sources
  - http://www.govtech.com/biz/To-Weed-Out-Irrelevant-Government-Bids-Tech-Company-Turns-to-AI.html
  - “And BidSync casts a wide net for government bids. Using Web-scraping tools, the company picks up procurements from virtually all areas of government work”

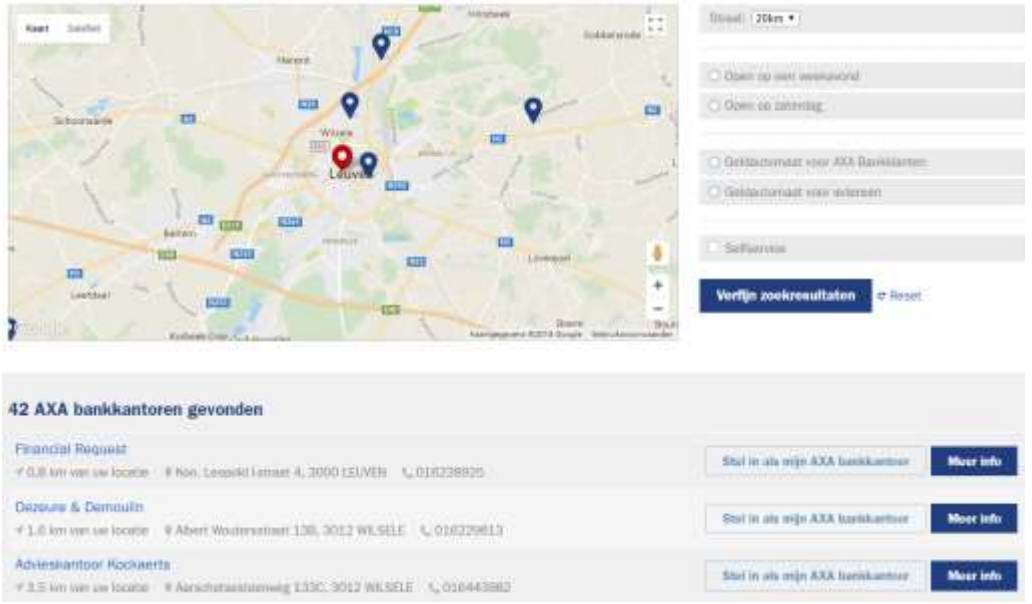


**To Weed Out Irrelevant Government Bids, Tech Company Turns to AI**

*The company, Periscope Holdings, also thinks it can use the technology to help government procurement officials find cooperative purchasing opportunities.*

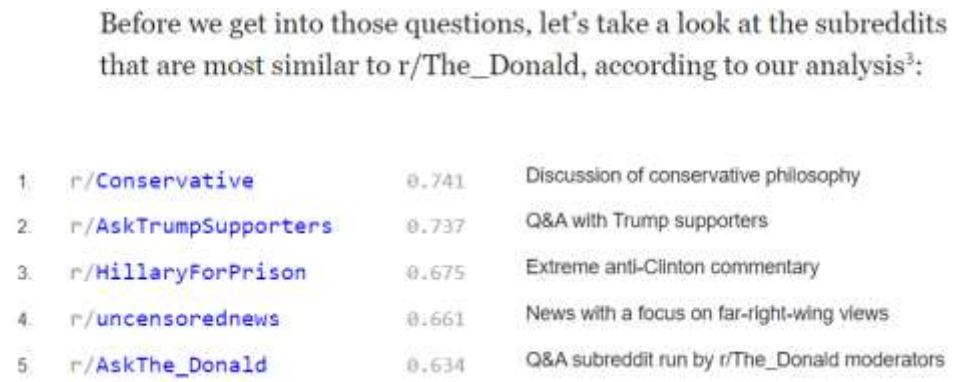
# Examples

- Competitor analytics
  - “Banks and other financial institutions are using web scraping for competitor analysis. For example, banks frequently scrape competitor's sites to get an idea of where branches are being opened or closed, or to track loan rates offered—all of which is interesting information which can be incorporated in their internal models and forecasting. Investment firms also often use web scraping, for instance to keep track of news articles regarding assets in their portfolio”



# Examples

- Data-driven journalism
  - https://fivethirtyeight.com/features/dissecting-trumps-most-rabid-online-following/
  - “Sociopolitical scientists are scraping social websites to track population sentiment and political orientation. A famous article called “Dissecting Trump’s Most Rabid Online Following” analyzes user discussions on reddit using semantic analysis to characterize the online followers and fans of Donald Trump”



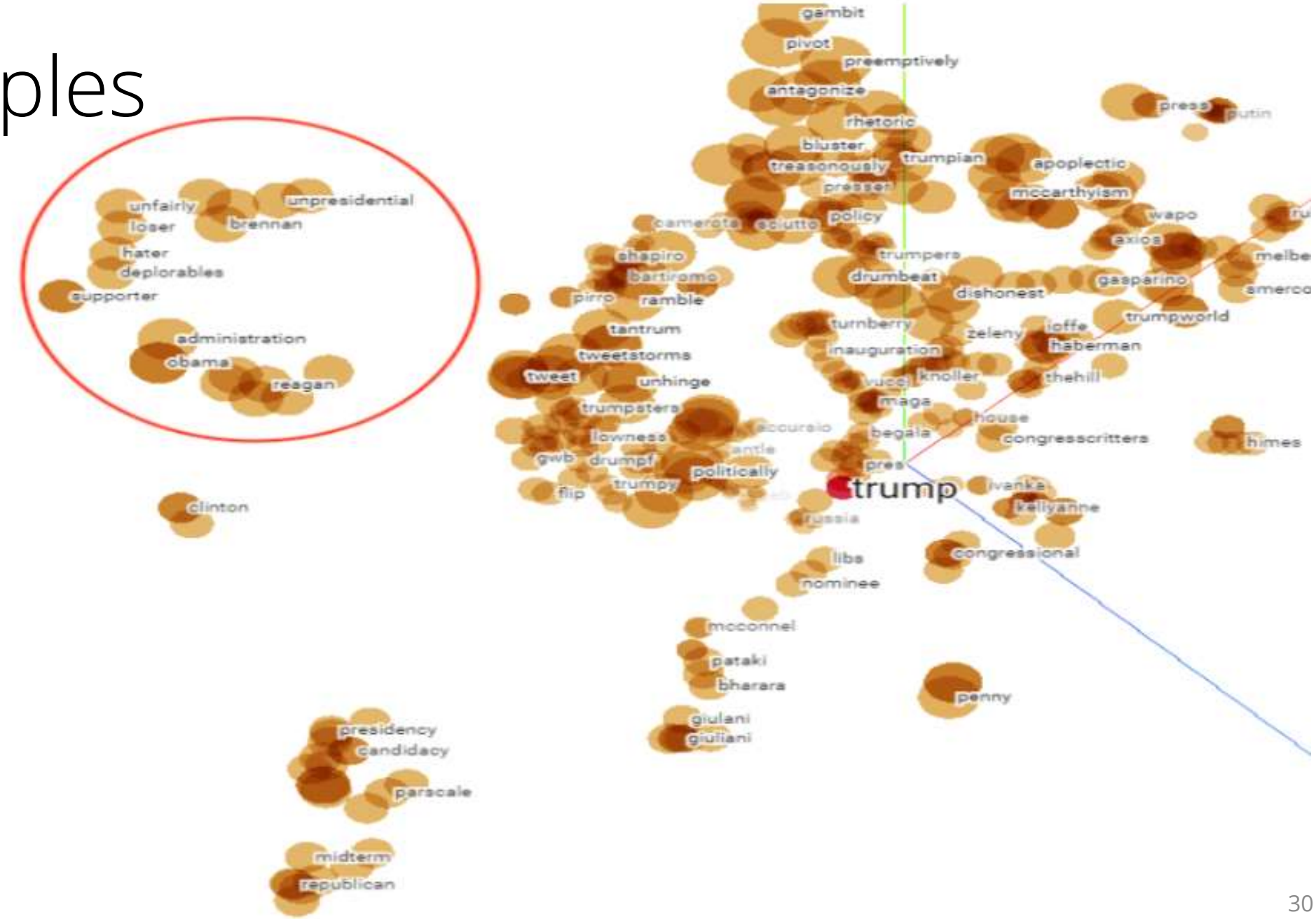
Before we get into those questions, let's take a look at the subreddits that are most similar to r/The_Donald, according to our analysis[3]:

| | | | |
|---|---|---|---|
| 1. | r/Conservative | 0.741 | Discussion of conservative philosophy |
| 2. | r/AskTrumpSupporters | 0.737 | Q&A with Trump supporters |
| 3. | r/HillaryForPrison | 0.675 | Extreme anti-Clinton commentary |
| 4. | r/uncensorednews | 0.661 | News with a focus on far-right-wing views |
| 5. | r/AskThe_Donald | 0.634 | Q&A subreddit run by r/The_Donald moderators |

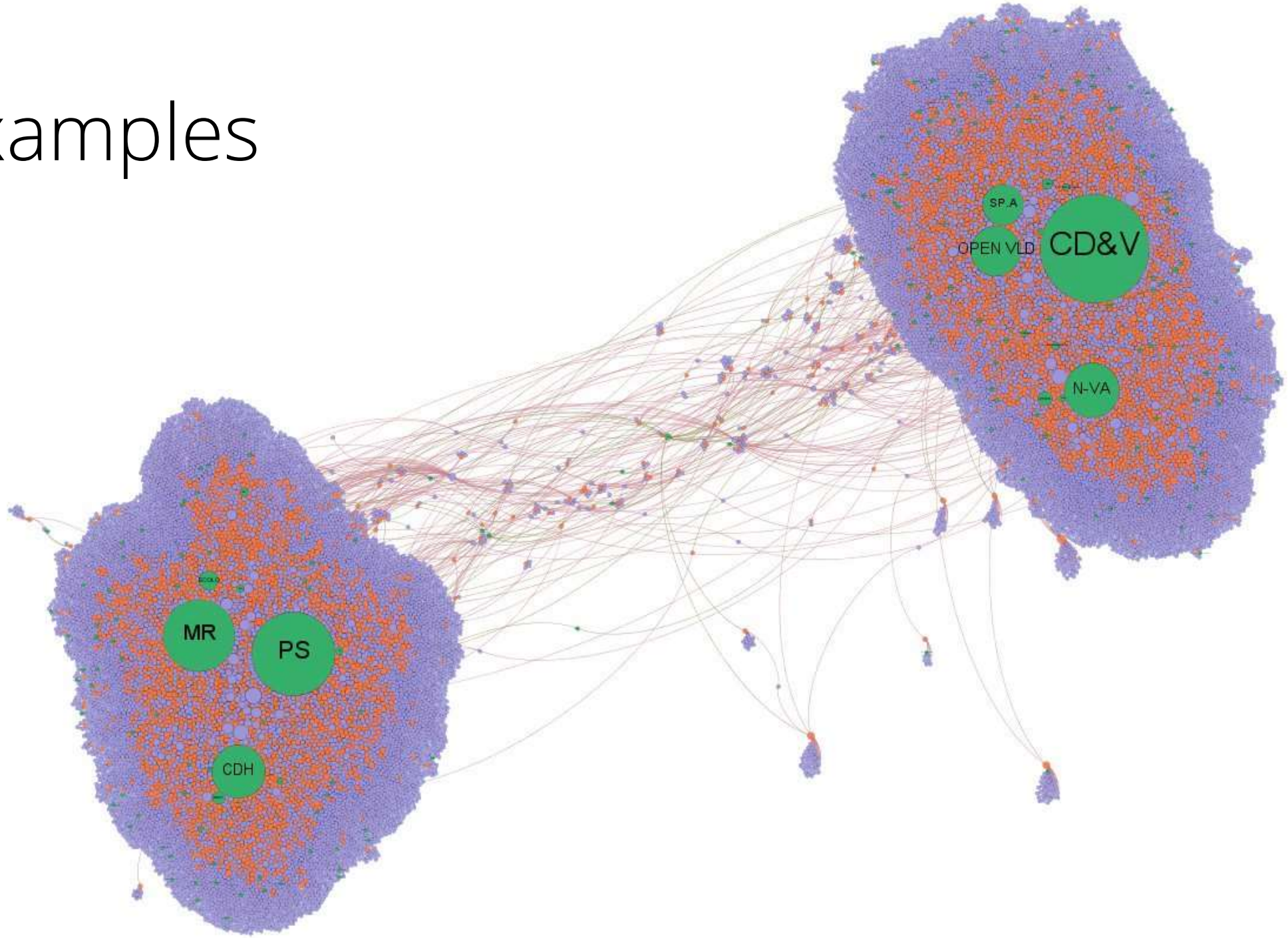
# Examples

- Scraper to download financial reports from National Bank to predict early bankruptcy
- Scraping job sites to identify trends within data science tooling
- Scraping news sites to analyze sentiment around Bitcoin
- Scraping news sites to compare biased and unbiased news source
- Scraping movie site to build recommender system
- Scraping political mandates and active political parties during voting season
- Scraping “curieuzeneuzen” air quality information

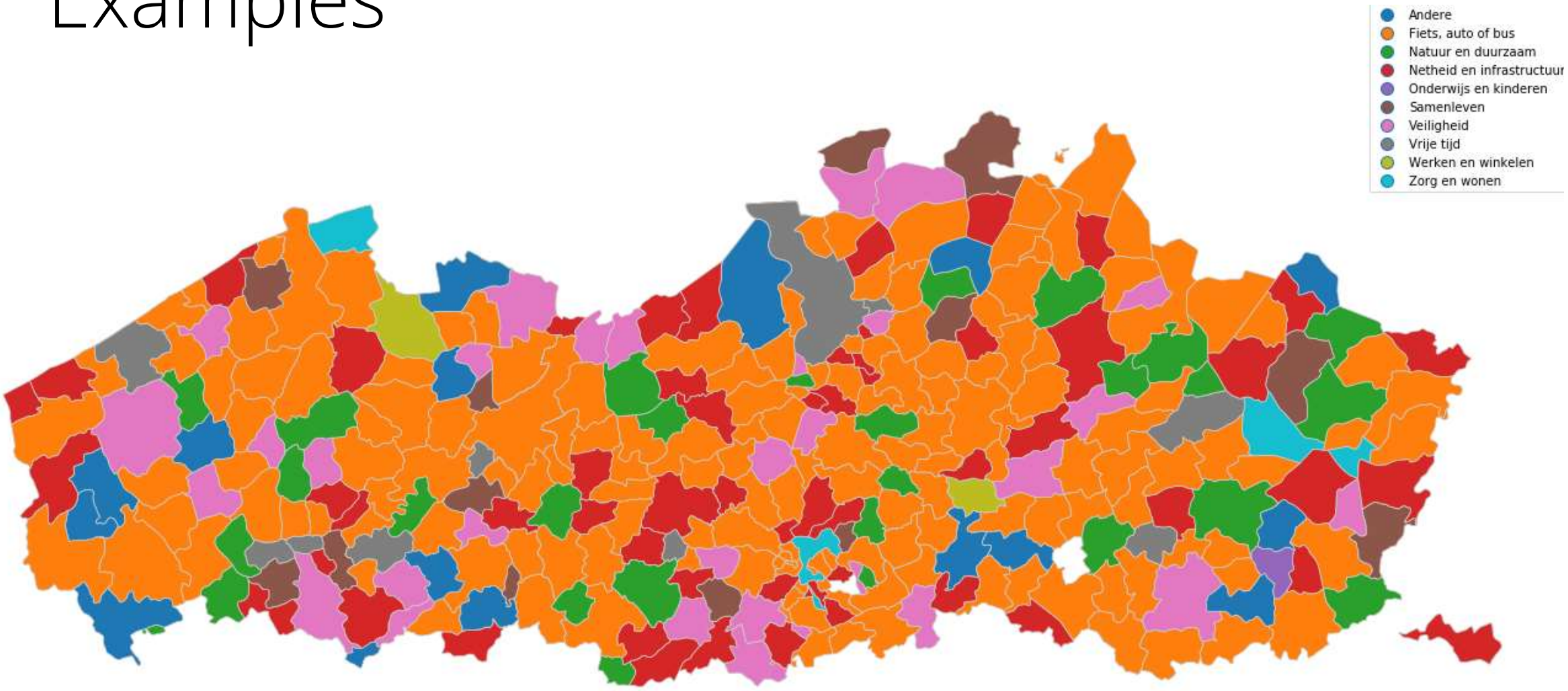
# Examples


gambit
pivot
preemptively
antagonize
rhetoric
bluster
treasonously
presser
policy
camerota
sciutto
shapiro
bartiromo
pirro
ramble
tantrum
tweetstorms
tweet
unhinge
trumpsters
lowness
gwb
drumpf
politically
trumpy
flip
unfairly
loser
brennan
unpresidential
hater
deplorables
supporter
administration
obama
reagan
clinton
presidency
candidacy
parscale
midterm
republican
trump
russia
pres
begala
accurso
antle
turnberry
inauguration
vuce
maga
knoller
zeleny
ioffe
haberman
thehill
house
congresscritters
himes
trumpers
drumbeat
dishonest
gasparino
trumpworld
wapo
axios
melbe
smerco
press
putin
trumpian
apoplectic
mccarthyism
ivanka
kellyanne
congressional
libs
nominee
moconnel
pataki
bharara
giulani
giuliani
penny

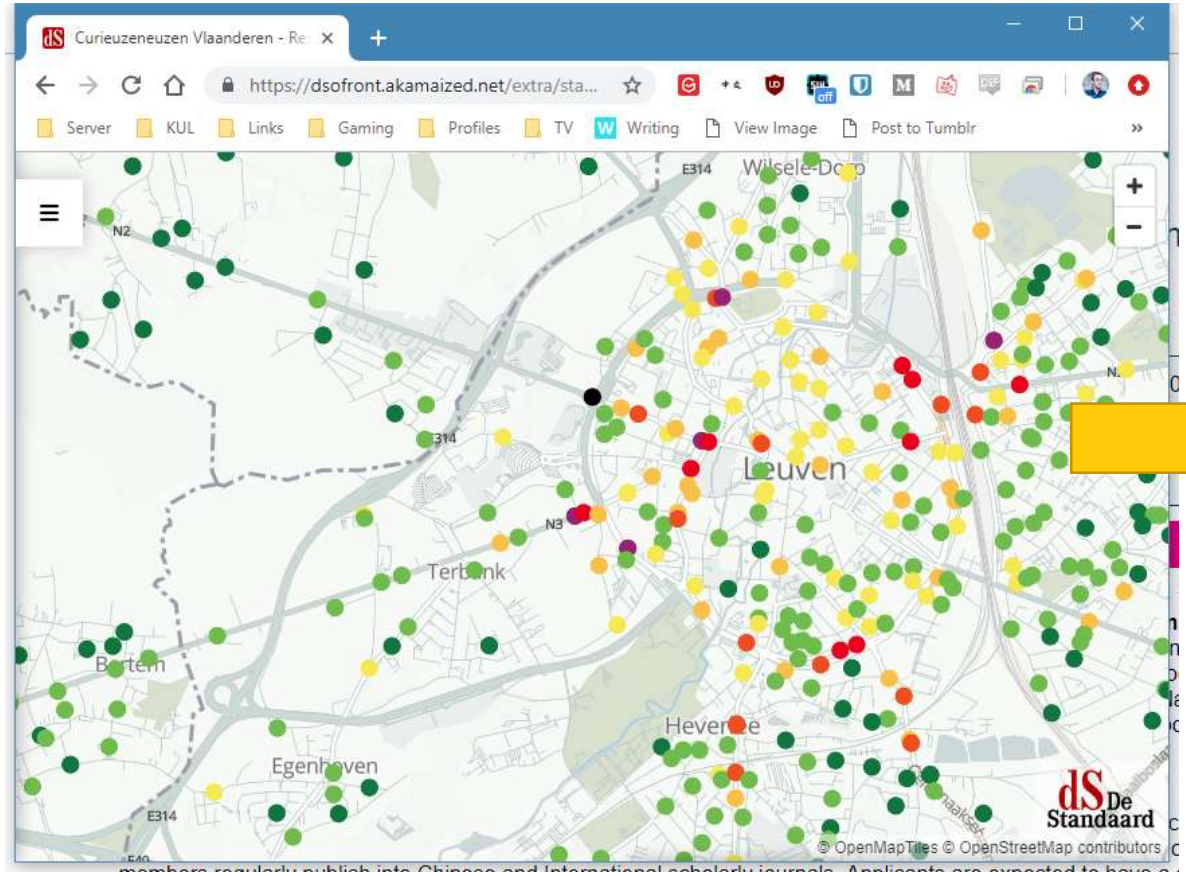
# Examples

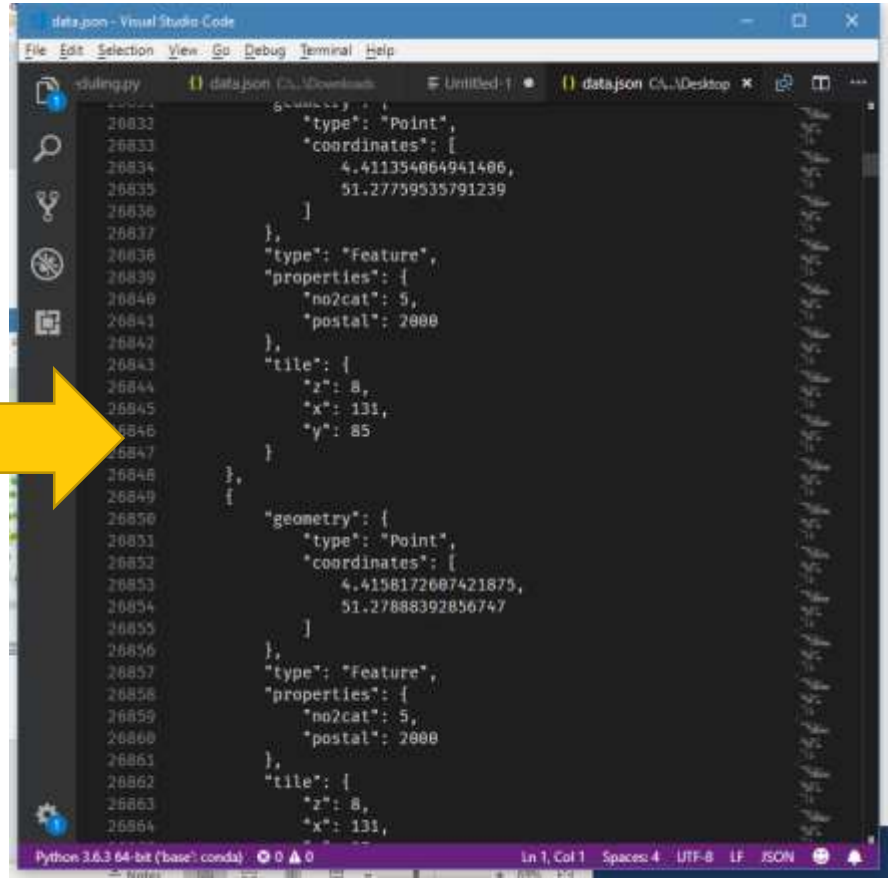


# Examples


Andere
Fiets, auto of bus
Natuur en duurzaam
Netheid en infrastructuur
Onderwijs en kinderen
Samenleven
Veiligheid
Vrije tijd
Werken en winkelen
Zorg en wonen

# Examples


Curieuzeneuzen Vlaanderen - Re
https://dsofront.akamaized.net/extra/sta...
Server
KUL
Links
Gaming
Profiles
TV
Writing
View Image
Post to Tumblr
E314
Wilsele-Dorp
N2
Leuven
N3
Terbank
Bertem
Heverlee
Egenhoven
E314
dS De Standaard
© OpenMapTiles © OpenStreetMap contributors


data.json - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
data.json
Untitled-1
data.json C:\...\Desktop
"type": "Point",
"coordinates": [
4.411354064941406,
51.27759535791239
]
},
"type": "Feature",
"properties": {
"no2cat": 5,
"postal": 2000
},
"tile": {
"z": 8,
"x": 131,
"y": 85
}
},
{
"geometry": {
"type": "Point",
"coordinates": [
4.4158172607421875,
51.27888392856747
]
},
"type": "Feature",
"properties": {
"no2cat": 5,
"postal": 2000
},
"tile": {
"z": 8,
"x": 131,
Python 3.6.3 64-bit ('base': conda)
Ln 1, Col 1
Spaces: 4
UTF-8
LF
JSON

# Web Scraping with Python

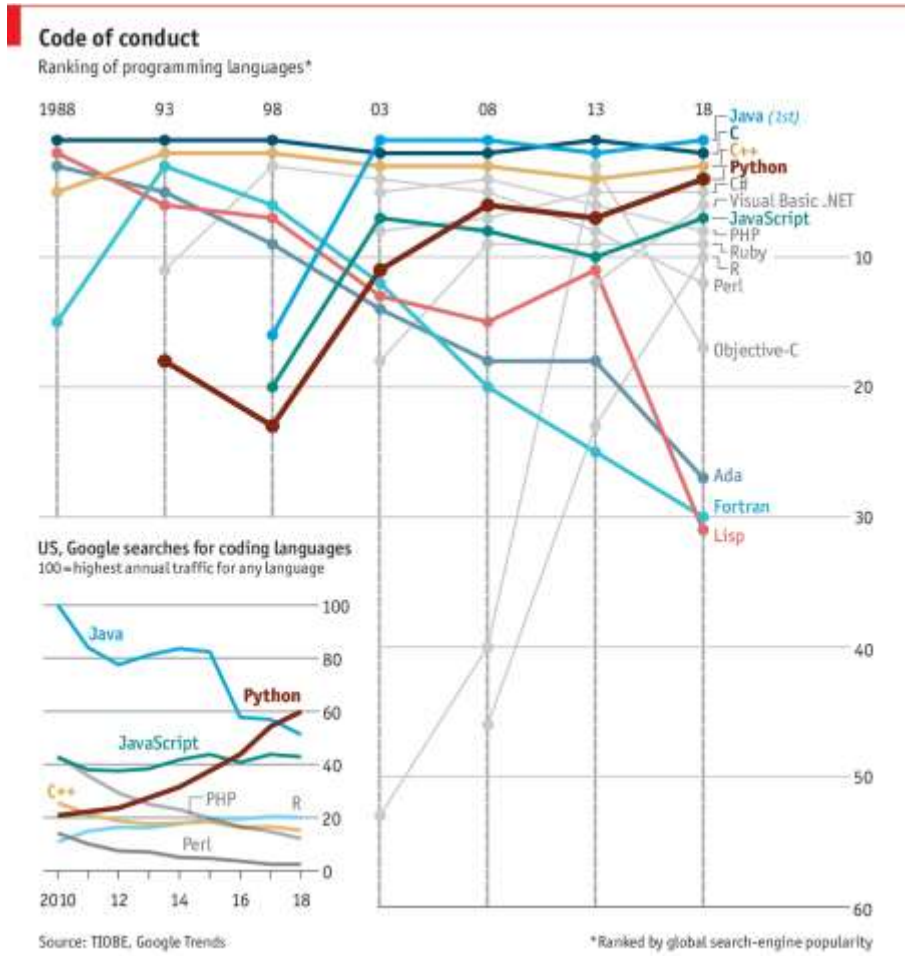
# Overview

- Why Python?
- The three core pillars: HTTP, HTML, and CSS
- The three core libraries: Requests, BeautifulSoup, Selenium
- Hands-on examples
- Other noteworthy libraries

# Why Python?

- https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language

# Why Python?


Code of conduct
Ranking of programming languages*
1988
93
98
03
08
13
18
Java (1st)
C
C++
Python
C#
Visual Basic .NET
JavaScript
PHP
Ruby
R
Perl
Objective-C
Ada
Fortran
Lisp
10
20
30
40
50
60
US, Google searches for coding languages
100=highest annual traffic for any language
Java
Python
JavaScript
C++
PHP
R
Perl
100
80
60
40
20
0
2010
12
14
16
18
Source: TIOBE, Google Trends
*Ranked by global search-engine popularity
The Economist

# Why Python?

- Widely used and increasingly getting more popular
- Known for its simplicity and flexibility (easy to learn, easy to do powerful things with)
- Support for many different platforms
- Increasingly being used in data science and AI settings (e.g. many deep learning frameworks like PyTorch, TensorFlow and Keras have Python bindings)
- Has many **well-written, elegant and powerful web scraping libraries** making it an excellent language in this setting

# Why Python?

- https://www.techrepublic.com/article/fastest-growing-programming-language-pythons-popularity-is-still-climbing/
  - “The current machine-learning boom has fueled a sharp uptick in the number of developers learning Python. Outside of the language's use in big-data analytics, Python's versatility is evident in its range of uses, from web and desktop apps to orchestrating system operations”

# Why Python?

- The Official Python 3 Documentation: https://docs.python.org/3/
- Dive Into Python 3: http://www.diveintopython3.net/index.html
- Automate the Boring Stuff with Python: https://automatetheboringstuff.com/
- The Hitchhiker's Guide to Python: http://docs.python-guide.org/en/latest/
- First Steps With Python: https://realpython.com/learn/python-first-steps/

# The three core pillars of the web

## HTTP

- Protocol used to communicate with web servers
- We'll need something that "speaks" HTTP

## HTML

- The format used to send web pages
- Semi-structured plain text format
- We'll need something that can parse this
- Other formats possible as well: e.g. downloading image, PDF, video files

## CSS

- Used in combination with HTML to style web pages
- We'll borrow the concept of CSS style selectors to quickly identify elements on a web page

## Others

- JavaScript: dynamic client side language to make web pages more interactive
- We might also want to use a database to store results, and learn a bit about other web query methods such as XPath

# The three core pillars of the web

## HTTP

- Protocol used to communicate with web servers
- We'll need something that "speaks" HTTP

## HTML

- The format used to send web pages
- Semi-structured plain text format
- We'll need something that can parse this
- Other formats possible as well: e.g. downloading image, PDF, video files

## CSS

- Used in combination with HTML to style web pages
- We'll borrow the concept of CSS style selectors to quickly identify elements on a web page

## Others

- JavaScript: dynamic client side language to make web pages more interactive
- We might also want to use a database to store results, and learn a bit about other web query methods such as XPath

# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser

Web browser

User enters “google.com”

# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser

Web browser

Web browser needs to figure out the IP address for “google.com”
IP stands for “Internet Protocol” IP (Internet Protocol) and forms a core protocol of the Internet,
as it enables networks to route and redirect communication packets between connected computers,
which are all given an IP address

# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser

Web browser

Web browser inspects its local cache
If the website was recently visited, a copy of the IP address will be stored in the browser's cache

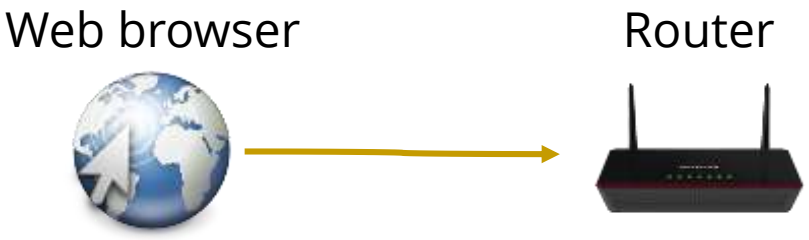
# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser


Web browser requests IP address from operating system
To do so, your web browser will use another protocol, called DNS (which stands for Domain Name System)

# The web speaks HTTP

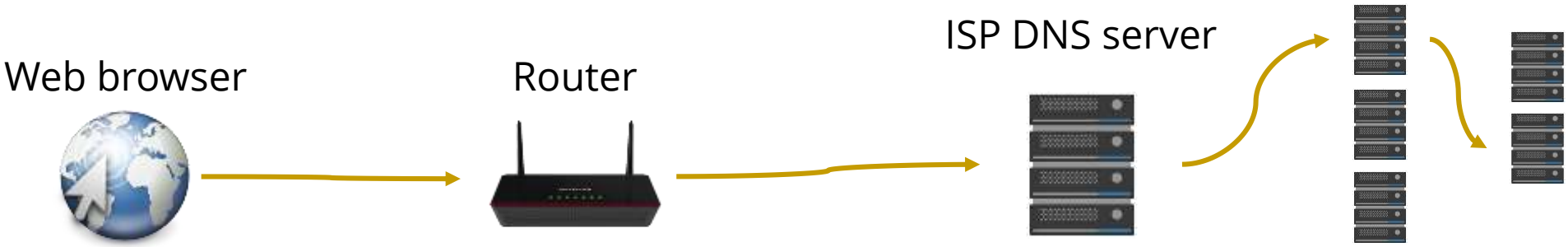
- A lot happens when you navigate to a web page in the browser




Web browser requests IP address from router
To do so, your web browser will use another protocol, called DNS (which stands for Domain Name System)

# The web speaks HTTP

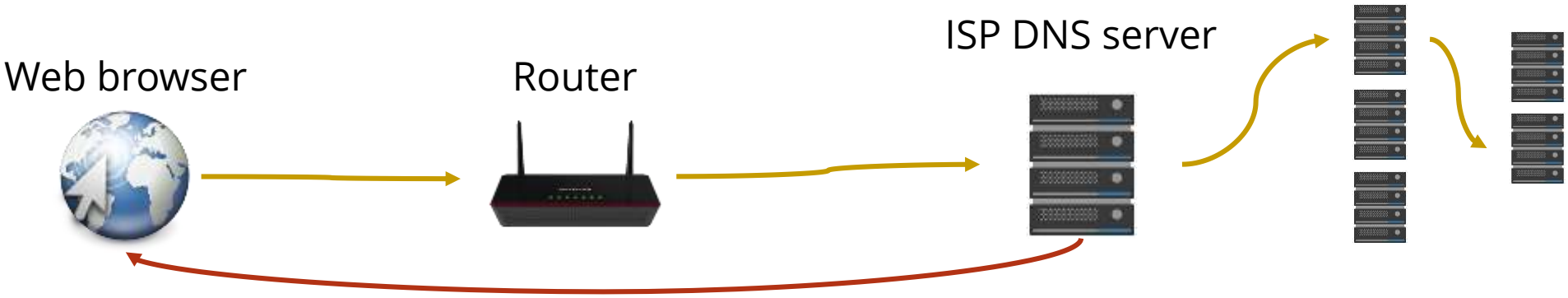
- A lot happens when you navigate to a web page in the browser




Request gets forwarded to your ISP's DNS server (which also has an IP address stored in your OS or router)
Note that this server might also forward the request to other DNS servers (these form a hierarchy with several main "root" DNS servers at the top: the "telephone" book of the Internet)
Many organizations also maintain their own DNS servers, e.g. Google has one running at 8.8.8.8

# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser




Finally, the IP address (172.217.17.68) is retrieved and returned to the OS / browser

So far, this conversation itself is not really specific to web browsing, many other Internet applications (email, voice calls, …) will also need to make use of this system to map domains to IP addresses

# The web speaks HTTP

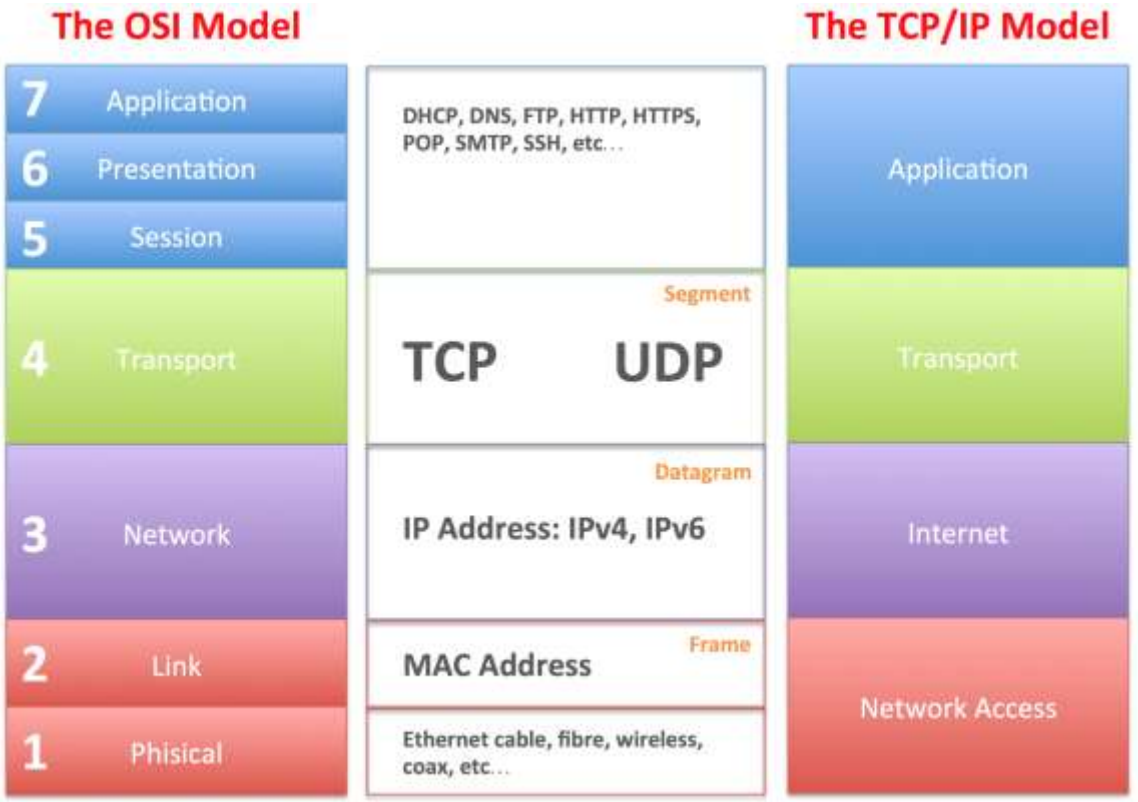
- A lot happens when you navigate to a web page in the browser


Your browser can now establish a connection to Google's web server

A number of protocols are combined here to construct a complex message. At the outermost part, we find the IEEE 802.3 (Ethernet) protocol, which is used to communicate with machines on the same network. Since we're not communicating on the same network, the Internet Protocol, IP (again) is used to embed another message indicating that we wish to contact the server at address 172.217.17.68.
Inside this, we find another protocol, called TCP (Transmission Control Protocol), which provides a general, reliable means to deliver network messages. Finally, inside the TCP message, we find another message, formatted according to the HTTP protocol (HyperText Transfer Protocol), which is the actual protocol used to request and receive web pages.

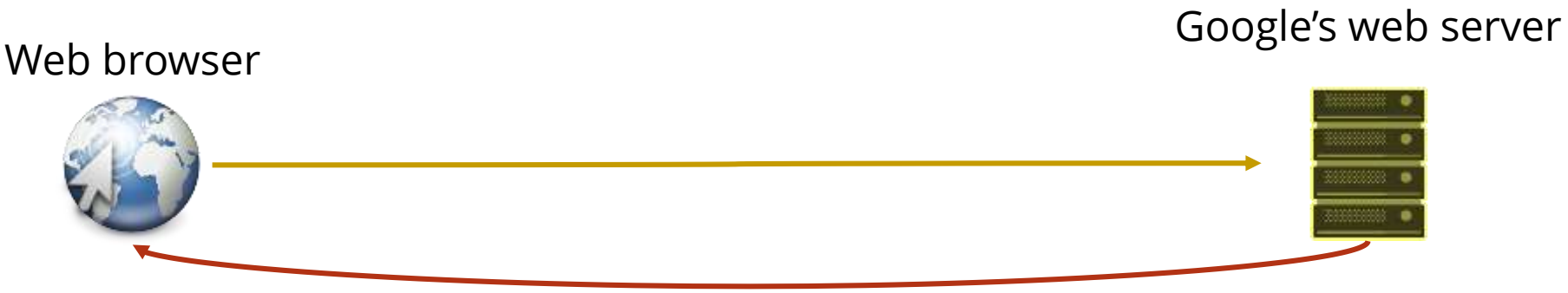
# The web speaks HTTP

- A lot happens when you navigate to a web page in the browser



# The web speaks HTTP

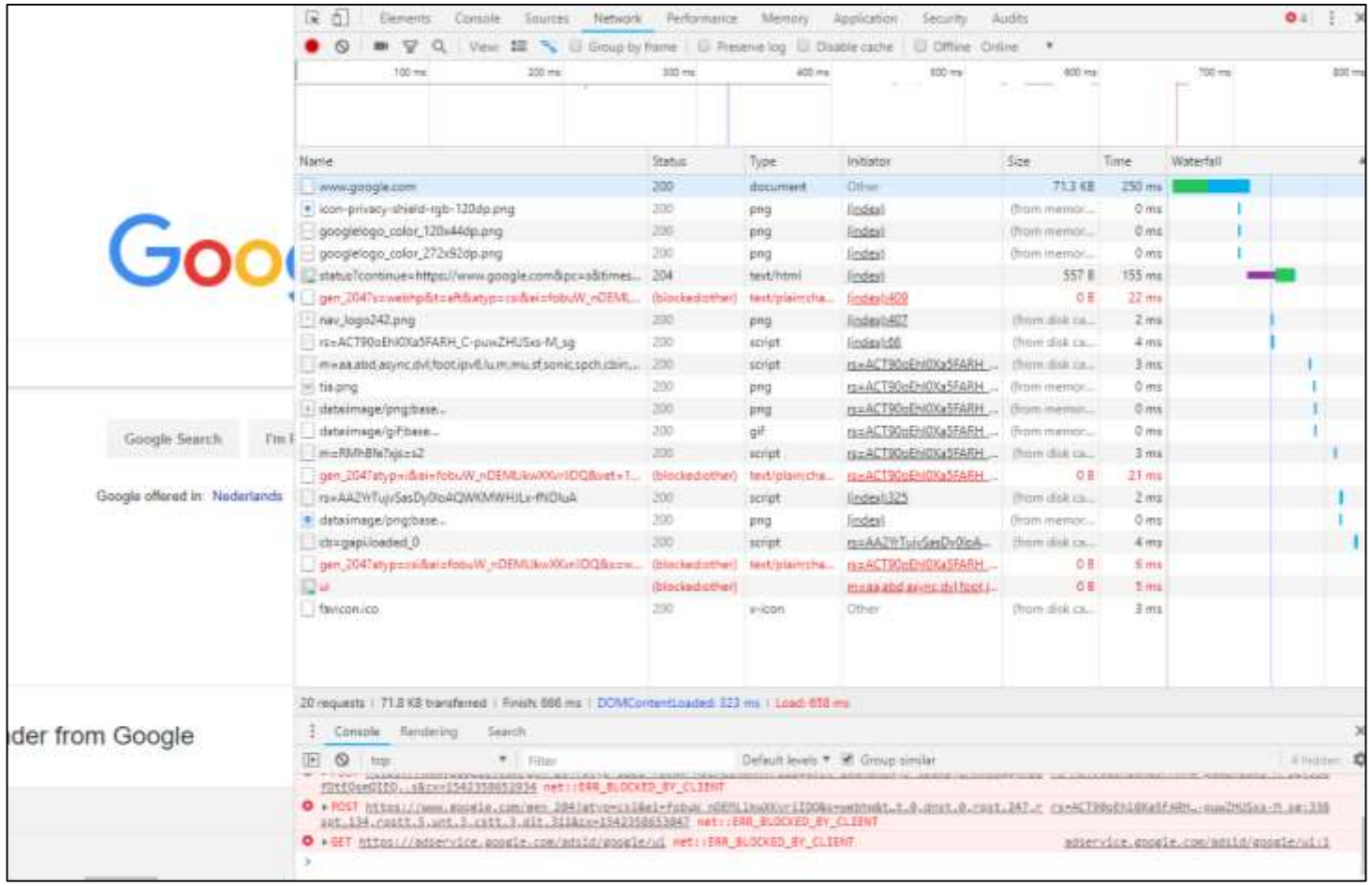
- A lot happens when you navigate to a web page in the browser




Google's web server now sends back an HTTP reply, containing the contents of the page we want to visit

In most cases, this textual content is formatted using HTML. From this (oftentimes large) bunch of text, our web browser can set off to render the actual page, i.e. making sure that everything appears neatly on screen as instructed by the HTML content. Note that a web page will oftentimes contain pieces of content for which the web browser will initiate new HTTP requests

# The web speaks HTTP



# The web speaks HTTP

- HTTP request format

- ```
  GET /docs/index.html HTTP/1.1                    Request method, url, version
  Host: www.example.com                            Request headers (host is mandatory)
  Accept: image/gif, image/jpeg, */*                        …
  Accept-Language: en-us                                    …
  Accept-Encoding: gzip, deflate                            …
  User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
  <blank line>
  <message body, optional, can span multiple lines>
  ```

- Every line is ended with <CR><LF> (the ASCII characters 0D and 0A)
  - The empty line is simply <CR><LF> with no other additional white space
  ```

# The web speaks HTTP

- Each header includes a name, followed by a colon (“:”) and the actual value of the header
  - Browsers are very chatty in terms of what they like to include in their headers (e.g. happily report their version through the User-Agent header)
  - The HTTP standard includes some headers which are standardized and which will be utilized by proper web browsers, though you are free to include additional headers as well
    - “Host”, for instance, is a standardized and mandatory header in HTTP 1.1 and higher

# The web speaks HTTP

- Some common request headers
  - “Host” to indicate which domain the request relates to
  - “Connection: keep-alive” signposts to the server that it should keep the connection open for subsequent requests if it can
  - The “User-Agent” contains a large text value through which the browser informs the server what it is and which version it is running as
    - Also often includes fields referring to other browser names to avoid broken User-Agent checks on websites
  - “Accept” tells the server which forms of content the browser prefers to get back
  - “Accept-Encoding” tells the server that the browser is also able to get back compressed content
  - The “Referer” header (a misspelling) tells the server from which page the browser comes from

- Even though your web browser will try to behave politely and, for instance, tell the web server which forms of content it accepts, there is no guarantee that a web server will actually look at these headers or follow up on them
  - A browser might indicate in its “Accept” header that it understands “webp” images, but the web server can just ignore this request and send back images as “jpg” or “png” anyway
  - Consider these request headers as polite requests

# The web speaks HTTP

- HTTP 0.9 and 1.0
  - Not used anymore
- HTTP 1.1
  - Thanks to the Host header, the ability to host different domains at the same IP address
  - A TCP connection can (but doesn't have to) be reused, saving the time to reopen it numerous times to display the resources embedded into the single original document retrieved
  - Caching support
  - Widely used and extended
- HTTP 2.0
  - A binary protocol rather than text. It can no longer be read and created manually
  - Faster, using parallel requests over the same connection, removing order and blocking constraints
  - Header compression
  - Limited support in libraries
- HTTP 3.0
  - Based on QUIC
  - Switch to UDP instead of TCP
  - More encryption support (merging of HTTP and HTTPS concerns)

# The web speaks HTTP

```
import socket


HOST = 'example.org'
PORT = 80


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    sock.sendall(b'GET / HTTP/1.1\r\n' +
                 b'Host: example.org\r\n' +
                 b'User-Agent: Python 3\r\n' +
                 b'\r\n')
    data = sock.recv(1024 * 10)


print(data.decode('utf-8'))
```

# The web speaks HTTP

```
HTTP/1.1 200 OK
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Sat, 10 Nov 2018 15:59:57 GMT
Etag: "1541025663+ident"
Expires: Sat, 17 Nov 2018 15:59:57 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (dca/2486)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 1270

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <style type="text/css">
    […]
    </style>
</head>

<body>
<div>
    <h1>Example Domain</h1>
    <p>This domain is established to be used for illustrative examples in documents. You may
       use this domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

# The web speaks HTTP

```
import socket


HOST = 'seppe.net'
PORT = 80


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT))
    sock.sendall(b'GET /seppe/cv.png HTTP/1.1\r\n' +
                 b'Host: seppe.net\r\n' +
                 b'User-Agent: Python 3\r\n' +
                 b'\r\n')
    data = sock.recv(1024 * 10)


print(data)
```

# The web speaks HTTP

```
HTTP/1.1 200 OK
Date: Sat, 10 Nov 2018 16:03:41 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Mon, 17 Aug 2015 19:54:14 GMT
ETag: "43aff-51d872a7cd27d“
Accept-Ranges: bytes
Content-Length: 277247
Content-Type: image/png

\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x01K\x00\x00\x01\xa9\x08\x02\x00\x00\x00\x8bY\x83@\x00\x00\x00\x04gAMA\x00\x00
\xb1\x8f\x0b\xfca\x05\x00\x00\x00\tpHYs\x00\x00\x0e\xc2\x00\x00\x0e\xc2\x01\x15(J\x80\x00\x00\x00\x16tEXtSoftware\x00paint
.net
4.0;\xe8\xf5i\x00\x00\xff\x90IDATx^t\xfd\x05\x97$\xc7\x15\xad\x81\xea\x9f\\[83\x8d\xc5LY\xcc\\]\xd8\xcc\xcc\xcc\xdc=\xcc\x
cc3\x92,\x181\xa3e\xc9\x92,\xdb\xb2\xc5`1\xcb\xb6lY\x8c\xf3\xf69Q=\xd2\xbd\xef\xbd\xb5\xce\xca\x15\x95\x95\x05]\x9d_\xec}N
DF^v\xac\xd9q\xb2\xc9y\xaa\xd5}\xba\xd5}\xb2\xdd}\xba\r\xe19\xd3\xee=\xdb\xe1\xe3\xf0\x9f\xe9\xf4\x9e\xed\xf2\x9f\xed\x0c\
x9c\xed\n\x9e\xef\x0c\x9dG\xa3\xc3\x8b\x83\x8f\xb7\xb8O69\x8e6\xd8\x0f\xd4I\x87\xaam\xfb\x0b\x96=Y\xf3\xee\x8cagZ\xc7\xa1\
xdf\x9d6\xe0\xe1\xae4\x05\x1a{2\xd8\xeawg\r{r\xc6\xfdY\x84i
```

# The web speaks HTTP

- HTTP reply format

- HTTP/1.1 200 OK
  Date: Sat, 10 Nov 2018 16:03:41 GMT
  Server: Apache/2.4.18 (Ubuntu)
  Last-Modified: Mon, 17 Aug 2015 19:54:14 GMT
  ETag: "43aff-51d872a7cd27d“
  Accept-Ranges: bytes
  Content-Length: 277247
  Content-Type: image/png
  **<blank line>**
  \x89PNG\r\n\x1a

**Version, status code and message**
**Reply headers**
…
…
…
…
…
…
…

**Message body (optional)**

# The web speaks HTTP

- Also here, the reply contains a lot of chatty headers
  - E.g. the server happily reports its version in the “Server” header
  - Other headers are pretty common as well
    - Content-Length
    - Content-Type: provides a hint to the browser what content will be in the message body
    - Date
    - Last-Modified
- Best known status codes: 200 and 404 (page not found)
  - Even for a 404 status, web servers will often return a message body with a message the browser can display (e.g. a nicer looking page not found page)
    - In some cases there is no message body

# HTTP in Python

- Recall the main purpose of web scraping: to retrieve data from the web in an automated manner
  - Basically, we're going to surf the web using a Python program
  - This means that our Python program will need to be able to speak and understand HTTP
  - We could continue to program this ourselves on top of standard networking functionality already built-in in Python, making sure that we neatly format HTTP request messages and are able to parse the incoming responses
  - However, we're not interested in re-inventing the wheel, and there are many Python libraries out there already that make this task a lot more pleasant: HTTP libraries

# HTTP in Python

- HTTP libraries
  - Python 3 comes with a built-in module called “urllib” which can deal with all things HTTP (see https://docs.python.org/3/library/urllib.html)
    - The module got heavily revised compared to its counterpart in Python 2, where HTTP functionality was split up in both “urllib” and “urllib2” and somewhat cumbersome to work with (especially urllib2 consisted of a lot of hacky code)
  - “httplib2” (see https://github.com/httplib2/httplib2): a small, fast HTTP client library. Originally developed by Googler Joe Gregorio, and now community supported
  - “urllib3” (see https://urllib3.readthedocs.io/): a powerful HTTP client for Python, used by the requests library below
  - “requests” (see http://docs.python-requests.org/): an elegant and simple HTTP library for Python, built “for human beings”

# HTTP in Python


- We will use “requests”
  - HTTP for humans: flexible and fun library to program in
  - Well-documented
  - Large feature-set: Keep-Alive support, sessions to support cookie-management, SSL support (HTTPS), automatic content decoding, HTTP auth support, compression (gzip) support, proxy support, Unicode support, streaming downloads
  - “urllib” provides solid HTTP functionality, but still involves lots of boilerplate code making the module less pleasant to use and not very elegant to read
    - “urllib3” (not part of the standard Python modules) extends the Python ecosystem regarding HTTP with some advanced features, but also doesn't really focus that much on being elegant or concise. “requests” builds on top of “urllib3”, but allows you to tackle the majority of HTTP use cases in code that is short, pretty, and easy to use
  - See also: “grequests” and “aiohttp”
    - More modern-oriented libraries and aim to make HTTP with Python more asynchronous
    - This becomes especially important for very heavy-duty applications
  - **HTTP 1.1, not HTTP 2!**

# HTTP in Python

```
import urllib.parse
import urllib.request

url = 'http://www.webscrapingfordatascience.com/postform2/'

formdata = {
   'name': 'Seppe',
   'gender': 'M',
   'pizza': 'like',
   'haircolor': 'brown',
   'comments': ''
}

data = urllib.parse.urlencode(formdata).encode("utf-8")
req = urllib.request.Request(url, data)
response = urllib.request.urlopen(req)
text = response.read()

print(text)
```

```
import requests

url = 'http://www.webscrapingfordatascience.com/postform2/’

formdata = {
   'name': 'Seppe',
   'gender': 'M',
   'pizza': 'like',
   'haircolor': 'brown',
   'comments': ‘’
}

text = requests.post(url, data=formdata).text



print(text)
```

# Using requests

- Standard usage (simple GET request)

```
import requests


url = 'http://www.webscrapingfordatascience.com/basichttp/'
r = requests.get(url)
print(r.text)
```

# Using requests

- Using the response object

```
import requests

url = 'http://www.webscrapingfordatascience.com/basichttp/'
r = requests.get(url)

# Which HTTP status code did we get back from the server?
print(r.status_code)
# What is the textual status code?
print(r.reason)
# What were the HTTP response headers?
print(r.headers)
# The request information is saved as a Python object in r.request:
print(r.request)
# What were the HTTP request headers?
print(r.request.headers)
# The HTTP response content:
print(r.text)
```

# Using requests

- URL parameters (the query string)

```
import requests

url = 'http://www.webscrapingfordatascience.com/paramhttp/?query=test'
r = requests.get(url)
print(r.text)
# Will show: I don't have any information on "test"

url = 'http://www.webscrapingfordatascience.com/paramhttp/?query=a query with spaces'
r = requests.get(url)
# Parameter will be encoded as 'a%20query%20with%20spaces'
# You can verify this be looking at the prepared request URL:
print(r.request.url)
# Will show [...]/paramhttp/?query=a%20query%20with%20spaces
print(r.text)
# Will show: I don't have any information on "a query with spaces"
```

# Using requests

- URL parameters (the query string): a case where it doesn't work

```
import requests


url = 'http://www.webscrapingfordatascience.com/paramhttp/?query=complex?&'
# Parameter will not be encoded
r = requests.get(url)
# You can verify this be looking at the prepared request URL:
print(r.request.url)
# Will show [...]/paramhttp/?query=complex?&
print(r.text)
# Will show: I don't have any information on "complex?"
```

# Using requests

- URL parameters (the query string): using urllib

```
from urllib.parse import quote, quote_plus


raw_string = 'a query with /, spaces and?&'
print(quote(raw_string))         # generally used in URL paths
print(quote_plus(raw_string))    # generally used in query string


# a%20query%20with%20/%2C%20spaces%20and%3F%26
# a+query+with+%2F%2C+spaces+and%3F%26
```

# Using requests

- URL parameters (the query string): a better way

```
import requests

url = 'http://www.webscrapingfordatascience.com/paramhttp/'

parameters = {
    'query': 'a query with /, spaces and?&'
}

r = requests.get(url, params=parameters)

print(r.url)
print(r.text)
```

# Using requests

```
import requests


url = 'https://news.ycombinator.com/'


r = requests.get(url)

print(r.text)


# How do we parse the HTML?
```

# The three core pillars of the web

## HTTP

- Protocol used to communicate with web servers
- We'll need something that "speaks" HTTP

## HTML

- The format used to send web pages
- Semi-structured plain text format
- We'll need something that can parse this
- Other formats possible as well: e.g. downloading image, PDF, video files

## CSS

- Used in combination with HTML to style web pages
- We'll borrow the concept of CSS style selectors to quickly identify elements on a web page

## Others

- JavaScript: dynamic client side language to make web pages more interactive
- We might also want to use a database to store results, and learn a bit about other web query methods such as XPath

# Parsing HTML

- We need something that can make sense of the HTML “soup”
- Not: regular expressions (can be used for selection and some cleaning, but not for full cleaning)
- Convert HTML DOM tree to structured object
- Quick selection of elements using CSS selectors or XPath queries, DOM tree navigation
- We need an HTML parsing library

# Parsing HTML

- We will use BeautifulSoup
  - https://www.crummy.com/software/BeautifulSoup/bs4/doc/
  - Relatively easy to use and learn, though it has some gotchas (e.g. complex selection with CSS selectors, combining filters)
  - Comes with some overhead, as it itself wraps around another XML/HTML tree parser (such as lxml or html5lib), so that some people ultimately prefer to drop it
    - Nevertheless still a very solid starting point

# Parsing HTML

- Other libraries
  - parse (https://pypi.python.org/pypi/parse): “the opposite of format()”:
    - `search('Age: {:d}\n', 'Name: Rufus\nAge: 42\nColor: red\n')`
    - For textual searches
  - pyquery (http://pyquery.readthedocs.io/en/latest/): “searching using jQuery style selectors”
    - Better CSS selector support
  - parsel (https://parsel.readthedocs.io/en/latest/): “search using XPath and CSS selectors”
    - Better CSS selector and XPath support
  - Cleaner libraries (Cleaner in lxml, htmllaundry, html2text, …)
  - These can be combined with BeautifulSoup to refine found information

# Using BeautifulSoup

- Using BeautifulSoup

```
import requests
from bs4 import BeautifulSoup

url = 'https://sai.be/home'

r = requests.get(url)
html_contents = r.text

html_soup = BeautifulSoup(html_contents, 'html.parser')

# built-in Python parser
# or lxml (fast), or html5lib (slower but very browser-similar): requires installation
```

# Using BeautifulSoup

- The two most important methods:
  - `find(name, attrs, recursive, string, **keywords)`
  - `find_all(name, attrs, recursive, string, limit, **keywords)`
    - `findAll` also allowed

# Using BeautifulSoup

- The **name** argument defines the tag names you wish to “find” on the page
  - You can pass a string, or a list of tags, or a regular expression, or a function
  - Leaving this argument as an empty string simple selects all elements
- The **attrs** argument takes a Python dictionary of attributes and matches HTML elements that match those attributes
- The **recursive** argument is a boolean and governs the depth of the search
  - If set to True (default), the find and find_all methods will look into children, children’s children, and so on… for elements that match your query. If it is False, it will only look at direct child elements
- The **string** argument is used to perform matching based on the text content of elements
  - In earlier Beautiful Soup versions, this argument was named text instead (you can still use this as well)
  - **Careful with nested tags: only matches on direct text descendants**
- The **limit** argument is only used in the find_all method and can be used to limit the number of elements that are retrieved
  - Note that find is functionally equivalent to calling find_all with the limit set to 1, with the exception that the former returns the retrieved element directly, and that the latter will always return a list of items, even if it just contains a single element. Also, when find_all cannot find anything, it returns an empty list, whereas if find cannot find anything, it returns None
- ****keywords** indicates that you can add in as much extra named arguments as you like, which will then simply be used as attribute filters
  - Offered as convenience; if you define both the attrs argument and extra keywords, all of these will be used together as filters
  - **Take care: find(class_='myclass'), and no name_ alternative**

# Using BeautifulSoup

- Both find and find_all return Tag objects, which expose
  - The **name** attribute to retrieve the tag name
  - The **contents** attribute to get a Python list containing the tag's children (its direct descendant tags) as a list
    - The **children** attribute does the same but provides an iterator instead
    - The **descendants** attribute also returns an iterator, now including all the tag's descendants in a recursive manner
    - The **parent** and **parents** attributes go “up the tree”
    - **next_sibling**, **previous_sibling** and **next_siblings** and **previous_siblings** can be used to go sideways
  - Converting the Tag object to a string shows both the tag and its HTML content as a string
  - Access the attributes of the element through the **attrs** attribute of the Tag object
    - For the sake of convenience, you can also directly use the Tag object itself as a dictionary
  - Use the **text** attribute to get the contents of the Tag object as clear text
    - Alternatively, you can use the get_text method as well, to which a strip boolean argument can be given: note that get_text(strip=True) strips more than simply text.strip()
    - It's also possible to specify a string to be used to join the bits of text enclosed in the element together, e.g. get_text('--')
    - If a tag only has one textual child, then you can also use the **string** attribute to get the textual content. However, in case a tag contains other HTML tags nested within, string will return None whereas text will recursively fetch all the text: note the difference in behavior compared to the argument before
  - Every Tag object itself can be used as a new root from which new searches can be started

# Using BeautifulSoup

```
tag.find('div').find('table').find('thead').find('tr')
```

Is the same as:

```
tag.div.table.thead.tr
```

```
tag.find_all('h1')
```

Is the same as calling:

```
tag('h1')
```

# Using BeautifulSoup

- For HTML attributes that can take multiple, space separated values (such as “class”), Beautiful Soup will perform **a partial match**
  - This can be tricky in case you want to perform an exact match such as “find me elements with the “myclass” class and only that class”, but also in cases where you want to select an HTML element matching more than one class
  - In this case, you can write something like “find(class_='class-one class-two')”, though this way of working is rather brittle and dangerous (the classes should then **appear in the same order and next to each other in the HTML page**, which might not always be the case)
  - Another approach is to wrap your filter in a list, i.e. “find(class_=['class-one', 'class-two'])”, though this will **also not obtain the desired result**: instead of matching elements having both “class-one” and “class-two” as classes, this statement will match with **elements having any of these classes**

# Using BeautifulSoup

- A basic example

```
import requests
from bs4 import BeautifulSoup

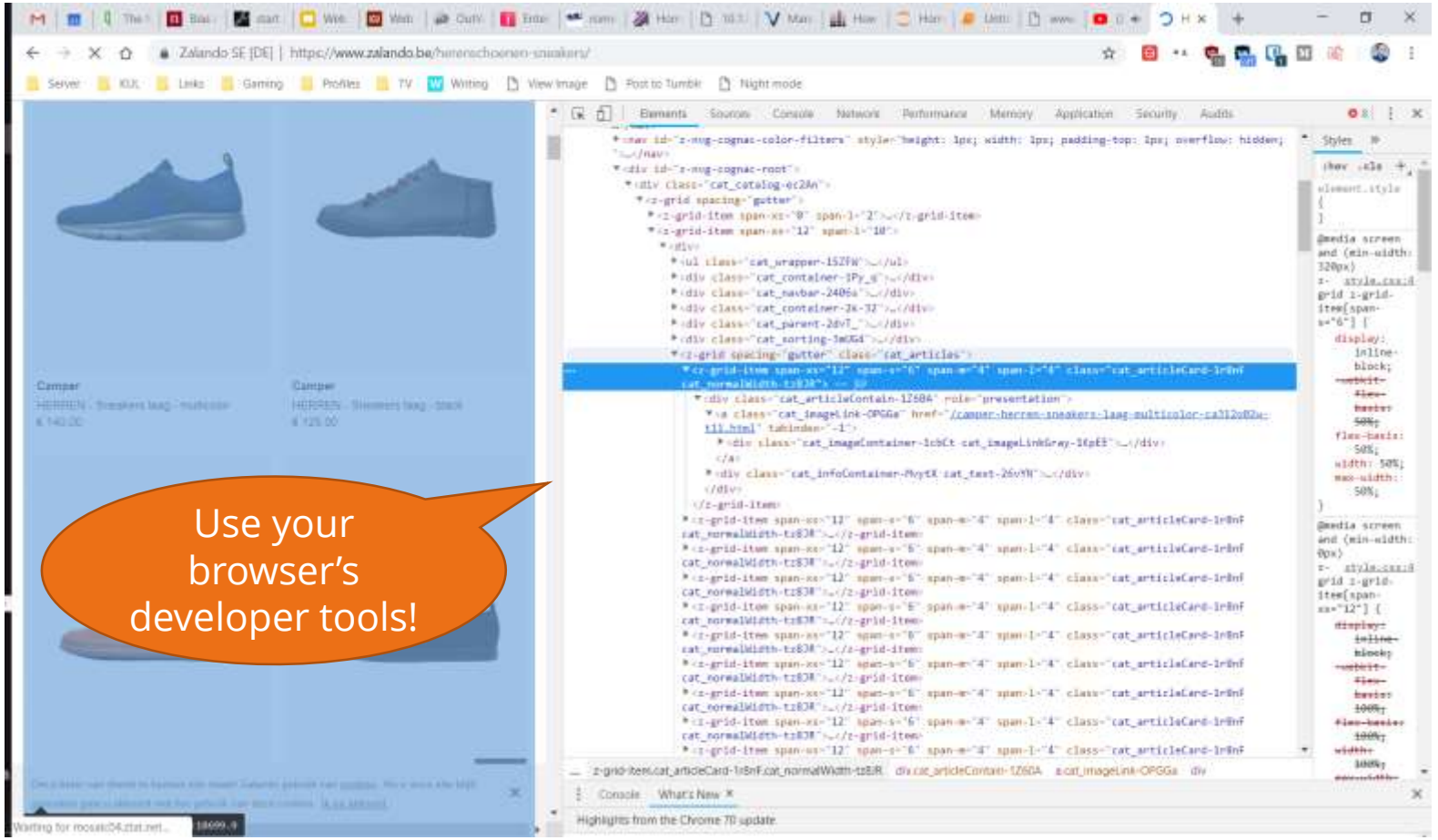
url = 'https://www.zalando.be/herenschoenen-sneakers/'

r = requests.get(url)
html_contents = r.text

html_soup = BeautifulSoup(html_contents, 'html.parser')

for item in html_soup.find('', class_='cat_articles').find_all('z-grid-item'):
  article = item.find('', class_='cat_articleName--arFp')
  if not article: continue
  print(article.text)
```

# Using BeautifulSoup



# Using BeautifulSoup

- This doesn't work as expected?

```
import requests
from bs4 import BeautifulSoup

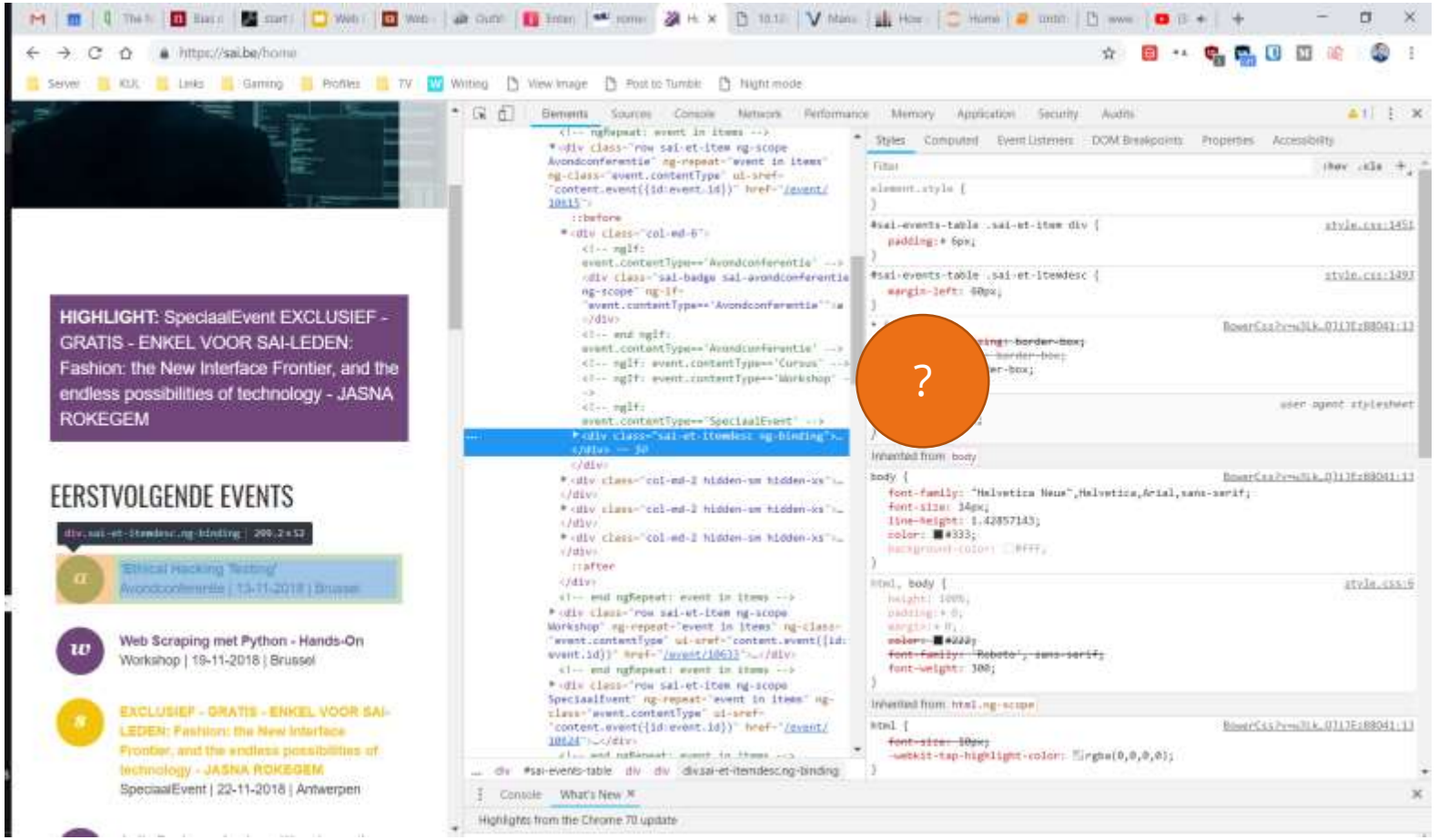
url = 'https://sai.be/home/'

r = requests.get(url)
html_contents = r.text

html_soup = BeautifulSoup(html_contents, 'html.parser')

for item in html_soup.find_all('div', class_='sai-et-item'):
  print(item)
```

# Using BeautifulSoup



# Using BeautifulSoup

- Inspect tool in browser shows the page as currently rendered
  - Including dynamic changes by JavaScript
  - A “live view”

- requests and BeautifulSoup cannot render JavaScript
  - Static view, as the page came in
  - “View source” in browser

# Using BeautifulSoup

- Using regular expression:

```
import re

html_soup.find(re.compile('^h'))
```

- Using filter functions:

```
def has_classa_but_not_classb(tag):
    cls = tag.get('class', [])
    return 'classa' in cls and not 'classb' in cls
html_soup.find(has_classa_but_not_classb)
```

# Using BeautifulSoup

- **find and find_all** work their way down the tree, looking at a tag's **children/descendants**
  - Remember that find and find_all work on the children attribute in case the recursive argument is set to False, and on the descendants attribute in case recursive is set to True
  - This are the methods you'll use most
- **find_parent and find_parents** work their way up the tree, looking at a tag's parents using its **parents** attribute
- **find_next_sibling and find_next_siblings** will iterate and match a tag's siblings using the **next_siblings** attribute
- **find_previous_sibling and find_previous_siblings** do the same but use the **previous_siblings** attribute
- **find_next and find_all_next** use the **next_elements** attribute to iterate and match over whatever comes after a tag
- **find_previous and find_all_previous** will perform the search backwards using the **previous_elements** attribute

- Although it's not really documented, it is also possible to use the findChild and findChildren methods (though not find_child and find_children), which are defined as aliases for find and find_all respectively
  - There is no findDescendant, however, so keep in mind that using findChild will default to searching throughout all the descendants (just like find does), unless you set the recursive argument to False
  - This is confusing, so its best to avoid these methods

# Using BeautifulSoup

- In most cases, start with:
  - find
  - find_all
  - name and text attributes, get_text method
  - attrs access as dictionary

# Using BeautifulSoup

- There is one more method, however, which is useful: **select**

- Using this method, **you can simply pass a CSS selector rule as a string**
  - Beautiful Soup will return a list of elements matching this rule

# The three core pillars of the web

## HTTP

- Protocol used to communicate with web servers
- We'll need something that "speaks" HTTP

## HTML

- The format used to send web pages
- Semi-structured plain text format
- We'll need something that can parse this
- Other formats possible as well: e.g. downloading image, PDF, video files

## CSS

- Used in combination with HTML to style web pages
- We'll borrow the concept of CSS style selectors to quickly identify elements on a web page

## Others

- JavaScript: dynamic client side language to make web pages more interactive
- We might also want to use a database to store results, and learn a bit about other web query methods such as XPath

# Cascading Style Sheets

- Common HTML tags which help to select elements:
  - “id”, which is used to attach a page-unique identifier to a certain tag
  - “class”, which lists a space-separated series of CSS class names

# Cascading Style Sheets

- Originally, HTML was meant as a way to define both the structure and formatting of a website
  - In the early days of the web, it was normal to find lots of HTML tags which were meant to define how content should look like, e.g. “...” for bold text , “...” for italics text, and so on
  - Web developers began to argue that the structure and formatting of documents basically relate to two different concerns
  - CSS to govern how a document should be styled, HTML governs how it should be structured

# Cascading Style Sheets

- In CSS, style information is written down as a list of colon-separated key-value based statements, with each statement itself being separated by a semicolon, as follows:

```
color: 'red';
background-color: #ccc;
font-size: 14pt;
border: 2px solid yellow;
```

# Cascading Style Sheets

- These style declarations can be included in a document in three different ways:
  - Inside a regular HTML “style” attribute, for instance as in:
    <p style=”color:'red';”>…</p>
  - Inside of HTML “<style>…</style>” tags , placed in inside the “<head>” tag of a page
  - Inside a separate file, which is then referred to by means of a “<link>” tag inside the “<head>” tag of a page. This is the most clean way of working. When loading a web page, your browser will perform an additional HTTP request to download this CSS file and apply its defined styles to the document

  - How to determine to which elements the styling should be applied?

# Cascading Style Sheets

```
h1 {
  color: red;
}

div.box {
  border: 1px solid black;
}

#intro-paragraph {
  font-weight: bold;
}
```

# Cascading Style Sheets

- **tagname** selects all elements with a particular tag name
- **.classname** (note the dot) selects all elements having a particular class defined in the HTML document (class attribute)
- **#idname** matches elements based on their “id” attribute
- These selectors can be combined in all sorts of ways. div.box for instance selects all “ tags, but not “ ” tags
- Multiple selector rules can be specified by using a comma, “,”, e.g. h1, h2, h3

- **selector1 selector2** defines a chaining rule (note the space) and selects all elements matching selector2 inside of elements matching selector1
- **selector1 > selector2** selects all elements matching selector2 where the direct parent element matches selector1
- **selector1 + selector2** selects all elements matching selector2 that are placed directly after (on the same level in the HTML hierarchy) elements matching selector1
- **selector1 ~ selector2** selects all elements matching selector2 that are placed after (on the same level in the HTML hierarchy) elements matching selector1

# Cascading Style Sheets

- **tagname[attributename]** selects all tagname elements where an attribute named attributename is present
- **[attributename=value]** checks the actual value of an attribute as well. If you want to include spaces, wrap the value in double quotes
- **[attributename~=value]** does something similar, but instead of performing an exact value comparison, here all elements are selected whose attributename attribute's value is a space-separated list of words, one of them being equal to value
- **[attributename|=value]** selects all elements whose attributename attribute's value is a space-separated list of words, with any of them being equal to “value” or starting with “value” and followed by a hypen (“-”)
- **[attributename^=value]** selects all element whose attribute value starts with the provided value
- **[attributename$=value]** selects all elements whose attribute value ends with the provided value
- **[attributename*=value]** selects all elements whose attribute value contains the provided value

- Finally, there are a number of “colon” and “double-colon” “pseudo-classes” that can be used in a selector rule as well. **p:first-child** selects every “<p>” tag that is the first child of its parent element, **p:last-child** and **p:nth-child(10)** provide similar functionality, and there’s also **:not()**, and some others (less used)

# Using BeautifulSoup

- Using the select method

```
# Find all <a> tags
html_soup.select('a')

# Find the element with the info id
html_soup.select('#info’)

# Find <div> tags with both classa and classb CSS classes
html_soup.select(div.classa.classb)
```

```
# Find <a> tags with an href attribute starting with
# http://example.com/
html_soup.select('a[href^="http://example.com/"]’)

# Find <li> tags which are children of <ul> tags
# with class lst
html_soup.select(ul.lst > li')
```

# Using BeautifulSoup

- However, the CSS selector rule engine in Beautiful Soup is not as powerful as the one found in a modern web browser
  - Some complex selectors might not work
    - Doesn't pose that much of an issue in most use cases
  - Use pyquery (http://pyquery.readthedocs.io/en/latest/), parsel (https://parsel.readthedocs.io/en/latest/) or Selenium (later)

# Using BeautifulSoup

- An example using select

```
import requests
from bs4 import BeautifulSoup
import re

url = 'https://www.zalando.be/herenschoenen-sneakers/'

r = requests.get(url)
html_contents = r.text

html_soup = BeautifulSoup(html_contents, 'html.parser')

for item in html_soup.select('.cat_articles > z-grid-item'):
  article = item.find('', class_=re.compile('^cat_articleName'))
  if not article: continue
  price = item.find('', class_=re.compile('^cat_originalPrice'))
  link = item.find('a', class_=re.compile('^cat_imageLink'))
  image = link.find('img')
  print(article.text, price.text)
  print(link['href'])
  print(image['src'])
```

# More on HTTP

- Forms and POST data
- Headers
- Cookies

# Forms and POST data

# Forms and POST data


<form method="get"> [...] </form>

<form method="post"> [...] </form>

# Forms and POST data

- In the case of a “get” method, the form data just get submitted as part of the GET request as URL parameters
  - Easy to handle with requests
  - E.g. https://www.zalando.be/adidas-shop-heren/**?sc=false&_q=adidas**

# Forms and POST data

- When a post method is used, the browser will perform an HTTP POST request
  - Used in cases where a request is not idempotent: the browser will warn you if you try to refresh a POST request

- POST /login HTTP/1.1 — **Request method, url, version**
  Host: www.example.com — **Request headers (host is mandatory)**
  Accept: image/gif, image/jpeg, */* — …
  Accept-Language: en-us — …
  Accept-Encoding: gzip, deflate — …
  User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
  **<blank line>**
  **name=Seppe&age=34** — **Message body contains POST data**

# POST requests with requests

- A POST request with requests

```
import requests

url = 'http://www.webscrapingfordatascience.com/postform2/'
r = requests.get(url)

formdata = {
    'name': 'Seppe',
    'gender': 'M',
    'pizza': 'like',
    'haircolor': 'brown',
    'comments': ''
    }

r = requests.post(url, data=formdata)
print(r.text)
```

# POST requests with requests

- Note that URL parameters can still be provided with a POST request

- Both parameters and POST data can contain duplicate keys
  - So requests also allows to pass in a list of (name, value) tuples for these rarer cases
  - OrderedDict also possible

# POST requests with requests

- It’s here where we can start seeing the first “picky” aspects of web servers
  - Sometimes servers check duplicate keys
  - The submit button in a web form can have a name and be included in parameters or POST data
  - The ordering of parameters or POST data fields can matter
  - Sometimes, a state or CSRF field needs to be re-included with every subsequent POST request (ASP.NET is especially noteworthy for this)
  - Sometimes, servers do not care whether data is in the URL parameter or POST data

# POST requests with requests

- A more involved POST request with requests

```
import requests
from bs4 import BeautifulSoup

url = 'http://www.webscrapingfordatascience.com/postform3/'

# First perform a GET request
r = requests.get(url)

# Get out the value for protection
html_soup = BeautifulSoup(r.text, 'html.parser')
p_val = html_soup.find('input', attrs={'name': 'protection'}).get('value')

# Then use it in a POST request
formdata = { 'name': 'Seppe', 'gender': 'M', 'pizza': 'like', 'haircolor': 'brown', 'comments': '', 'protection': p_val }

r = requests.post(url, data=formdata)
print(r.text)
```

# Other HTTP methods

- **GET**: requests a representation of the specified URL. Requests using GET should only retrieve data and should have no other effect, such as saving or changing user information or perform other actions. GET requests can—technically—include an optional request body as well, but this is not recommended by the HTTP standard. As such, web browser don't include anything in the request body when perform GET requests, and it is not used by most (if not all) APIs either
- **POST**: the POST method indicates that data is being submitted as part of a request to a particular URL, e.g. a forum message, a file upload, a filled in form, and so on. Contrary to GET, POST requests are not expected to be idempotent, meaning that submitting a POST request can bring about changes on the web server's end of things. POST request encode the submitted data as part of the request body
- **HEAD**: the HEAD method requests a response just like the GET request does, but indicates to the web server that it does not need to send the response body. HEAD requests cannot have a request body
- **PUT**: the PUT method requests that the submitted data should be stored under the supplied request URL, thereby creating it if it does not exist already. Just as with a POST, PUT requests have a request body
- **DELETE**: the DELETE method requests that the data listed under the request URL should be removed. The DELETE request does not have a request body
- **CONNECT** , **OPTIONS** , **TRACE** and **PATCH**: less-commonly encountered request methods. CONNECT is generally used to request a web server to set up a direct TCP network connection between the client and the destination (web proxy servers will use this type of request), TRACE instructs the web server to just send the request back to the client (used for debugging to see if a middleman in the connection has changed your request somewhere in-between), OPTIONS requests the web server to list the HTTP methods it accepts for a particular URL (which might seem helpful, though is rarely used). PATCH finally allows to request a partial modification of a specific resource

- **GET and POST are most used (especially by web browsers), though REST APIs might sometimes require others**

# HTTP headers

```
import requests


url = 'http://www.webscrapingfordatascience.com/usercheck/'
r = requests.get(url)


print(r.text)
# Shows: It seems you are using a scraper


print(r.request.headers)


# { 'User-Agent': 'python-requests/2.18.4',
#   'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive' }
```

# HTTP headers

- Setting headers in requests

```
import requests

url = 'http://www.webscrapingfordatascience.com/usercheck/'

my_headers = {
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 ' +
                '(KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36'
}

r = requests.get(url, headers=my_headers)

print(r.text)
print(r.request.headers)
```

# HTTP headers

- Apart from the “User-Agent” header, there is another header that deserves special mention: the “Referer” header (originally a misspelling of referrer and kept that way since then)
  - Browsers will include this header to indicate the URL of the web page that linked to the URL being requested
  - Some websites will check this to prevent “deep links” (or “hotlinks”) from working

# HTTP headers

- Just as we've seen at various occasions before, remember that web servers can get very picky in terms of headers that are being sent as well
  - Rare edge cases such as the order of headers, multiple header lines with the same header name, or custom headers being included in requests can all occur in real-life situations
  - If you see that requests is not returning the results you expect and have observed when using the site in your browser, inspect the headers through your browser’s developer tools to see exactly what is going on and duplicate it as well as possible in Python

# HTTP headers

- Just like the data and params arguments, headers can accept an OrderedDict object in case the ordering of the headers is important
  - Passing a list, however, is not permitted here, as the HTTP standard does not allow multiple request header lines bearing the same name
- What is allowed is to provide multiple values for the same header by separating them with a comma, as in the line “Accept-Encoding: gzip, deflate”
  - In that case, you can just pass the value as is with requests
  - However, that's not to say that some extremely weird websites or APIs might still use a setup where it deviates from the standard and checks for the same headers on multiple lines in the request (very exceptional)
  - In that case, you'll have no choice but to implement a hack to extend requests
- Note that response headers can contain multiple lines with the same name
  - Requests will automatically join them using a comma and put them under one entry when you access r.headers

# Cookies

- HTTP is a simple networking protocol
  - Text based and follows a simple request-and-reply based communication scheme (HTTP 1.1)
  - In the simplest case, every request-reply cycle in HTTP involves setting up a fresh new underlying network connection as well, though the 1.1 version of the HTTP standard allows to set up “keep alive” connections
- This simple request-reply based approach poses some problems for website
  - From a web server's point of view, every incoming request is completely independent of any previous ones and can be handled on its own
  - This is not, however, what users expect from most websites
  - For instance an online shop where items can be added to a cart. When visiting the checkout page, we expect the web server to “remember” the items we selected and added previously
- How to add a state mechanism to HTTP?
  - We could include a special identifier as a URL parameter that “links” multiple visits to the same user, e.g. “checkout.html?visitor=20495”, but easy to leak? What if browser is reopened from start?
  - For POST requests, we could either use the same URL parameter, or include the “session” identifier in a hidden form field, but would make everything a POST request?
  - IP address based? But this might be shared
  - Some older websites indeed use such mechanisms

# Cookies

- A better mechanism: cookies
  - Simple header mechanism: server sends cookies the browser should resend with every subsequent request to that domain

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: sessionToken=20495; Expires=Wed, 09 Jun 2021 10:10:10 GMT
Set-Cookie: siteTheme=dark
[...]
```

  - Or all in one line and comma-separated (though less common given that comma can appear in cookie line)
  - (Some servers use “set-cookie” in lowercase – allowed for header names)

# Cookies

- Every cookie is well-defined:
  - A name and value, separated with “=“
  - Additional attributes, “;” separated: Expires, Max-Age, Domain, Path, Secure, HttpOnly

- At every subsequent request, browser checks and sends its cookies (here also semicolon separated)

```
GET /anotherpage.html HTTP/1.1
Host: www.example.com
Cookie: sessionToken=20495; siteTheme=dark
[...]
```

# Cookies

- Note: advertisers have come up with many different ways to perform fingerprinting:
  - JSON Web Tokens, IP addresses, ETag headers, web storage, Flash and many other approaches
  - See e.g. “evercookie”
  - Luckily, oftentimes not necessary to “emulate” all this for web scraping

# Cookies

- Setting cookies manually in requests

```
import requests

url = 'http://www.webscrapingfordatascience.com/cookielogin/secret.php'

my_cookies = {'PHPSESSID': 'ijfatbjege43lnsfn2b5c37706'}

r = requests.get(url, cookies=my_cookies)

print(r.text)
# Shows: This is a secret code: 1234
```

# Cookies

- Setting cookies manually in requests: login process

```
import requests

url = 'http://www.webscrapingfordatascience.com/cookielogin/'

# First perform a POST request
r = requests.post(url, data={'username': 'dummy', 'password': '1234'})

# Get the cookie value, either from  r.headers or r.cookies
print(r.cookies)
my_cookies = r.cookies

# r.cookies is a RequestsCookieJar object which can also be accessed like a dictionary. The following also works:
my_cookies['PHPSESSID'] = r.cookies.get('PHPSESSID')

# Now perform a GET request to the secret page using the cookies
r = requests.get(url + 'secret.php', cookies=my_cookies)

print(r.text)
# Shows: This is a secret code: 1234
```

# Cookies

- Setting cookies manually in requests: login with redirect

```
import requests

url = 'http://www.webscrapingfordatascience.com/redirlogin/'

# First perform a POST request -- do not follow the redirect
r = requests.post(url, data={'username': 'dummy', 'password': '1234'}, allow_redirects=False)

print(r.cookies)

my_cookies = r.cookies

# Now perform a GET request manually to the secret page using the cookies
r = requests.get(url + 'secret.php', cookies=my_cookies)

print(r.text)
# Shows: This is a secret code: 1234
```

# Sessions in requests

- A better approach: sessions

```
import requests

url = 'http://www.webscrapingfordatascience.com/trickylogin/'

my_session = requests.Session()

r = my_session.post(url)
r = my_session.post(url, params={'p': 'login'},
                    data={'username': 'dummy', 'password': '1234'})
r = my_session.get(url, params={'p': 'protected'})

print(r.text)
# Shows: Here is your secret code: 3838.
```

# Sessions in requests

- Changing a header across the whole session

```
import requests

url = 'http://www.webscrapingfordatascience.com/trickylogin/'

my_session = requests.Session()
my_session.headers.update({'User-Agent': 'Chrome!'})

# All requests in this session will now use this User-Agent header

r = my_session.post(url)
print(r.request.headers)

r = my_session.post(url, params={'p': 'login'}, data={'username': 'dummy', 'password': '1234'})
print(r.request.headers)

r = my_session.get(url, params={'p': 'protected'})
print(r.request.headers)
```

# Sessions in requests

- Using a Session object is generally the best way to use requests
  - Headers can be set globally
  - Cookies managed automatically
  - Can be cleared using my_session.cookies.clear()
    - Since the cookiejar is compatible with standard Python dictionary

# Other content: binary files

```
import requests


url = 'http://www.webscrapingfordatascience.com/files/kitten.jpg'


r = requests.get(url)


with open('image.jpg', 'wb') as my_file:
    my_file.write(r.content)
```

# Other content: binary files streaming

```
import requests


url = 'http://www.webscrapingfordatascience.com/files/kitten.jpg'


r = requests.get(url, stream=True)
# You can now use r.raw
# r.iter_lines
# and r.iter_content


with open('image.jpg', 'wb') as my_file:
    # Read by 4KB chunks
    for byte_chunk in r.iter_content(chunk_size=4096):
        my_file.write(byte_chunk)
```

# Other content: JSON

```
import requests


url = 'http://www.webscrapingfordatascience.com/jsonajax/results.php'


r = requests.post(url, data={'api_code': 'C123456'})


print(r.json())
print(r.json().get('results'))
```

# Other content: JSON POST

```
import requests

url = 'http://www.webscrapingfordatascience.com/jsonajax/results2.php'

# Use the json argument to encode the data as JSON:
# Some APIs and sites will use an “application/json” “Content-Type” header for formatting
# the request and submit the POST data as plain JSON
r = requests.post(url, json={'api_code': 'C123456'})

# Note the Content-Type header in the request:
print(r.request.headers)

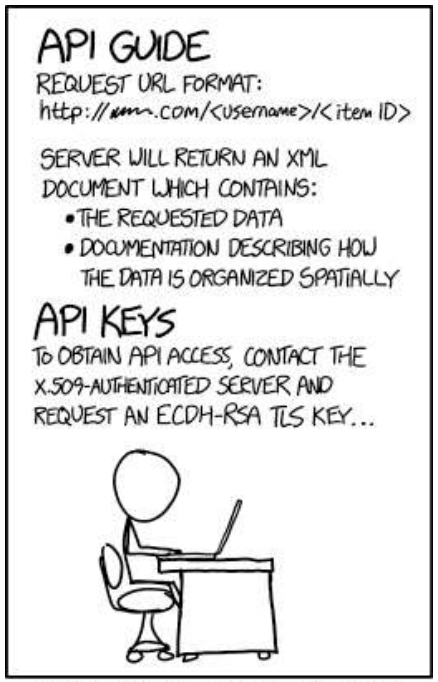
print(r.json())
```

# Other content

- Even if the website you wish to scrape does not provide an API, it's always recommended to keep an eye on your browser's developer tools networking information to see if you can spot JavaScript-driven requests to URL endpoints which return nicely structured JSON data

- Even although an API might not be documented, fetching the information directly from such “internal APIs” is always a good idea, as this will avoid having to deal with the HTML soup

# Other content




https://xkcd.com/1481/

# The three core pillars of the web

## HTTP

- Protocol used to communicate with web servers
- We'll need something that "speaks" HTTP

## HTML

- The format used to send web pages
- Semi-structured plain text format
- We'll need something that can parse this
- Other formats possible as well: e.g. downloading image, PDF, video files

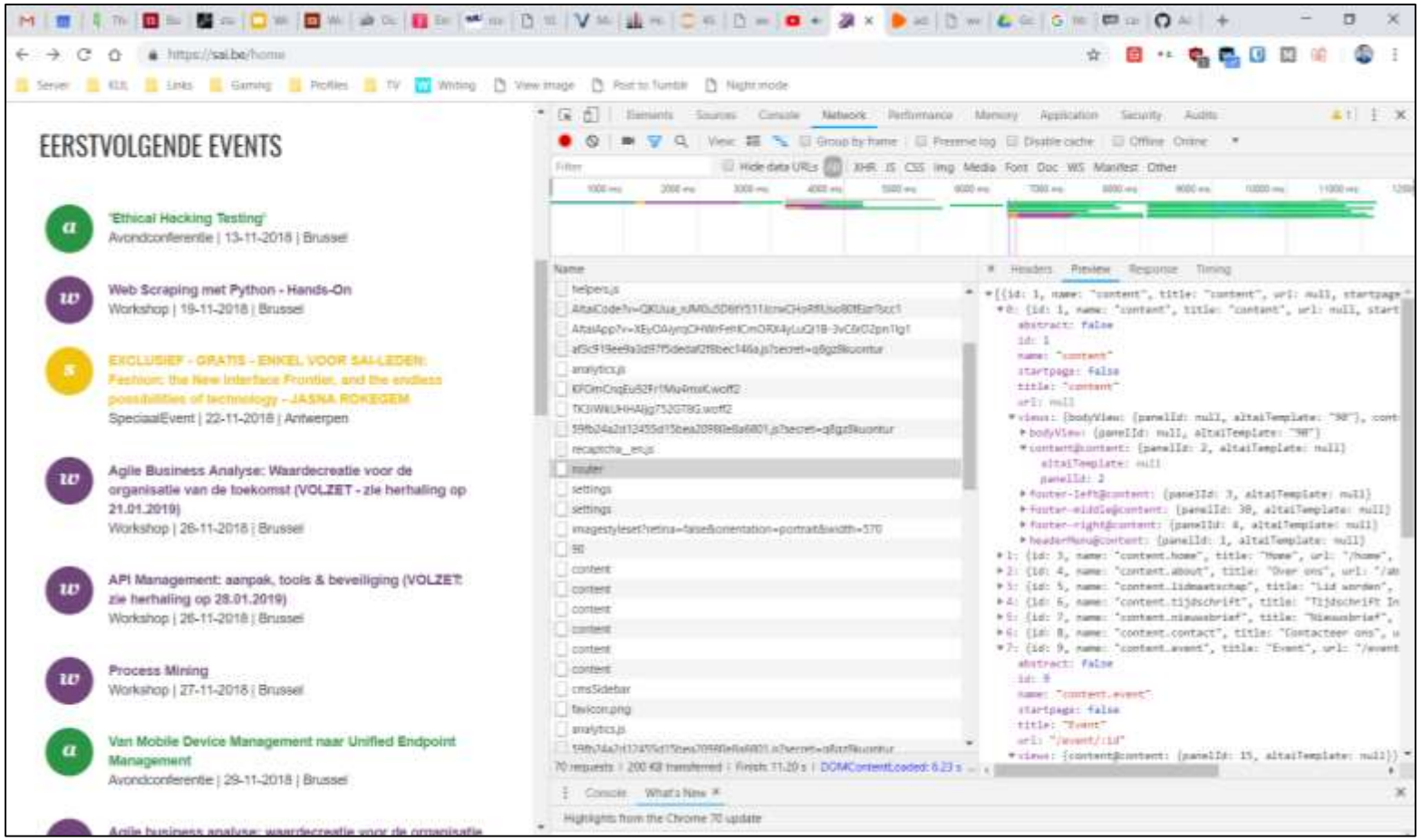
## CSS

- Used in combination with HTML to style web pages
- We'll borrow the concept of CSS style selectors to quickly identify elements on a web page

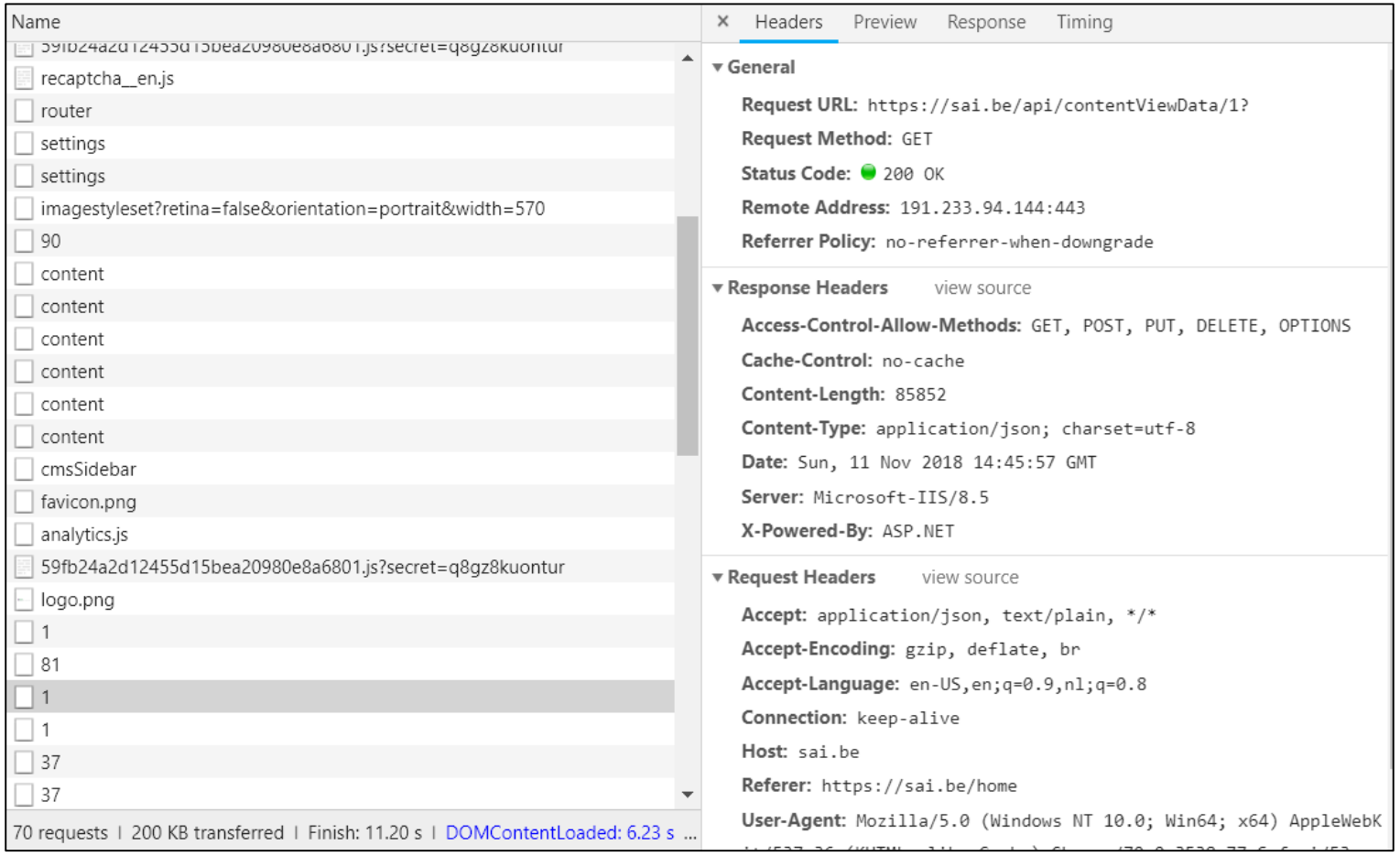
## Others

- JavaScript: dynamic client side language to make web pages more interactive
- We might also want to use a database to store results, and learn a bit about other web query methods such as XPath

# Speaking of JavaScript



# Speaking of JavaScript



# Not even an HTML parser required

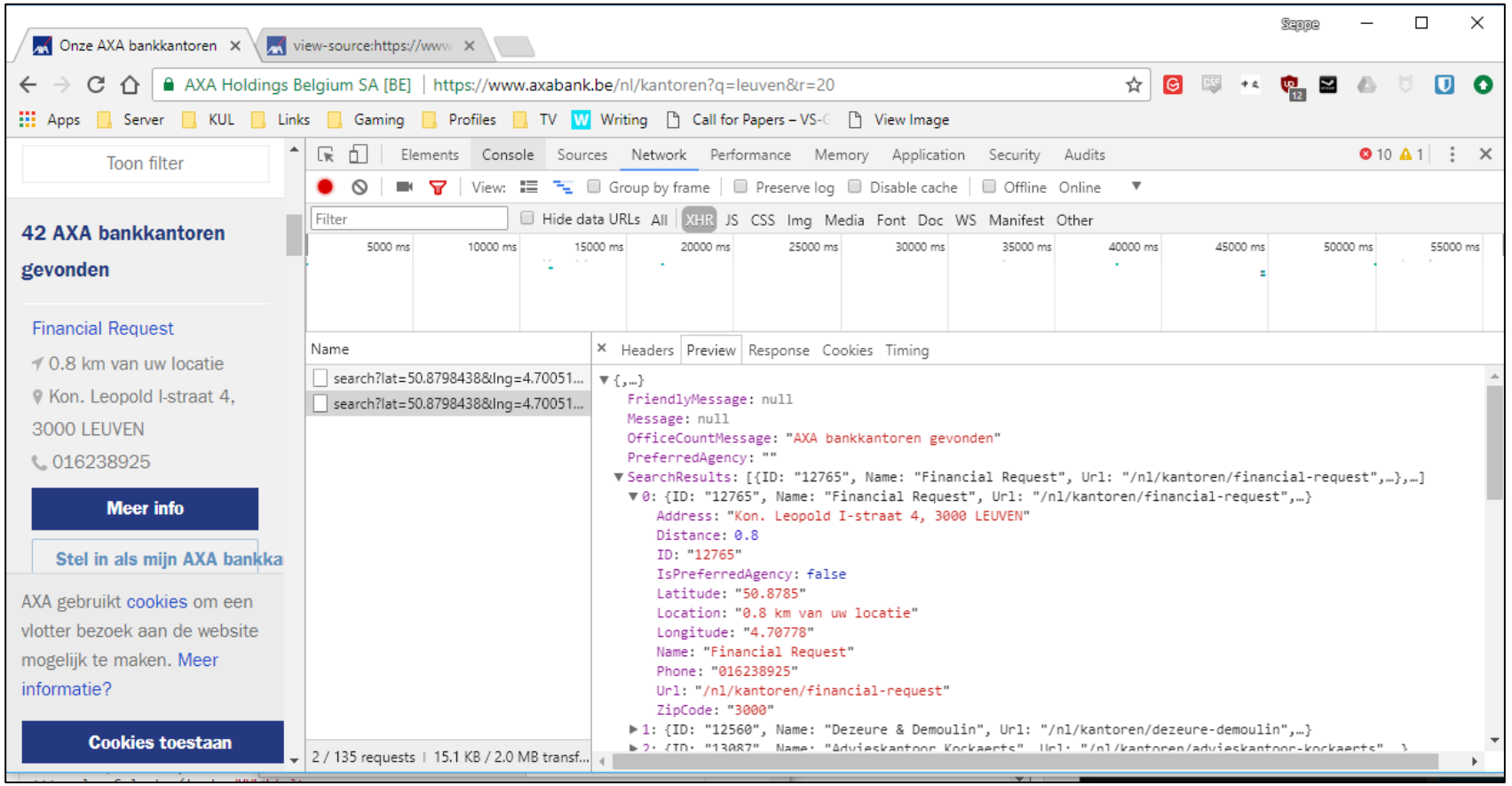
```
import requests

url = 'https://sai.be/api/contentViewData/1?'

r = requests.get(url)
content = r.json()

for piece in content:
  print(piece['name'], piece['datum'], piece['contentType'])
```

# Same here…


Onze AXA bankkantoren
AXA Holdings Belgium SA [BE] | https://www.axabank.be/nl/kantoren?q=leuven&r=20
Toon filter
42 AXA bankkantoren gevonden
Financial Request
0.8 km van uw locatie
Kon. Leopold I-straat 4,
3000 LEUVEN
016238925
Meer info
Cookies toestaan
Elements Console Sources Network Performance Memory Application Security Audits
Name
search?lat=50.8798438&lng=4.70051...
Headers Preview Response Cookies Timing
FriendlyMessage: null
Message: null
OfficeCountMessage: "AXA bankkantoren gevonden"
PreferredAgency: ""
SearchResults: [{ID: "12765", Name: "Financial Request", Url: "/nl/kantoren/financial-request",…},…]
Address: "Kon. Leopold I-straat 4, 3000 LEUVEN"
Distance: 0.8
ID: "12765"
IsPreferredAgency: false
Latitude: "50.8785"
Location: "0.8 km van uw locatie"
Longitude: "4.70778"
Name: "Financial Request"
Phone: "016238925"
Url: "/nl/kantoren/financial-request"
ZipCode: "3000"

# It’s not always that easy

- For more complex sites, JavaScript can end up doing a lot more
  - Single-page apps
  - Dynamically changed and added elements
  - Setting cookies
  - Performing browser checks

- requests / BeautifulSoup does not come with a JavaScript engine
  - For those JavaScript-heavy web sites, will have to emulate a full browser

# Selenium


- Selenium: a browser instrumentation / automation framework
  - https://www.seleniumhq.org/
  - For other languages too
  - Depends on a WebDriver: a browser (Firefox, Chrome, …)
  - Most close to a real browser, but comes with overhead
  - Also a bit harder to grok: no longer: perform HTTP requests and parse HTML, but “click buttons”, “find text”, “wait for this to load”, “inject this JavaScript”

# Selenium

```
from selenium import webdriver
from selenium.webdriver.support.ui import Select

url = 'http://www.iata.org/publications/Pages/code-search.aspx'

driver = webdriver.Chrome()
driver.implicitly_wait(10)

def get_results(airline_name):
    driver.get(url)
    # Make sure to select the right part of the form
    # This will make finding the elements easier as #aspnetForm wraps the whole page, including the search box
    form_div = driver.find_element_by_css_selector('#aspnetForm .iataStandardForm')
    select = Select(form_div.find_element_by_css_selector('select'))
    select.select_by_value('ByAirlineName')
    text = form_div.find_element_by_css_selector('input[type=text]')
    text.send_keys(airline_name)
    submit = form_div.find_element_by_css_selector('input[type=submit]')
    submit.click()
    table = driver.find_element_by_css_selector('table.datatable')
    table_html = table.get_attribute('outerHTML')
    return table_html

tbl = get_results('Lufthansa')
print(tbl)

driver.quit()
```

# Selenium

- Selection methods
  - find_element_by_id
  - find_element_by_name
  - find_element_by_xpath
  - find_element_by_link_text
  - find_element_by_partial_link_text
  - find_element_by_tag_name
  - find_element_by_class_name
  - find_element_by_css_selector

# Selenium

- Implicit waits and explicit wait conditions
- Actions, keypresses, form elements (select dropdowns), mouse movement
- JavaScript injection

# Selenium

```
from selenium import webdriver
from selenium.webdriver.support.select import Select
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.action_chains import ActionChains

url = 'http://www.webscrapingfordatascience.com/postform2/'

driver = webdriver.Chrome()
driver.implicitly_wait(10)

driver.get(url)

chain = ActionChains(driver)
chain.send_keys_to_element(driver.find_element_by_name('name'), 'Seppe')
chain.click(driver.find_element_by_css_selector('input[name="gender"][value="M"]'))
chain.click(driver.find_element_by_name('pizza'))
chain.click(driver.find_element_by_name('salad'))
chain.click(driver.find_element_by_name('comments'))
chain.send_keys(['This is a first line', Keys.ENTER, 'And this a second'])
chain.perform()

Select(driver.find_element_by_name('haircolor')).select_by_value('brown')

input('Press ENTER to submit the form')

driver.find_element_by_tag_name('form').submit()
# Or: driver.find_element_by_css_selector('input[type="submit"]').click()

input('Press ENTER to close the automated browser')
driver.quit()
```

# Selenium

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys

class at_least_n_elements_found(object):
    def __init__(self, locator, n):
        self.locator = locator
        self.n = n
    def __call__(self, driver):
        elements = driver.find_elements(*self.locator)
        if len(elements) >= self.n:
            return elements
        else:
            return False

url = 'http://www.webscrapingfordatascience.com/complexjavascript/'

driver = webdriver.Chrome()
driver.get(url)
driver.implicitly_wait(10)

div_element = driver.find_element_by_class_name('infinite-scroll')
quotes_locator = (By.CSS_SELECTOR, ".quote:not(.decode)")
```

```
nr_quotes = 0
while True:
    action_chain = ActionChains(driver)
    action_chain.move_to_element(div_element)
    action_chain.click()
    action_chain.send_keys([Keys.PAGE_DOWN for i in range(10)])
    action_chain.perform()

    try:
        all_quotes = WebDriverWait(driver, 3).until(
            at_least_n_elements_found(quotes_locator, nr_quotes + 1) )
    except TimeoutException as ex:
        # No new quotes within 3 seconds, assume this is all there is
        print("... done!")
        break

    # Otherwise, update the quote counter
    nr_quotes = len(all_quotes)
    print("... now seeing", nr_quotes, "quotes")

# all_quotes will contain all the quote elements
print(len(all_quotes), 'quotes found\n')
for quote in all_quotes:
    print(quote.text)

driver.quit()
```

# Selenium

```
result = browser.execute_script('''
  var script = document.createElement('script');
  script.src = 'https://cdn.firebase.com/js/client/2.2.7/firebase.js';
  document.head.appendChild(script);''')


result = browser.execute_script('''
  var script = document.createElement('script');
  script.src='https://www.gstatic.com/firebasejs/5.4.1/firebase.js';
  document.head.appendChild(script);''‘’)


result = browser.execute_script('''
    firebaseConfig = {
      apiKey: 'AIzaSyCzDDU6UV_V-oRK3JSUHLAHG5ZvX2JLn9g',
      authDomain: 'kies18-iedereenkiest.firebaseapp.com',
      databaseURL: 'https://kies18-iedereenkiest.firebaseio.com',
      projectId: 'kies18-iedereenkiest',
      storageBucket: 'kies18-iedereenkiest.appspot.com',
      messagingSenderId: '647166621498'
    };
    firebaseApp = firebase.initializeApp(firebaseConfig);
    firebaseDatabase = firebaseApp.database();
    ''')
```

```
def get_keys(name):
    result = browser.execute_async_script('''
        var done = arguments[0];
        var result = firebaseDatabase.ref(
            "''' + name + '''").once("value")
            .then(function(snapshot) {
                done(Object.keys(snapshot.val()));
        });
        ''')
    return result

def get_ref(name):
    result = browser.execute_async_script('''
        var done = arguments[0];
        var result = firebaseDatabase.ref(
            "''' + name + '''").once("value")
            .then(function(snapshot) {
                done(snapshot.val());
        });
        ''')
    return result

def get_user_info(key):
    user = {'verified': None, 'name': None, 'party': None}
    user['verified'] = get_ref('prod/users/{}/isVerified'.format(key))
    user['name'] = get_ref('prod/users/{}/name'.format(key))
    user['party'] = get_ref('prod/users/{}/party'.format(key))
    return user
```

# Other libraries

- Some other noteworthy Python libraries and tools to know about...

# HTTP in Python

- HTTP libraries
  - Python 3 comes with a built-in module called “urllib” which can deal with all things HTTP (see https://docs.python.org/3/library/urllib.html)
    - The module got heavily revised compared to its counterpart in Python 2, where HTTP functionality was split up in both “urllib” and “urllib2” and somewhat cumbersome to work with
  - “httplib2” (see https://github.com/httplib2/httplib2): a small, fast HTTP client library. Originally developed by Googler Joe Gregorio, and now community supported
  - “urllib3” (see https://urllib3.readthedocs.io/): a powerful HTTP client for Python, used by the requests library below
  - “requests” (see http://docs.python-requests.org/): an elegant and simple HTTP library for Python, built “for human beings”
  - “grequests” (see https://pypi.python.org/pypi/grequests), which extends requests to deal with asynchronous, concurrent HTTP requests
  - “aiohttp” (see http://aiohttp.readthedocs.io/): another library focusing on asynchronous HTTP
  - “hyper” (see https://hyper.readthedocs.io/en/latest/): one of the few libraries offering HTTP 2.0 support. Harder to use

# Parsing HTML

- Parsing libraries
  - parse (https://pypi.python.org/pypi/parse): “the opposite of format()”:
    - `search('Age: {:d}\n', 'Name: Rufus\nAge: 42\nColor: red\n')`
    - For textual searches
  - pyquery (http://pyquery.readthedocs.io/en/latest/): “searching using jQuery style selectors”
    - Better CSS selector support
  - parsel (https://parsel.readthedocs.io/en/latest/): “search using XPath and CSS selectors”
    - Better CSS selector and Xpath support
  - Cleaner libraries (Cleaner in lxml, htmllaundry, html2text, …)
  - These can be combined with BeautifulSoup to refine found information

# Parsing HTML

- If you don't want to use Beautiful Soup, keep in mind that the Beautiful Soup library itself depends on an HTML parser to perform most of the bulk parsing work
  - It is hence also possible to use these lower level parsers directly should you wish to do so
  - The “html.parser” module in Python provides such a parser which we've been using already as the “engine” for Beautiful Soup, but can be used directly as well, with “lxml” and “html5lib” being popular alternatives
  - Some people prefer this approach, as it can be argued that the additional overhead added by Beautiful Soup causes some slowdown
    - This is true, though we find that in most uses, you'll first have to deal with other issues before scraping speed becomes a real concern, like e.g. setting up a parallel scraping mechanism

# Alternative scraper toolkits

- MechanicalSoup (http://mechanicalsoup.readthedocs.io/en/stable/): a replacement for the older Mechanize
  - Less used nowadays
  - No JavaScript support

- Scrapy (https://scrapy.org/): solid package to write scrapers and crawlers (see later)
  - Strong logging facilities, well-supported deployment using Scrapy Cloud (commercial)
  - JavaScript supported through “Splash”, though somewhat limited

# Caching

- CacheControl
  - http://cachecontrol.readthedocs.io/en/latest/

```
import requests
from cachecontrol import CacheControl
session = requests.Session()
cached_session = CacheControl(session)
# You can now use cached_session like a normal session
# All GET requests will be cached
```

- Or through a local HTTP proxy server (like Fiddler or Squid)

# Smart retries

- https://www.peterbe.com/plog/best-practice-with-retries-with-requests

```
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

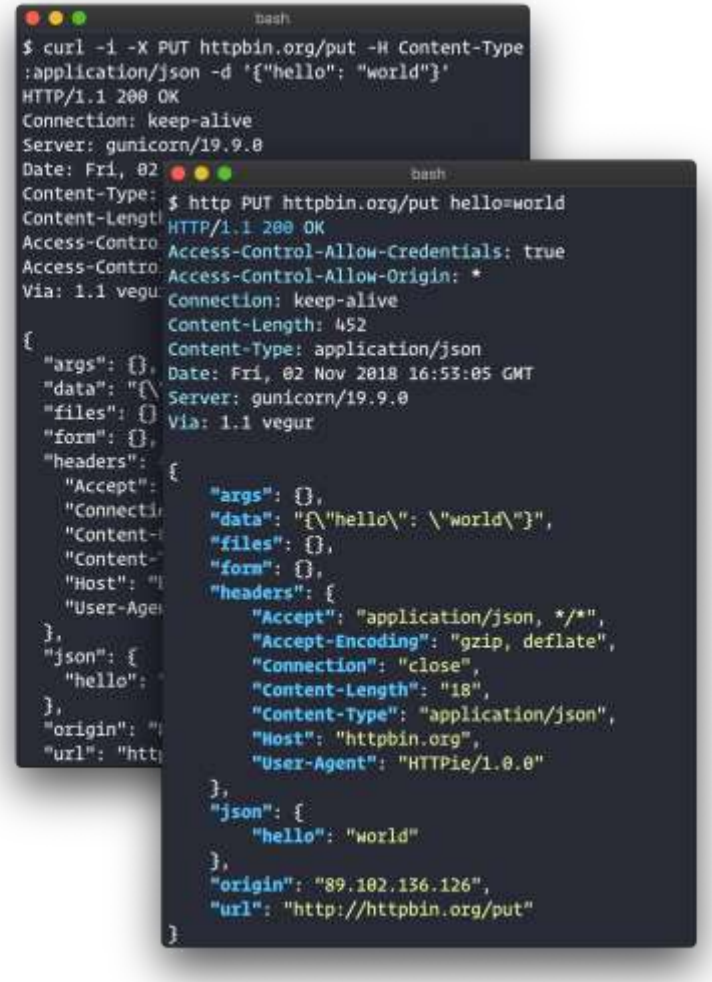
def requests_retry_session(retries=3, backoff_factor=0.3, status_forcelist=(500, 502, 504), session=None):
    session = session or requests.Session()
    retry = Retry(
        total=retries,
        read=retries,
        connect=retries,
        backoff_factor=backoff_factor,
        status_forcelist=status_forcelist,
    )
    adapter = HTTPAdapter(max_retries=retry)
    session.mount('http://', adapter)
    session.mount('https://', adapter)
    return session
```

# News articles

- Basically boils down to getting out the main content from a page
  - Much trickier as it might seem at first sight
  - You might try to iterate all the lowest-level HTML elements and keeping the one with the most text embedded in it, though this approach will break if the text in an article is split up over multiple sibling elements, like a series of “<p>” tags inside a larger “ ”
  - Considering all elements does not resolve this issue, as you'll end up by simple selecting the top element (e.g. “<html>” or “<body>”) on the page, as this will always contain the largest amount (i.e. all) text
  - The same holds in case you'd rely on the rect attribute Selenium provides to apply a visual approach (i.e. find the element taking up most space on the page)
- A large number of libraries and tools have been written to solve this issue
  - https://github.com/masukomi/ar90-readability
  - https://github.com/misja/python-boilerpipe
  - https://github.com/codelucas/newspaper
  - https://github.com/fhamborg/news-please
  - Specialized APIs: https://newsapi.org/ or https://webhose.io/news-api
  - https://github.com/mozilla/readability: a Mozilla JavaScript library, but possible to use it with Python and Selenium
- Alternative: RSS (Rich Site Summary) : a web feed which allows users to access updates to online content in a standardized, XML-based format. Keep an eye out for “<link>” tags with their “type” attribute set to "application/rss+xml". The “href” attribute will then announce the URL where the RSS feed can be found
  - Sadly less used in recent years

# Command line tools

- Nice as helpers
  - https://httpie.org/
  - https://curl.haxx.se/ and https://www.gnu.org/software/wget/

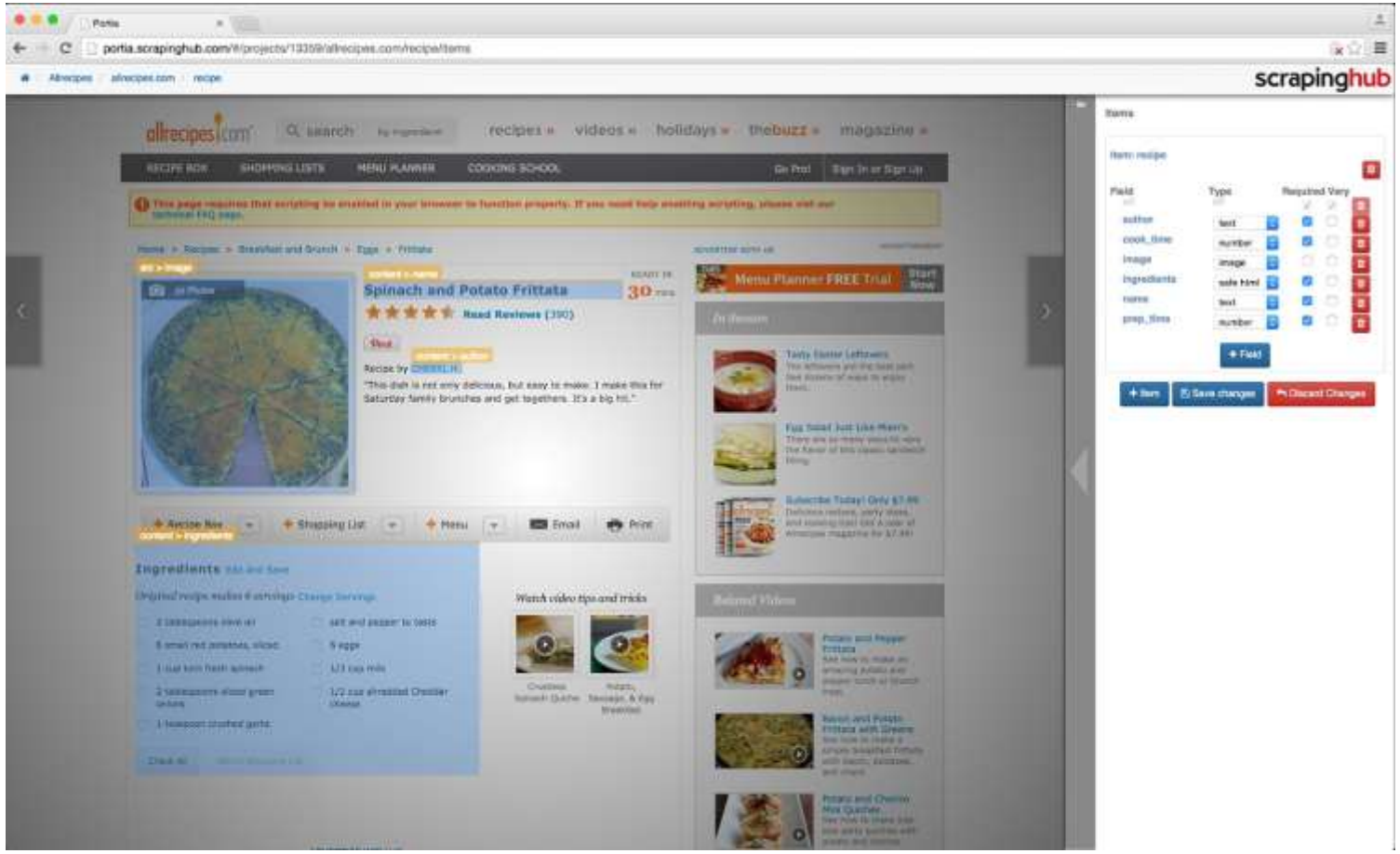


# Other Tools and Cross-overs

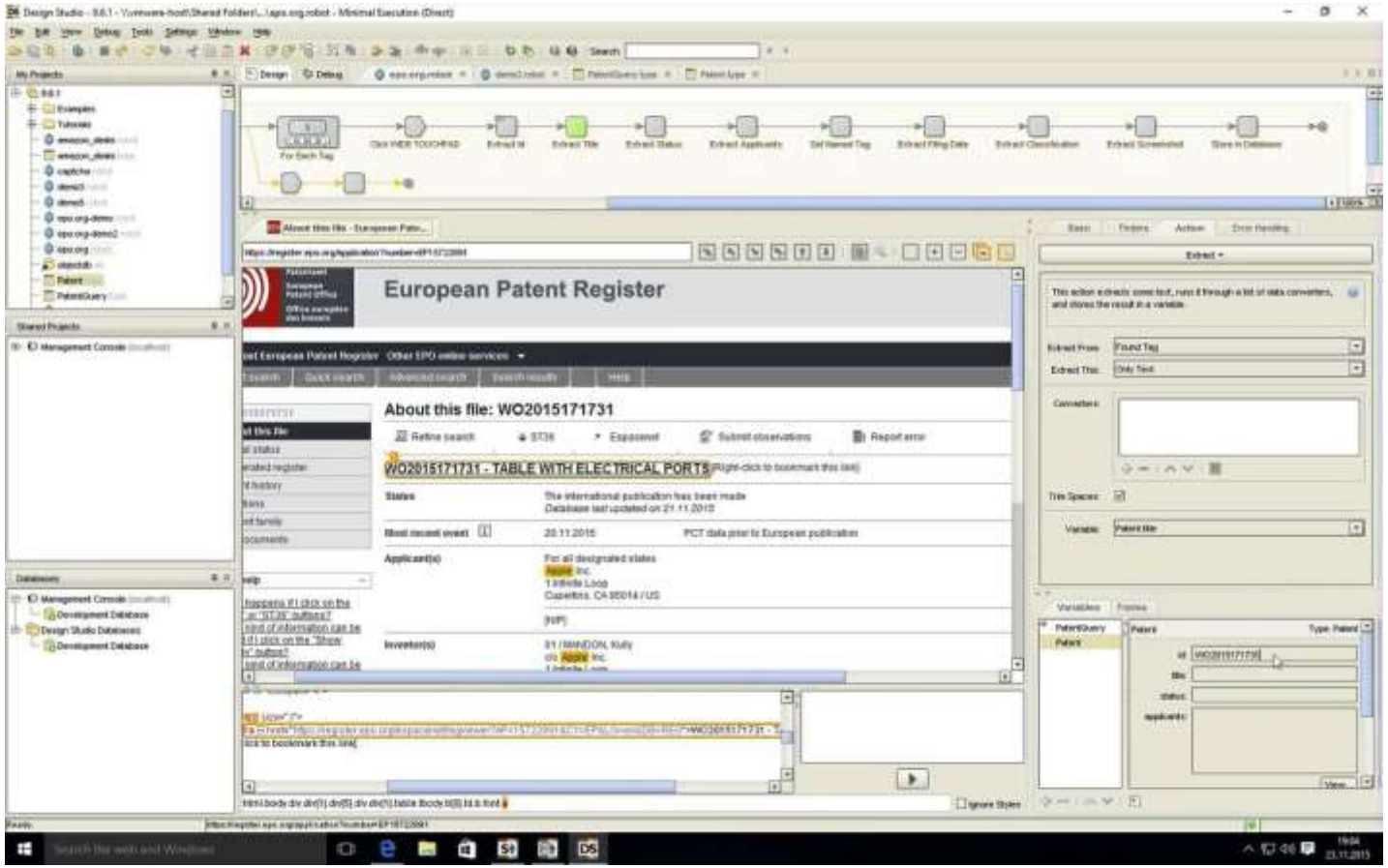
# Overview

- Commercial products
- What without Python?
- Web scraping versus web crawling
- Web scraping versus AI and ML
- Web scraping versus RPA
- From web scraping to <blank> scraping

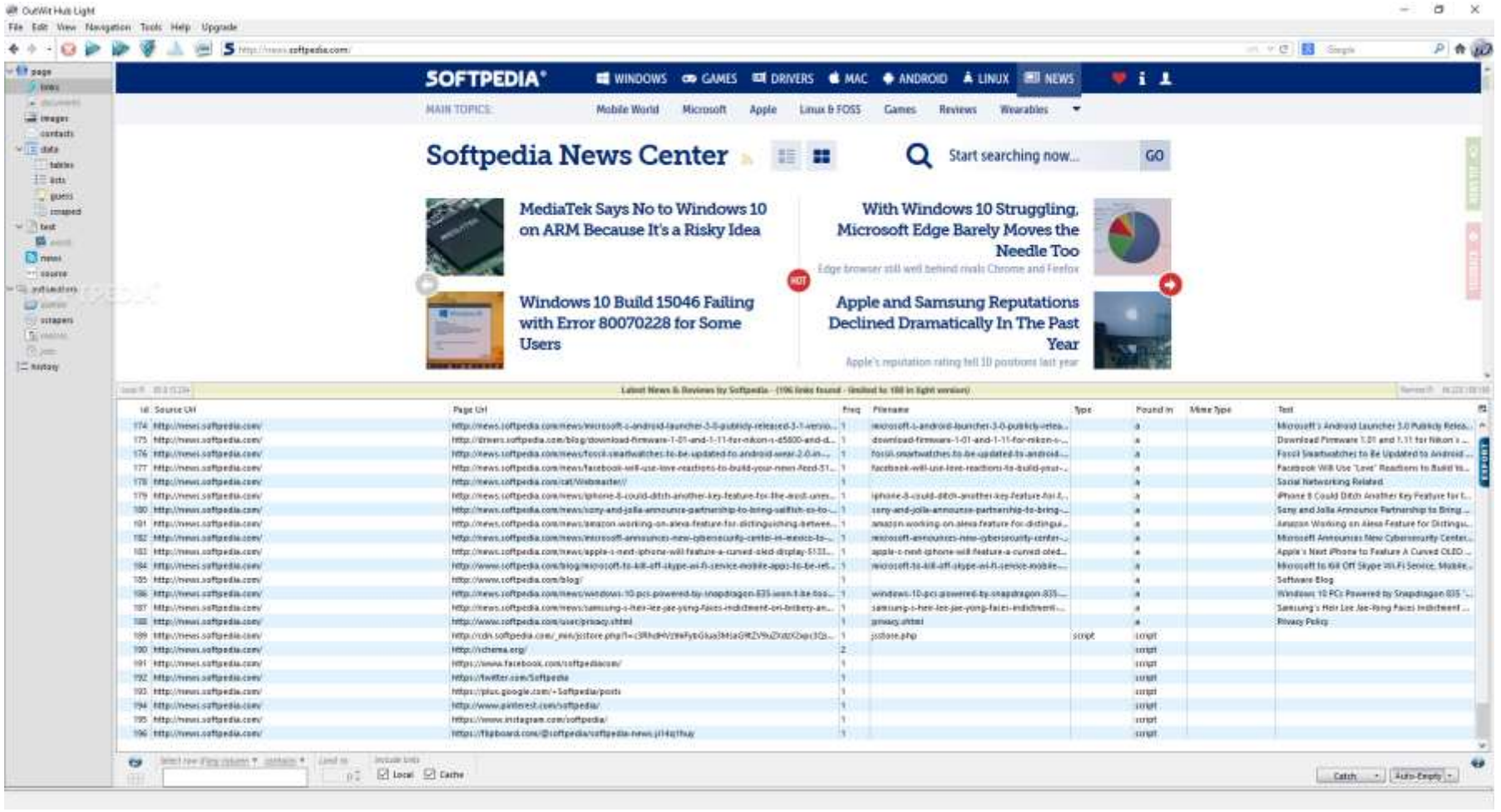
# Commercial products



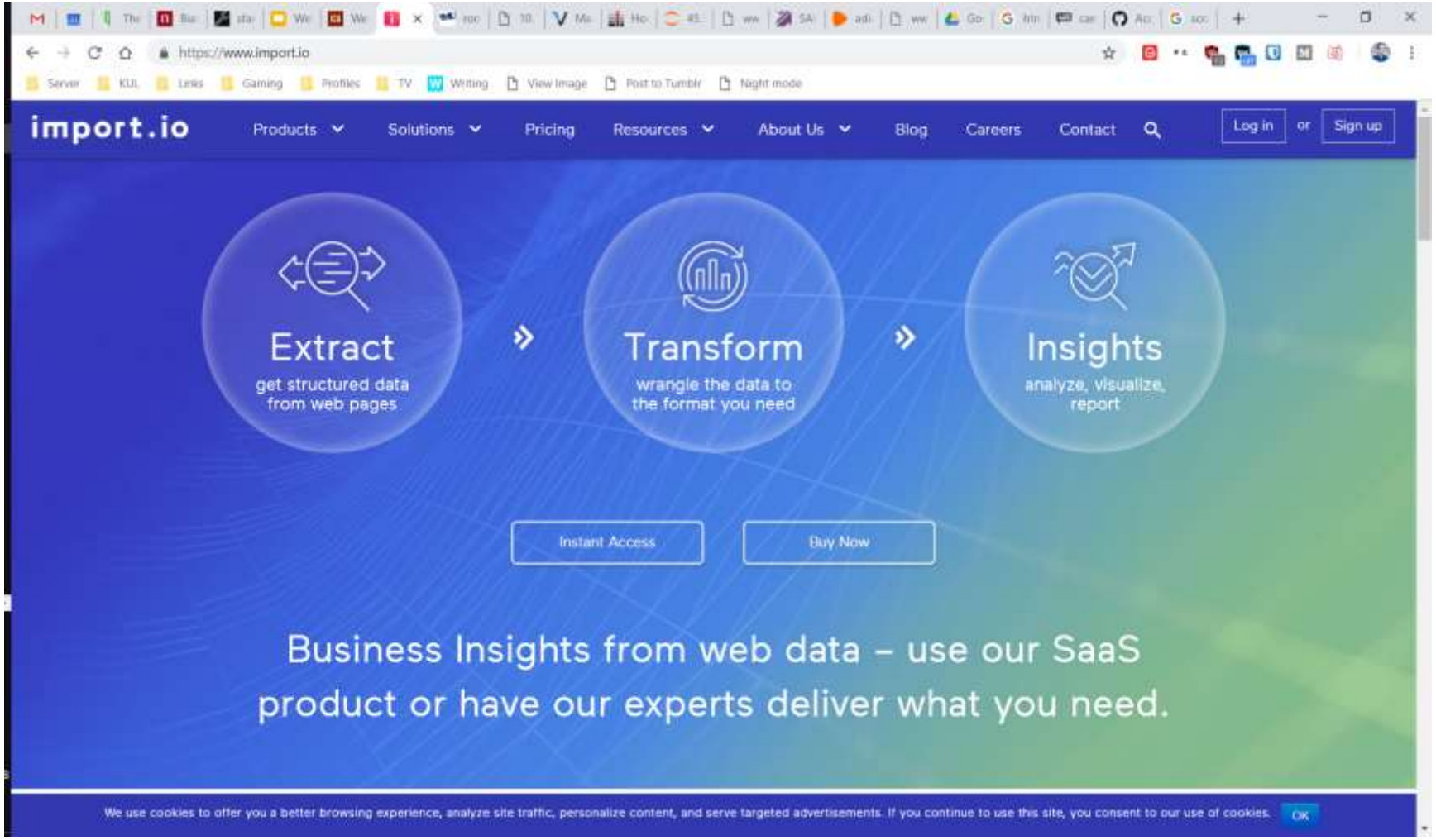
# Commercial products



# Commercial products



# Commercial products



# Commercial products

- Browser plugins
  - https://portia.scrapinghub.com
  - https://www.parsehub.com
  - Many others, even Google Sheets supports importing HTML
  - Very limited, e.g. simple tabular or list extraction, GET requests only

  - **Nice selection interface in most cases (“click what you want to get”, but breaks down in semi-complex use cases)**

# Commercial products

- Full package
  - https://www.kofax.com/data-integration-extraction
  - http://www.fminer.com/
  - https://dexi.io/
  - Often take a workflow oriented designer approach
  - But: expensive, who maintains the workflows? In terms of time: designing a simple workflow can also be rather time-consuming
  - Also: underlying browser engines not always very robust (e.g. JavaScript support) with no way to "code your way out of this"), no way to circumvent detection mechanisms
  - Generated element selection rules often very granular: so workflow breaks once site changes a little

  - **Consider maturity of team and vision regarding web scraping projects!**

# Commercial products

- Scraping helpers
  - Proxy servers
  - Cloud deployment
  - Captcha cracking services

# Other programming languages

- R: rvest or Selenium
  - Complex web sites a bit more cumbersome to set up
- C# and Java: Selenium
  - Other libraries exist in the .NET ecosystem as well
  - Also quite mature
- JavaScript: PhantomJS (can also be used as a “browser” for Selenium), Nightmare, SlimerJS, CasperJS
  - Mainly replaced by Puppeteer (https://github.com/GoogleChrome/puppeteer): high-level JavaScript API for headless Chrome
    - https://intoli.com/blog/running-selenium-with-headless-chrome/
    - https://pypi.python.org/pypi/pyppeteer
    - https://github.com/ariya/phantomjs/issues/15344: Archiving the project: suspending the development
    - Similar Firefox project exists
  - Common use case: develop in Selenium using standard browser (for debugging), then port to Puppeteer
- IE as browser driver: not recommended (older commercial tools use this e.g. through ActiveX component: slow and prone to crashing)

# Chrome headless

```
from selenium import webdriver


options = webdriver.ChromeOptions()
options.add_argument('headless')
options.add_argument('window-size=1200x600')


driver = webdriver.Chrome(options=options)
```

# Web scraping vs. web crawling

- The difference between “web scraping” and “web crawling” is somewhat vague
  - Many authors and programmer will use both terms interchangeably
  - In general terms, the term “crawler” indicates a program's ability to navigate web pages on its own, perhaps even without a well-defined end-goal or purpose
  - Also often called a web spider
  - Web crawlers are heavily used by search engines like Google to retrieve contents for a URL, examine that page for other links, retrieve the URLs for those links, and so on

# Web scraping vs. web crawling

- Design choices:
  - In many cases, crawling will be restricted to a well-defined set of pages, e.g. product pages of an online shop
    - These cases are relatively easy to handle, as you're staying within the same domain and have an expectation about what each product page will look like, or about the types of data you want to extract
  - In other cases, you will restrict yourself to a single website (a single domain name), but do not have a clear target regarding information extraction in mind. Instead, you simple want to create a copy of the site
    - In such cases, manually writing a scraper is not an advisable approach
    - There are many tools available (for Windows, Mac, and Linux) which will help you to make an offline copy of a website, including lots of configurable options (wget, curl, GUI based tools)
  - You might want to keep your crawling very open ended. For example, you might wish to start from a series of keywords, Google each of them, crawl to the top ten results for every query, and crawl those pages for, say, images, tables, articles, and so on
    - Advanced use case
    - Requires well thought out setup!

# Web scraping vs. web crawling

- Think carefully about which data you actually want to gather
  - Can you extract what you need by scraping a set of predefined websites, or do you really need to discover websites you don't know about yet?
  - The first option will always lead to easier code in terms of writing and maintenance
- Use a database
  - It's best to use a database to keep track of links to visit, visited links, and gathered data
  - Make sure to timestamp everything so you know when something was created and last updated
  - Keep versions of scraped data
- Separate crawling from scraping
  - Most robust crawlers separate the “crawling” part (visiting websites, extracting links and putting them in a queue, i.e. gathering the pages you wish to scrape) from the actual “scraping” part (extracting information from pages). Doing both in one and the same program or loop is quite error-prone
  - In some cases, it might be a good idea to have the crawler store a complete copy of a page's HTML contents so that you don't need to revisit it once you want to scrape out information
- Stop early
  - When crawling pages, it's always a good idea to incorporate stopping criteria as soon as possible
  - That is, if you can already determine that a link is not interesting at the moment of seeing it, don't put it in the “to crawl” queue
  - The same applies when you scrape information from a page. If you can quickly determine that the contents are not interesting, then don't bother continuing with that page

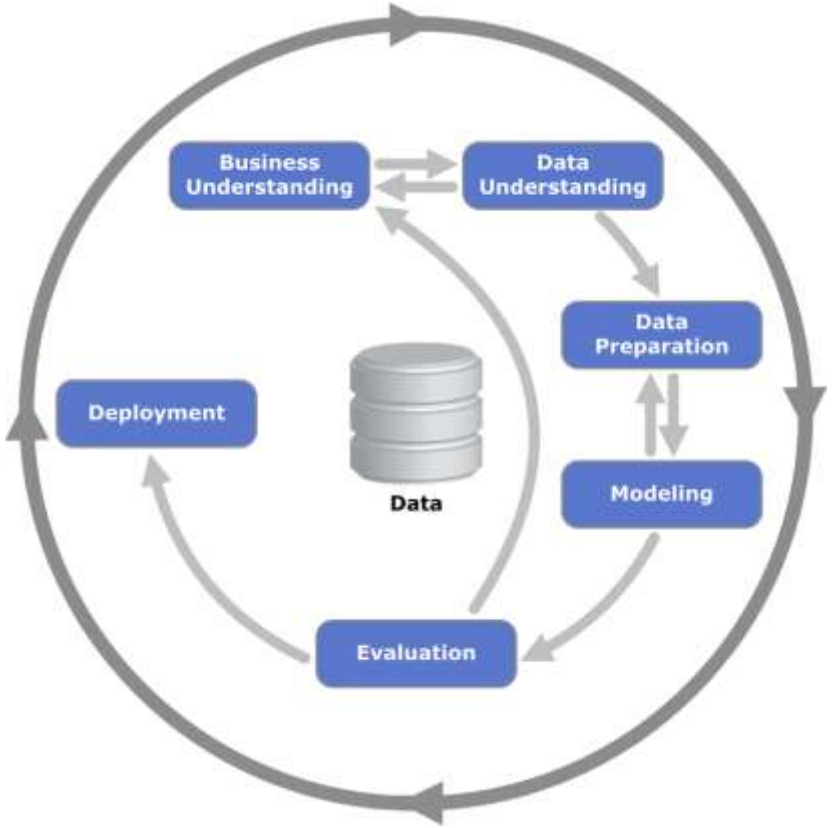
# Web scraping vs. web crawling

- Retry or abort
  - Note that the web is a dynamic place, and links can fail to work or pages can be unavailable
  - Think carefully about how many times you'll want to retry a particular link
- Crawling the queue
  - That said, the way how you deal with your queue of links is important as well
  - If you just apply a simple FIFO (first in first out) or LIFO (last in first out) approach, you might end up retrying a failing link in quick succession, which might not be what you want to do. Building in cool down periods is hence important as well
- Parallel programming
  - In order to make your program efficient, you'll want to write it in such a way that you can spin up multiple instances which all work in parallel
  - Hence the need for a database backed data store as well. Always assume that your program might crash at any moment and that a fresh instance should be able to pick up the tasks right away

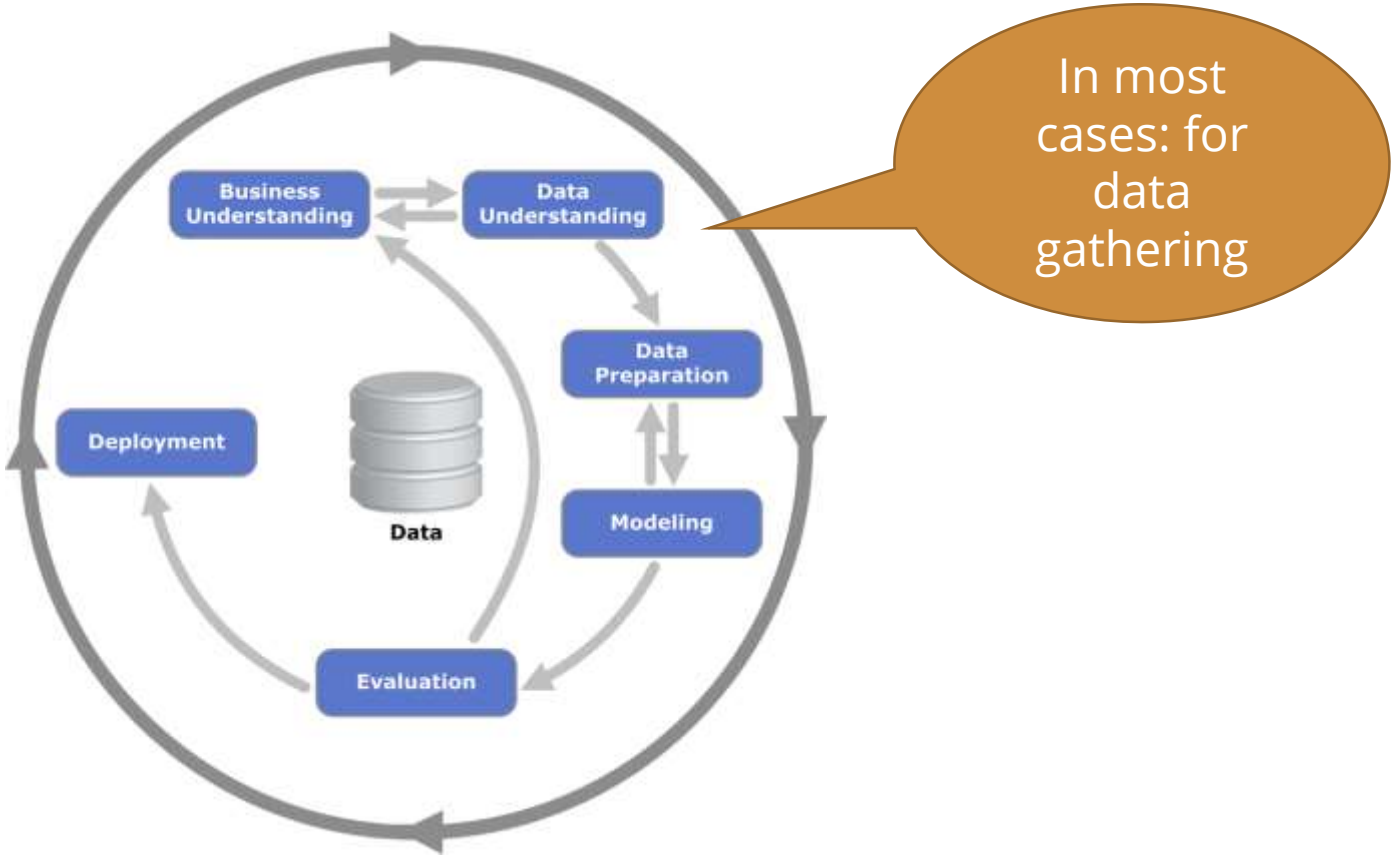
- **Keep in mind the legal aspects of scraping!**
- **Usage of pre-built APIs can be beneficial here (e.g. for news article scraping)**

# Web scraping vs. AI and ML

Where does web scraping fit?



# Web scraping vs. AI and ML

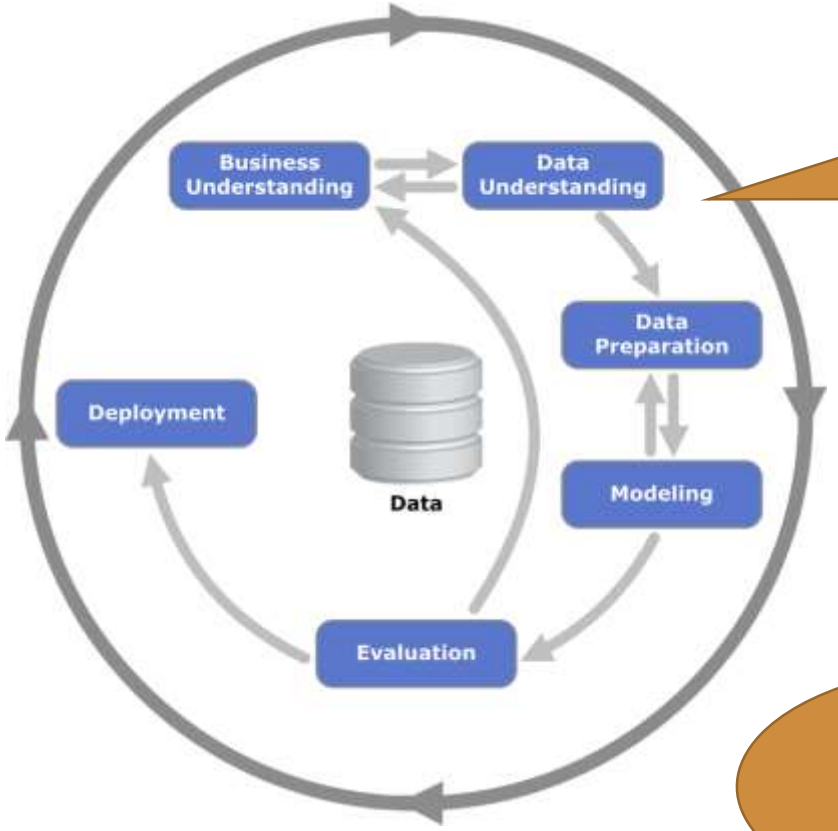


# Web scraping vs. AI and ML


In most cases: for data gathering

To improve predictive models

# Web scraping vs. AI and ML

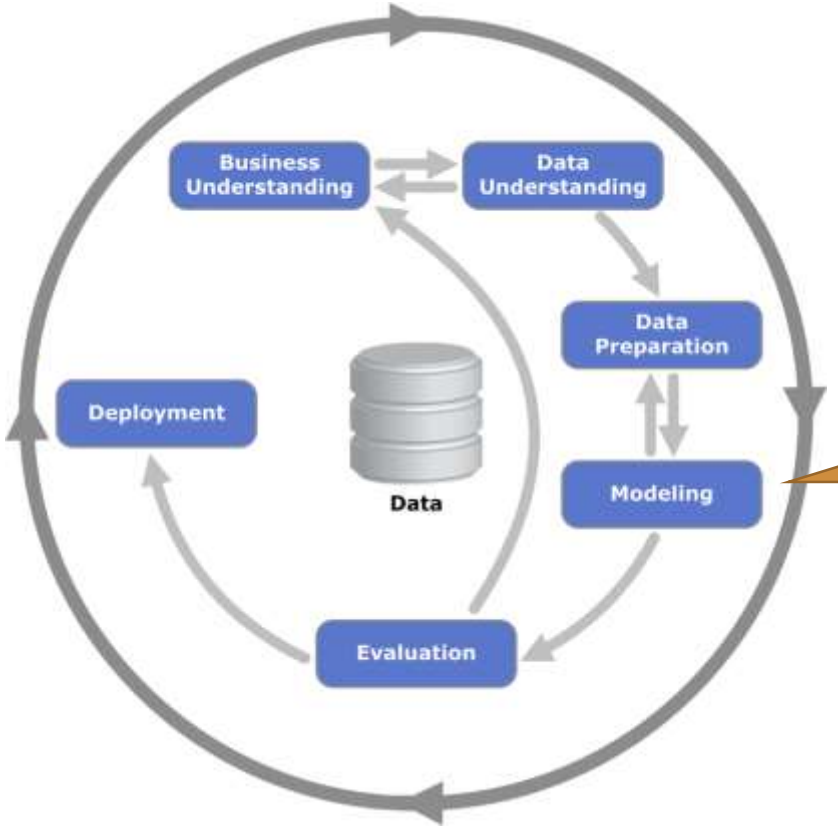



In most cases: for data gathering

To improve predictive models

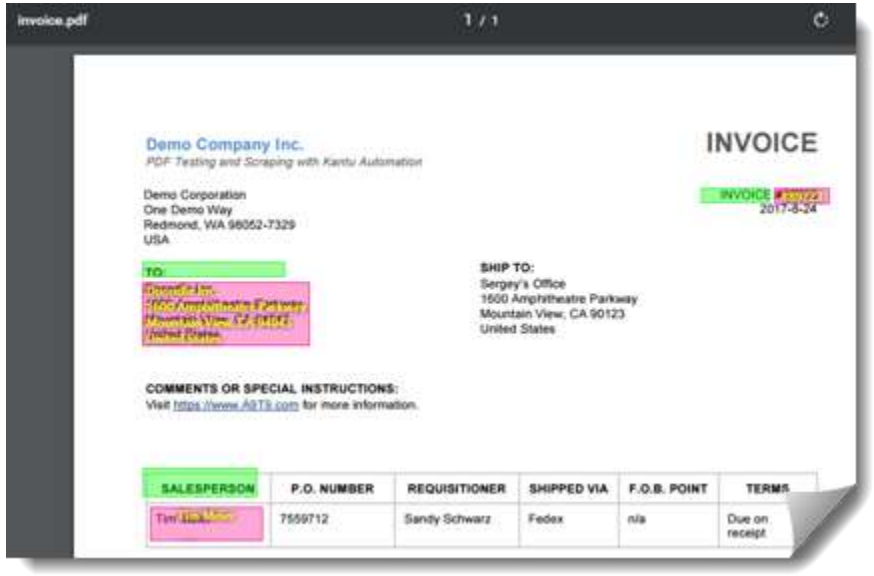
Requires some managerial considerations (see later)

# Web scraping vs. AI and ML

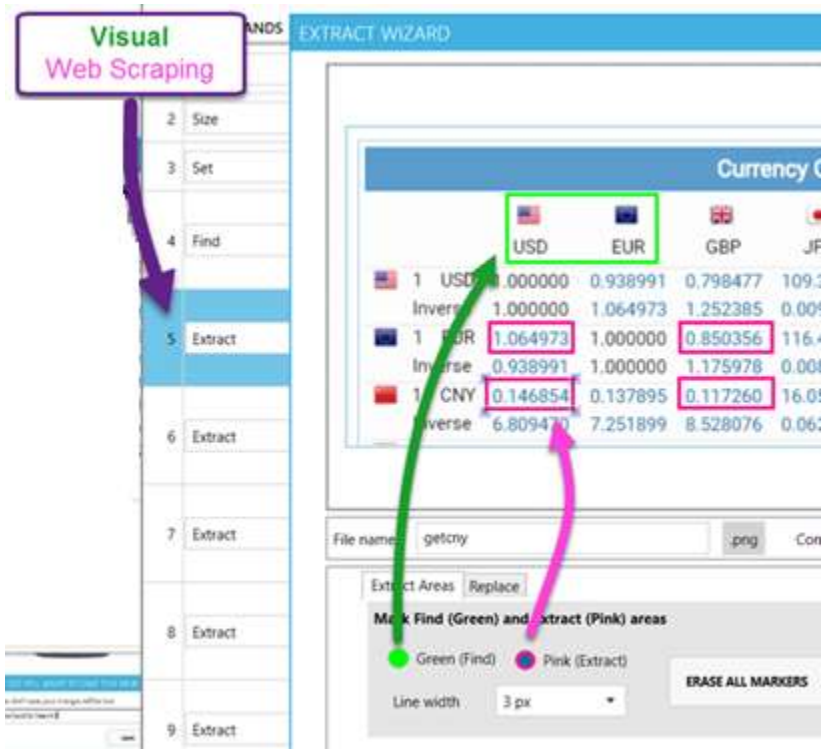



Sometimes, ML and AI also used in scrapers

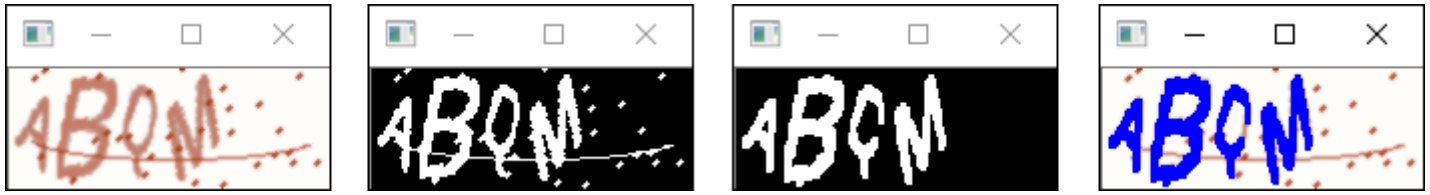
# Web scraping vs. AI and ML



# Web scraping vs. AI and ML



# Web scraping vs. AI and ML



# Web scraping vs. AI and ML

- Text mining
- OCR
- “Visual web scraping”
- Computer vision
- Captcha cracking

- **Used in heavy crawling projects where the scraper is the main product!**
  - **Often additional tooling required**

# Web scraping vs. RPA

- Robotic Process Automation
  - To automate simple back-end processes
  - “Second wave of automation”

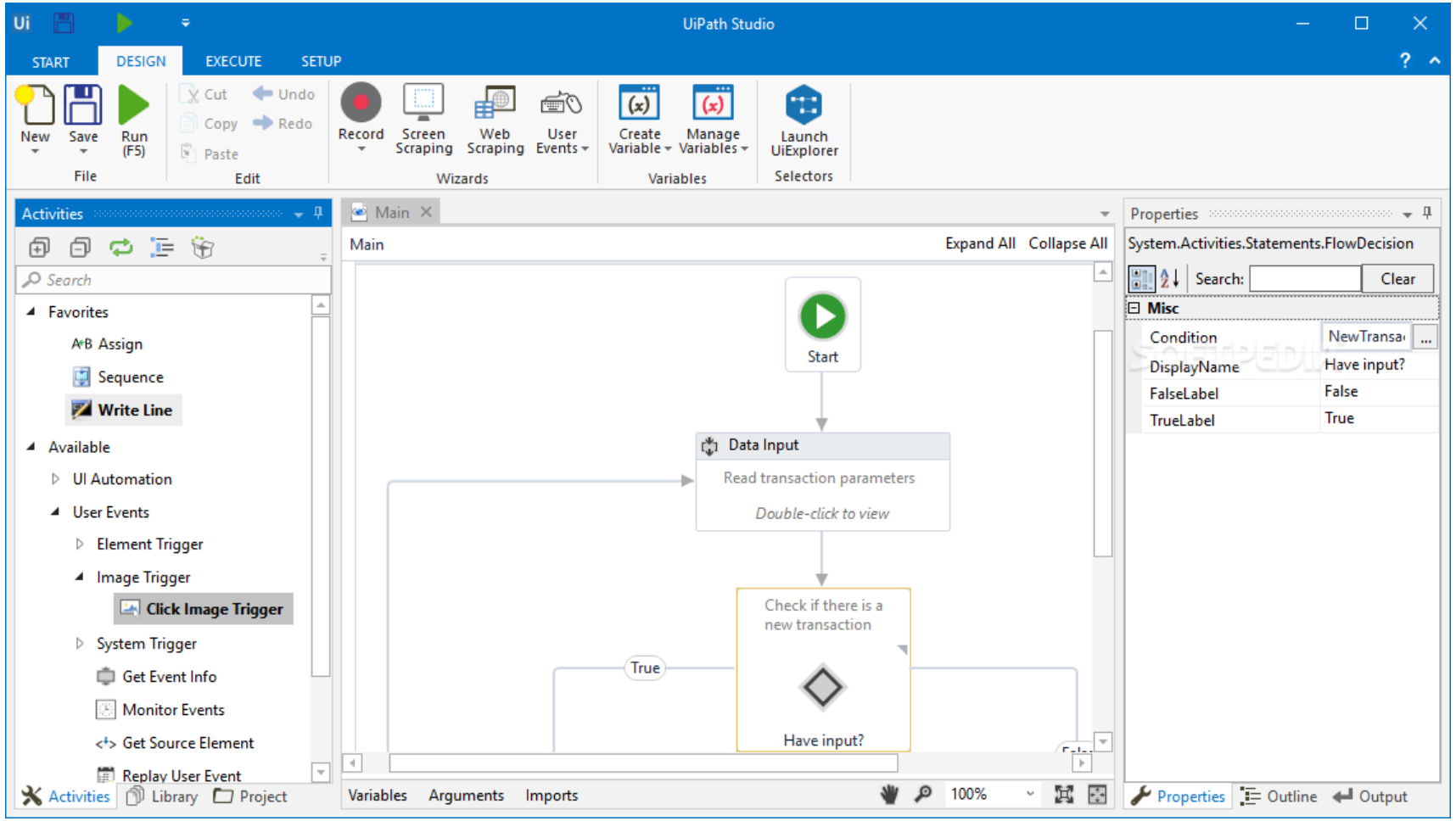
# Web scraping vs. RPA


Source: HfS

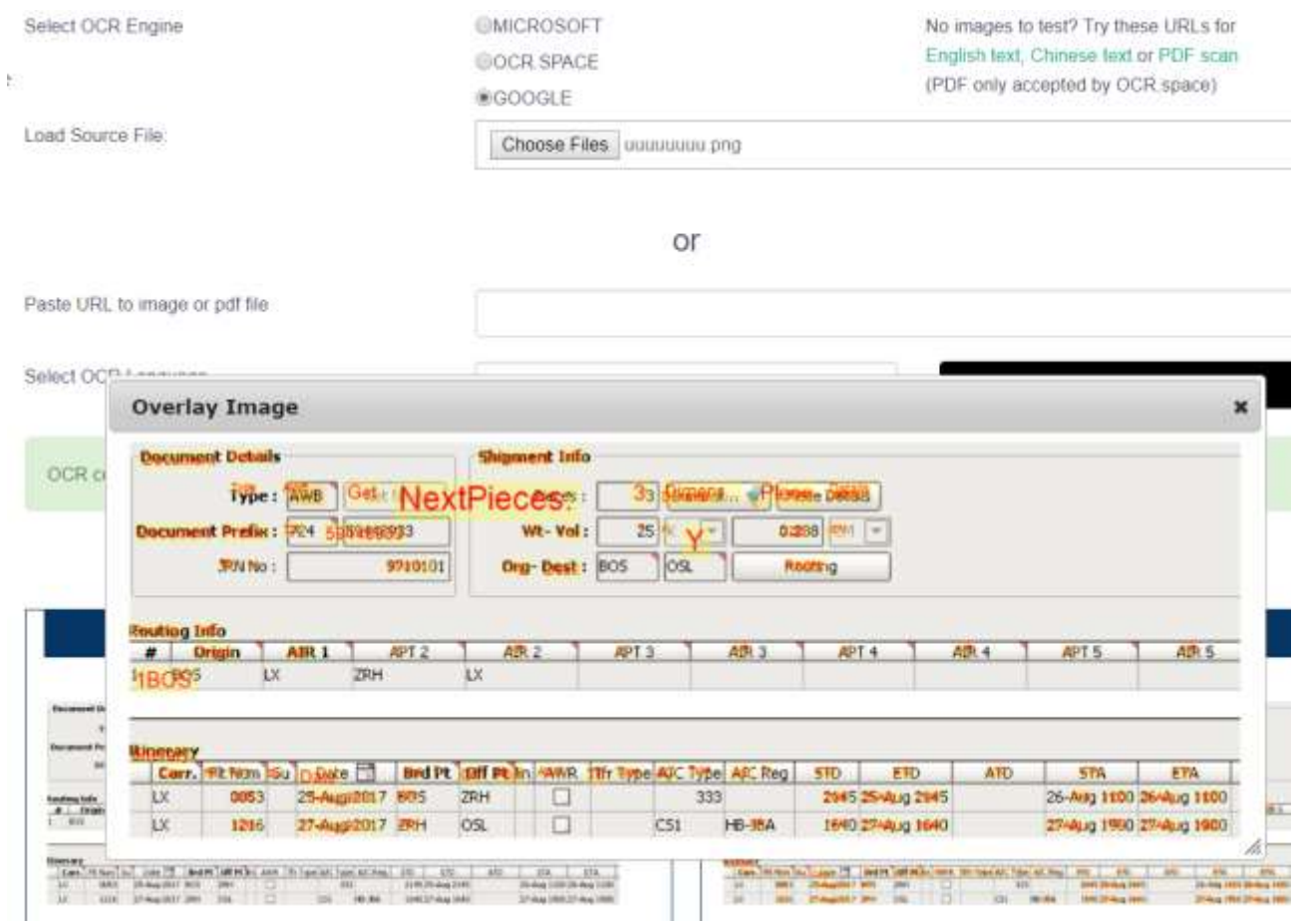
# Web scraping vs. RPA


Source: Forrester

# Web scraping vs. RPA



# Web scraping vs. RPA



# Web scraping vs. RPA

- Many commercial web scraping tools have rebranded themselves as RPA
- RPA: web scraping but also + PDF scraping + screen (UI) scraping
- Also a workflow oriented design
- Maturity is higher than before
  - Though can still be tricky with complex sites
  - Expensive licenses
  - Maintenance required for workflows
  - Also not 100% accurate (OCR, granular selection rules)
  - But: interesting to consider if value-proposition also there for typical RPA-purposes and scraping requirements not that high

# <blank> scraping

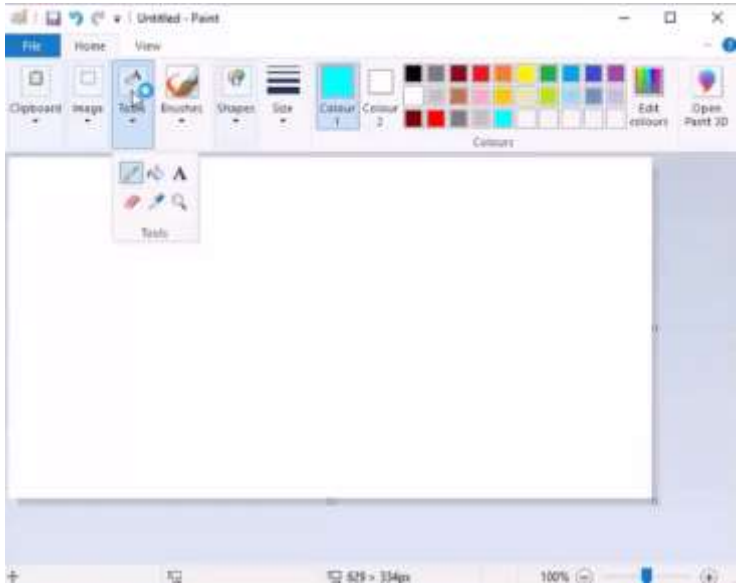
- PDF scraping?
  - PDF to text tools
  - PDF libraries (https://github.com/pmaupin/pdfrw, for example)
  - Tabula: for table extraction (https://tabula.technology/)
  - Camelot:
    - https://blog.socialcops.com/technology/engineering/camelot-python-library-pdf-data/
    - Newer tool
    - Very good tabular extraction!
    - Results as good as commercial (smallpdf and pdftables)

# <blank> scraping

- OCR
  - Tesseract (https://github.com/tesseract-ocr/) still reasonable in terms of complexity versus power
    - Does require manual training
- Unstructured text
  - Toolkits such as SpaCy, AllenNLP, nltk, …
- Computer vision
  - Using deep learning approaches to detect objects or scenes

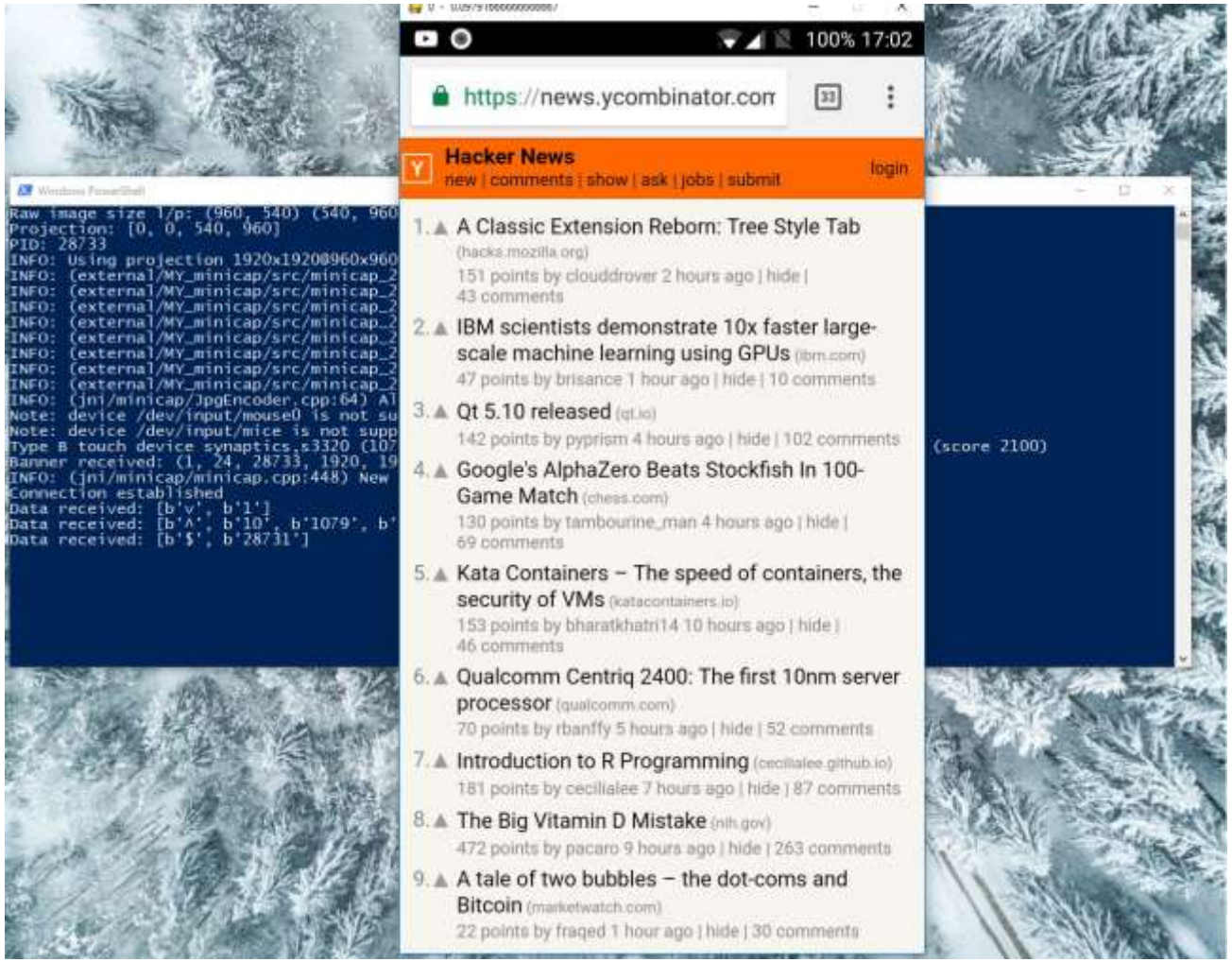
# <blank> scraping

- Screen scraping and instrumentation
  - https://pypi.org/project/uiautomation/
  - https://github.com/pywinauto/pywinauto
  - https://pywinauto.github.io/
  - https://github.com/OakwoodAI/Automagica



- Mobile APP scraping
  - Using e.g. Android emulator combined with automation framework or screen scraping

# <blank> scraping



# <blank> scraping

- Other proprietary web technologies
  - Java applets: can be easily decompiled
  - Flash: similar
  - WebGL: still a relatively hard case to crack (requires heavy debugging tools setup)
    - https://www.khronos.org/webgl/wiki/Debugging
    - http://www.realtimerendering.com/blog/debugging-webgl-with-spectorjs/
    - http://www.realtimerendering.com/blog/webgl-debugging-and-profiling-tools/
    - https://benvanik.github.io/WebGL-Inspector/

# Managerial Aspects

# Overview

- Legal concerns
- Web scraping as part of your data science pipeline
- Buy or build?
- When not to opt for web scraping?
- How to manage a web scraping project?

# Legal concerns

- “In the case of Ticketmaster vs. Riedel Marketing Group (RMG), the latter was web scraping Ticketmaster's site so that it could harvest large quantities of desirable tickets for resale. Ticketmaster argued that RMG had agreed to the terms and conditions of the site but ignored them and the court held that RMG had infringed on Ticketmaster’s copyrighted material.”
- “In Ryanair Ltd vs. PR Aviation BV, the European Court found that Ryanair was free to create contractual limits on the use of its database, and the case was hence ruled in its favor.”
- “In 2006, Google got involved in a long-winding legal battle with Belgian media firm Copiepresse. This led to an ugly battle between Copiepresse and Google, ending with the two of them reaching an agreement to include the sites again in 2011.”
- “In Facebook vs. Power Ventures, Facebook also claimed that the defendant has violated the CFAA and the CAN-SPAM Act, a federal law that prohibits sending commercial e-mails with materially misleading information. The judge ruled in favor of the plaintiff.”

# Legal concerns

- “LinkedIn and Microsoft executives, were understandably not so very happy with this state of affairs. The data belonged to LinkedIn, so they thought, and a cease and desist order was sent out to request hiQ to stop scraping LinkedIn's data, as well as various technical measures were implemented to keep hiQ Labs' bots out.”
  - https://medium.com/@chris_70736/hiq-v-linkedin-and-the-legality-of-web-scraping-e80b9ab06f1d
  - “On August 14, 2017, **the judge granted hiQ’s motion for a temporary restraining order preventing LinkedIn from blocking hiQ’s access to their site while the case was pending**, the decision which LinkedIn then appealed to the Ninth Circuit.”

# Legal concerns

- Alan Ross Machinery Corp. v. Machino Corporation (November 16, 2018)
  - Violation of the Copyright Act's prohibitions against distributing false copyright management information ("CMI") (17 U.S.C. §1202(a)), and removing or altering CMI (Id., §1202(b))
  - Violation of the Lanham Act for reverse passing off and false endorsement
  - http://blog.internetcases.com/2018/11/19/web-scraping-case-fails-under-dastar/
  - "The **court dismissed the claim asserting distribution of false CMI because plaintiff alleged that the false CMI was a blanket copyright notice found on defendant's website's terms of use, and not on the pages where the copied content was displayed**. It also dismissed plaintiff's claim for removal of CMI because it found that the allegations concerning the CMI allegedly removed – a copyright notice found at the bottom of the pages of plaintiff's website – covered the pages of the website itself, not the particular listings that were allegedly copied without the CMI."

# Legal concerns

- Breach of Terms and Conditions (US)
- Copyright or Trademark Infringement (US)
- Computer Fraud and Abuse Act (CFAA) (US)
- Trespass to Chattels (US)
- Robots Exclusion Protocol (industry agreement)
- The Digital Millennium Copyright Act (DMCA), CAN-SPAM Act, … (US)
- The EU Database Directive of 1996
- The Computer Misuse Act and Trespass to Chattels (UK)
- Computer Misuse Act 1990 (UK)
- General Data Protection Regulation (GDPR) (EU)
- Article 13 and 11 (EU)

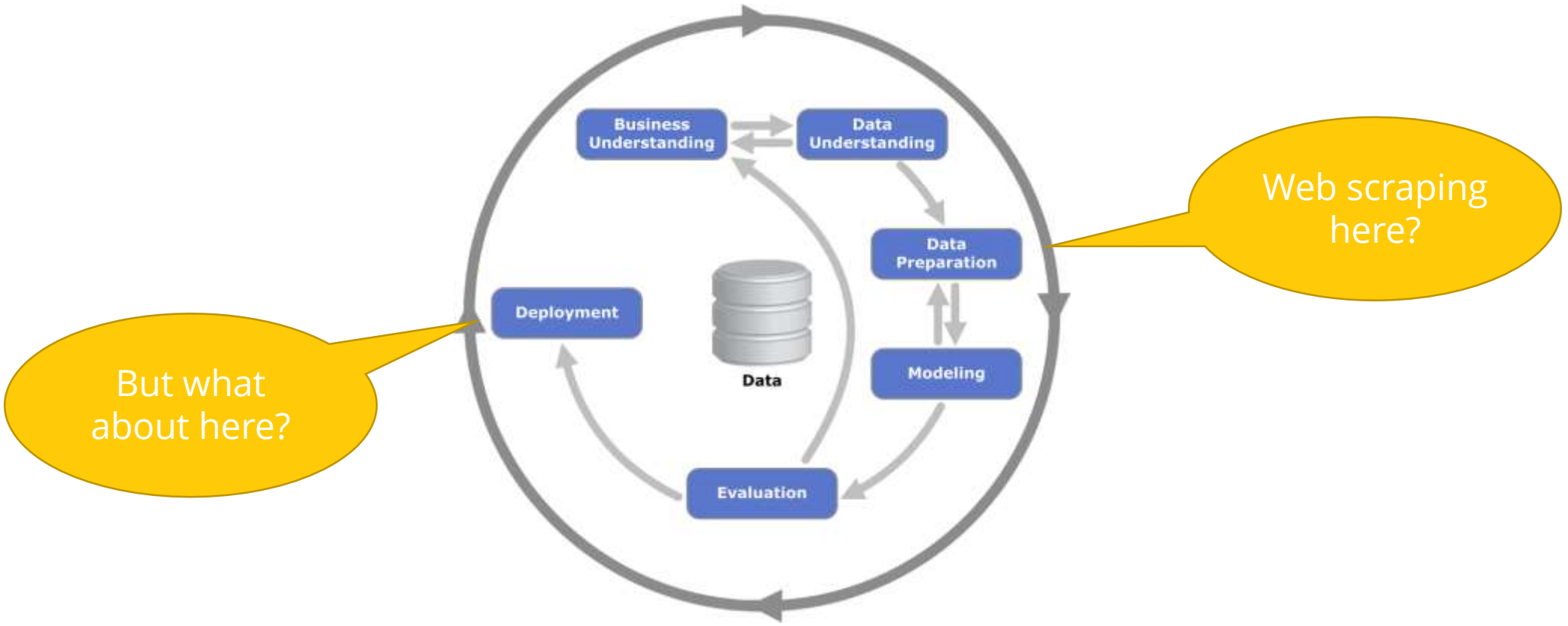
# Legal concerns

- Copyright
- Privacy
- Breach

# Legal concerns

- Get written permission
  - The best way to avoid legal issues is to get written permission from a website's owner covering which data you can scrape and to which extent
- Check the terms of use
  - These will often include explicit provisions against automated extraction of data
  - Oftentimes, a site's API will come with its own terms of use regarding usage, which you should check as well
- Public information only
  - If a site exposes information publicly, without explicitly requiring acceptance of terms and conditions, moderated scraping is most likely fine
  - Don’t login into sites
- And no personal information
  - Privacy concerns
- Don't cause damage
  - Don't hammer websites with lots of requests, overloading their network and blocking them of normal usage
  - Stay away from protected computers and do not try to access servers you're not given access to
- Copyright and fair use
  - Copyright law seems to provide the strongest means for plaintiffs to argue their case, so far
  - Check carefully whether your scraping case would fall under fair use and do not use copyrighted works in commercial projects.Bekijk de terms of use
- Check robots.txt
  - And behave accordingly
- **Allowed, but no private information, no personal information, no copyrighted works, not on a massive scale…**
  - And still everybody does it

# Web scraping as part of data science



# Web scraping as part of data science

- A one-shot project where web scraping can offer valuable data?
  - E.g. a single report, or descriptive model
  - Then not really a deployment or maintainability issue
- A predictive model trained on web scraped data?
  - Which features need to be refeshed?
  - Puts extra pressure on production
  - How long can we use the scraped data? What if the data source goes down? How long will the model be used for? How important is the model?
  - GIGO: clean as much as possible during or right after scraping
  - When used in reporting context: same concerns in case reporting is continuous and repeated!
- Cultural issue: “why can’t we use / don’t we have Facebook’s data?”
  - Consider internal data sources

# Managerial concerns

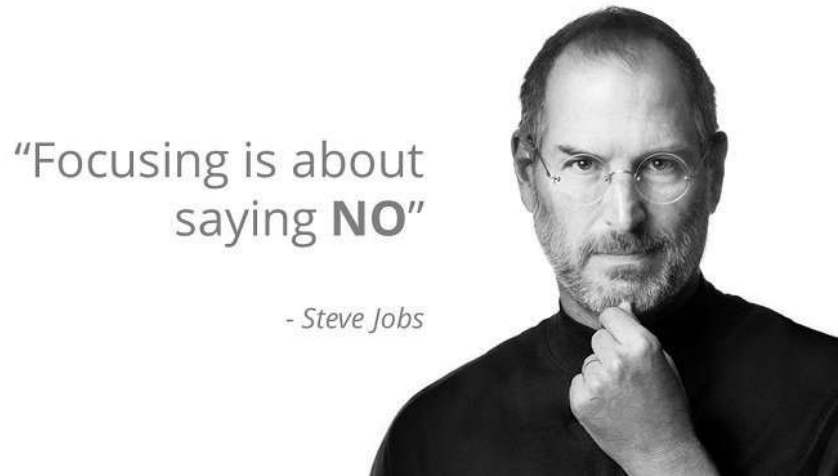
- RPA has lead in a number of new job roles
  - Robot supervisor, robot developer, idea champion
- Similar roles apply in a web scraping context: multidisciplinary team including
  - Database expert
  - Programmers
  - Data scientists
  - Web developers
  - Compliance
  - Management

# Managerial concerns

- Scope, scale and size depends on view on web scraping projects
  - From strategic core of main business (e.g. aggregator sites, real estate, …) to tactical (scraping competitors) to simply operational or one-off projects
  - Development and governance environment to be scoped accordingly
- Consider buy versus build decision
  - Build: initial training and setup cost, offers more flexibility in long run
  - Buy: quicker to get started, though expensive and less flexible
  - **Both require stringent maintenance governance**

# Managerial concerns

- Saying no
  - When there is no well-defined question (e.g. we want a copy of Facebook)
  - When the legal risk is deemed too high
  - When technological capabilities are lacking
  - When there is no clear value



# Conclusions

# The cat and mouse game

- Websites take increasingly more advanced measures to block scrapers
  - Rate limiting
  - IP blocking
  - Browser checking
  - JavaScript based checks
  - HTML and JavaScript obfuscation
  - UI event fingerprinting

# The cat and mouse game

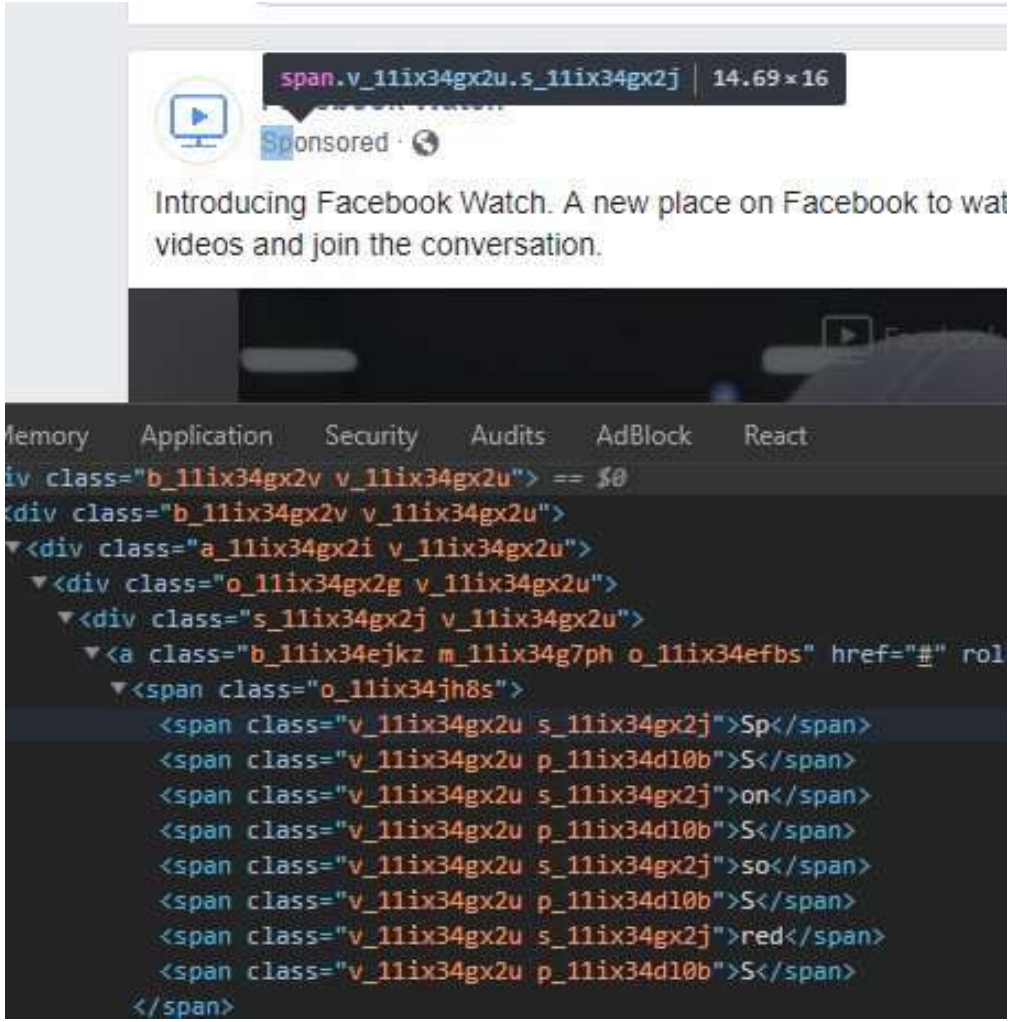
**Checking your browser before accessing example.com**

This process is automatic. Your browser will redirect to your requested content shortly.

Please allow up to 5 seconds…

DDoS protection by CloudFlare

# The cat and mouse game



# The cat and mouse game

“**There was quite speculation recently over browser plugins (extensions) which are causing some LinkedIn accounts to be suspended.** To avoid any doubts, I scrutinised the LinkedIn source code to find out what exactly LinkedIn checks every time we open the LinkedIn.com website. Source code never lies. The fact that LinkedIn tracks some plugins doesn’t necessarily mean that they are illegal or prohibited in any way. While there might be other reasons for the suspensions, there is no doubt the plugin list is full of data and email scrapers that LinkedIn disapproves of. Even if some of them are produced by established brands such as Hubspot and TalentBin. **As of writing this article, LinkedIn is checking over 25 plugins for the LinkedIn basic accounts.** If you have the Recruiter Lite, RPS or Recruiter account, you are being checked for other sets of plugins on top of that.”

https://www.josefkadlec.com/blog/the-complete-list-of-prohibited-linkedin-plugins

# The cat and mouse game

- Avoidance
  - Fake as much as possible: User-Agent, Referer, other headers, cookies, order of parameters
  - Use proxy's or the cloud: but not all providers welcome scrapers, and not all websites like all providers
  - Timing and retry mechanisms
  - Captcha's: http://www.deathbycaptcha.com, OCR, deep learning
  - Fake UI events (typing speed, mouse movement, scrolling)

# The cat and mouse game

- http://www.deathbycaptcha.com
- Other services: click farms, Amazon Mechanical Turk

# The cat and mouse game

```
Using CNTK backend

Selected GPU[0] GeForce GTX 980M as the process wide default device.


Train on 1665 samples, validate on 555 samples

Epoch 1/10

 32/1665 [..............................] - ETA: 36s - loss: 3.0294 - acc: 0.0312

 64/1665 [>.............................] - ETA: 22s - loss: 5.1515 - acc: 0.0312

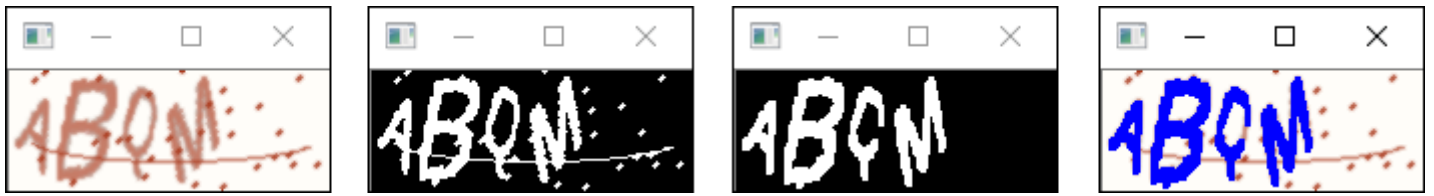
[...]

1600/1665 [===========================>..] - ETA: 0s - loss: 7.6135e-04 - acc: 1.0000

1632/1665 [============================>.] - ETA: 0s - loss: 8.3265e-04 - acc: 1.0000

1664/1665 [============================>.] - ETA: 0s - loss: 8.2343e-04 - acc: 1.0000

1665/1665 [==============================] - 3s 2ms/step - loss: 8.2306e-04 - acc: 1.0000 - val_loss: 0.3644 - val_acc: 0.9207
```



# The cat and mouse game

- Also see
  - https://medium.com/towards-data-science/deep-learning-drops-breaking-captcha-20c8fc96e6a3
  - https://medium.com/@ageitgey/how-to-break-a-captcha-system-in-15-minutes-with-machine-learning-dbebb035a710
  - http://www.npr.org/sections/thetwo-way/2017/10/26/560082659/ai-model-fundamentally-cracks-captchas-scientists-say
- Check whether the captcha appears every time, or only after some amount of time or every so often

# The cat and mouse game

- https://blog.shapesecurity.com/2015/01/22/detecting-phantomjs-based-visitors/
- https://intoli.com/blog/making-chrome-headless-undetectable/
- http://antoinevastel.github.io/bot%20detection/2017/08/05/detect-chrome-headless.html

```
if(navigator.plugins.length == 0) {
   console.log("It may be Chrome headless");
}

var canvas = document.createElement('canvas');
var gl = canvas.getContext('webgl');

var debugInfo = gl.getExtension('WEBGL_debug_renderer_info');
var vendor = gl.getParameter(debugInfo.UNMASKED_VENDOR_WEBGL);
var renderer = gl.getParameter(debugInfo.UNMASKED_RENDERER_WEBGL);

if(vendor == "Brian Paul" && renderer == "Mesa OffScreen") {
   console.log("Chrome headless detected");
}
```

# Closing best practices

- **Go for an API first**
  - Always check first whether the site you wish to scrape offers an API. If it doesn't, or it doesn't provide the information you want, or it applies rate limiting, then you can decide to go for a web scraper instead
- **Use the best tools**
  - Don’t speak HTTP manually, but know how it works
  - Don't parse HTML manually, use a parser such as Beautiful Soup instead of trying to untangle the soup manually
- **Play nice**
  - Don't hammer a website with hundreds of HTTP requests, as this will end up with a high chance of you getting blocked
  - Consider contacting the webmaster of the site and work out a way to work together
- **Consider the user agent and referrer**
  - Remember the “User-Agent” and “Referer” headers
  - Many sites will check these to prevent scraping or unauthorized access, this is often your first check to perform

# Closing best practices

- **Web servers are picky**
  - Whether it's URL parameters, headers, or form data, some web servers come with very picky and strange requirements regarding their ordering, presence, and values
  - Some might even deviate from the HTTP standard
- **Check your browser**
  - If you can't figure out what's going wrong, start from a fresh browser session and use your browser's developer tools to follow along through a normal web session—preferably opened as an “Incognito” or “private browsing” window (to make sure you start from an empty set of cookies)
  - If everything goes well there, you should be able to simulate the same behavior as well
  - Remember that you can use “curl” and other command line tools to debug difficult cases
- **Before going for a full JavaScript engine, consider internal APIs**
  - Check your browser's network requests to see whether you can access a data source used by JavaScript directly before going for a more advanced solution like Selenium
- **Assume it will crash**
  - The web is a dynamic place. Make sure to write your scrapers in such a way that they provide early and detailed warnings when something goes wrong

# Closing best practices

- **Crawling is hard**
  - When writing an advanced crawler, you'll quickly need to incorporate a database, deal with restarting scripts, monitoring, queue management, timestamps, and so on to create a robust crawler
- **Some tools are helpful, some are not**
  - There are various companies offering “cloud scraping” solutions, like e.g. Scrapy
  - The main benefit of using these is that you can utilize their fleet of servers to quickly parallelize a scraper
  - Don't put too much trust in expensive graphical scraping tools, however. In most cases, they'll only work with basic pages, cannot deal with JavaScript, or will lead to the construction of a scraping pipeline that might work but uses very fine-grained and specific selector rules which will break the moment the site changes its HTML a little bit
- **Scraping is a cat-and-mouse game**
  - Some sites can go very far in their use of ML, profiling and detection tools
- **Keep in mind the managerial and legal concerns, and where web scraping fits in your data science process**
  - As discussed, consider the data quality, robustness and deployment challenges that come with web scraping
  - Similarly, keep in mind the potential legal issues that might arise when you start depending on web scraping a lot or start to misuse it

# End