Session 8

Scraping 1 - Data Collection

Snorre Ralund

Motivation

Getting information and data on a subject of interest is no longer reduced to *tedious* survey designs, and *expensive* collections, *time consuming* interviews and *standardized* register data.

This session will teach you how to harvest some of the billions of data points being generated and shared on the internet everyday.

We need data now! fast, free of charge, and in abundance.

Agenda

Collecting raw data (*Tomorrow we shall focus on parsing and cleaning*)

- Lots of hand-on examples.
- The basics of webscraping
    - Connecting, Crawling, Parsing, Storing, Logging.
- Ethical and Legal issues around scraping publicly available data.
- Hacks: Backdoors, url construction and analyzing webpages.
- Reliability of your data collection.

Main take-aways

- Utilize the data sources around you.
- Knowing how to build your own custom datasets from web sources, without having to spend a month manually curating the data.
- Get to know some of the most valuable tricks,
- And instructions on how to Handle with care.

Ethics / Legal Issues

- If a regular user can't access it, we shouldn't try to get it (That is considered hacking)https://www.dr.dk/nyheder/penge/gjorde-opmaerksom-paa-cpr-hul-nu-bliver-han-politianmeldt-hacking (https://www.dr.dk/nyheder/penge/gjorde-opmaerksom-paa-cpr-hul-nu-bliver-han-politianmeldt-hacking).
- Don't hit it to fast: Essentially a DENIAL OF SERVICE attack (DOS). Again considered hacking (https://www.dr.dk/nyheder/indland/folketingets-hjemmeside-ramt-af-hacker-angreb).
- Add headers stating your name and email with your requests to ensure transparency.
- Be careful with copyrighted material.
- Fair use (don't take everything)
- If monetizing on the data, be careful not to be in direct competition with whom you are taking the data from.

11. DEC. 2015 KL. 13.13

Folketingets hjemmeside ramt af hacker-angreb

Forsøg på at komme ind på Folketingets hjemmeside resulterer i besked om, at siden ikke er tilgængelig.

Folketinget er blevet ramt af et hacker-angreb, bekræfter Finn Tørngren Sørensen, presseansvarlig i Folketinget, over for Avisen.dk.

Siden fredag formiddag har man fået beskeden "Denne webside er ikke tilgængelig", hvis man har forsøgt at komme ind på Folketingets hjemmeside, ft.dk.

- Det er rigtigt, at der er lukket for den eksterne adgang til Folketingets hjemmeside. Vi er under et såkaldt 'Denial of service"-angreb, og det har vi været siden klokken ti i formiddags. Det fungerer på den måde, at vi får så mange opkald til vores hjemmeside, at systemet bliver overbelastet. Derfor har vi måttet lukke ned for adgangen, siger han.

Folketinget har endnu ikke noget overblik over, hvem der står bag hacker-angrebet, eller hvornår hjemmesiden kan komme op at køre igen.

/ritzau/

Packages used

Today we will mainly build on the python skills you have gotten so far, and tomorrow we will look into more specialized packages.

- for connecting to the internet we use: **requests**
- for parsing: **beautifulsoup** and **regex**
- for automatic browsing / screen scraping (not covered in detail here): **selenium**
- for behaving responsibly we use: **time** and ***our minds***

We will write our scrapers with basic python, for larger projects consider looking into the packages **scrapy** or **pyspider**

In [36]:
```
import requests, re, time
from bs4 import BeautifulSoup
```

quick 101

To scrape information from the web is:

```
1. Finding URLs of the pages containing the information you want.
2. Fetching the pages via HTTP.
3. Extracting the information from HTML.
4. Finding more URL containing what you want, go back to 2.
```

Connecting to the internet HTTP

URL : the adressline in our browser.

Via HTTP we send a **get** request to an *address* with *instructions* (- or rather our dns service provider redirects our request to the right address)

Address: www.google.com

Instructions: /trends?query=social+data+science

Header: information send along with the request, including user agent (operating system, browser), cookies, and prefered encoding.

HTML: HyperTextMarkupLanguage the language of displaying web content. More on this tomorrow.

Now we are ready to get some data

3 basic examples

Lets get some action.

collecting data on display

static webpage example

visit the following website (https://www.basketball-reference.com/leagues/NBA_2018.html (https://www.basketball-reference.com/leagues/NBA_2018.html)).

The page displays tables of data that we want to collect. Tomorrow you will see how to parse such a table, but for now I want to show you a neat function that has already implemented this.

In [72]:
```
url = 'https://www.basketball-reference.com/leagues/NBA_2018.html' # link to the website
import pandas as pd
dfs = pd.read_html(url) # parses all tables found on the page.
```

In [78]:
```
EC_df = pd.read_html(url,attrs={'id':'confs_standings_E'}) # only parse the tables with
 attribute confs_standings_E
```

Out[78]:

[

	Eastern Conference	W	L	W/L%	GB	PS/G	PA/G	SRS
0	Toronto Raptors* (1)	59	23	0.720	—	111.7	103.9	7.29
1	Boston Celtics* (2)	55	27	0.671	4.0	104.0	100.4	3.23
2	Philadelphia 76ers* (3)	52	30	0.634	7.0	109.8	105.3	4.30
3	Cleveland Cavaliers* (4)	50	32	0.610	9.0	110.9	109.9	0.59
4	Indiana Pacers* (5)	48	34	0.585	11.0	105.6	104.2	1.18
5	Miami Heat* (6)	44	38	0.537	15.0	103.4	102.9	0.15
6	Milwaukee Bucks* (7)	44	38	0.537	15.0	106.5	106.8	-0.45
7	Washington Wizards* (8)	43	39	0.524	16.0	106.6	106.0	0.53
8	Detroit Pistons (9)	39	43	0.476	20.0	103.8	103.9	-0.26
9	Charlotte Hornets (10)	36	46	0.439	23.0	108.2	108.0	0.07
10	New York Knicks (11)	29	53	0.354	30.0	104.5	108.0	-3.53
11	Brooklyn Nets (12)	28	54	0.341	31.0	106.6	110.3	-3.67
12	Chicago Bulls (13)	27	55	0.329	32.0	102.9	110.0	-6.84
13	Orlando Magic (14)	25	57	0.305	34.0	103.4	108.2	-4.92
14	Atlanta Hawks (15)	24	58	0.293	35.0	103.4	108.8	-5.30]

collecting data behind the display

dynamic webpage example

Websites that continually show new data (jobsites, real-estate pages, social media), are as a rule dynamic webpages, where the whole page is not send as raw HTML. Instead a set of instructions (JavaScripts) on how to build it is send. Within those instructions we can often find direct calls to the data displayed.

Click on the following link: https://trends.google.com/trends/explore?date=all&geo=DK&q=H%C3%A5ndbold,Fodbold (https://trends.google.com/trends/explore?date=all&geo=DK&q=H%C3%A5ndbold,Fodbold)

Here we want to collect the data behind the graph. We open your browsers **>Network Monitor<** tool and search for the request that contains the data.

In [22]:
```
import requests
url = 'https://trends.google.com/trends/api/widgetdata/multiline?hl=da&tz=-120&req={"tim
e":"2004-01-01+2018-08-07","resolution":"MONTH","locale":"da","comparisonItem":[{"geo":
{"country":"DK"},"complexKeywordsRestriction":{"keyword":[{"type":"BROAD","value":"Håndb
old"}]}},{"geo":{"country":"DK"},"complexKeywordsRestriction":{"keyword":[{"type":"BROA
D","value":"Fodbold"}]}}],"requestOptions":{"property":"","backend":"IZG","category":0}}
&token=APP6_UEAAAAAW2s2Bpioky7lfJMWt4LDyhLInpC0jFzC&tz=-120'
response = requests.get(url)
response.ok
```

Out[22]:
```
True
```

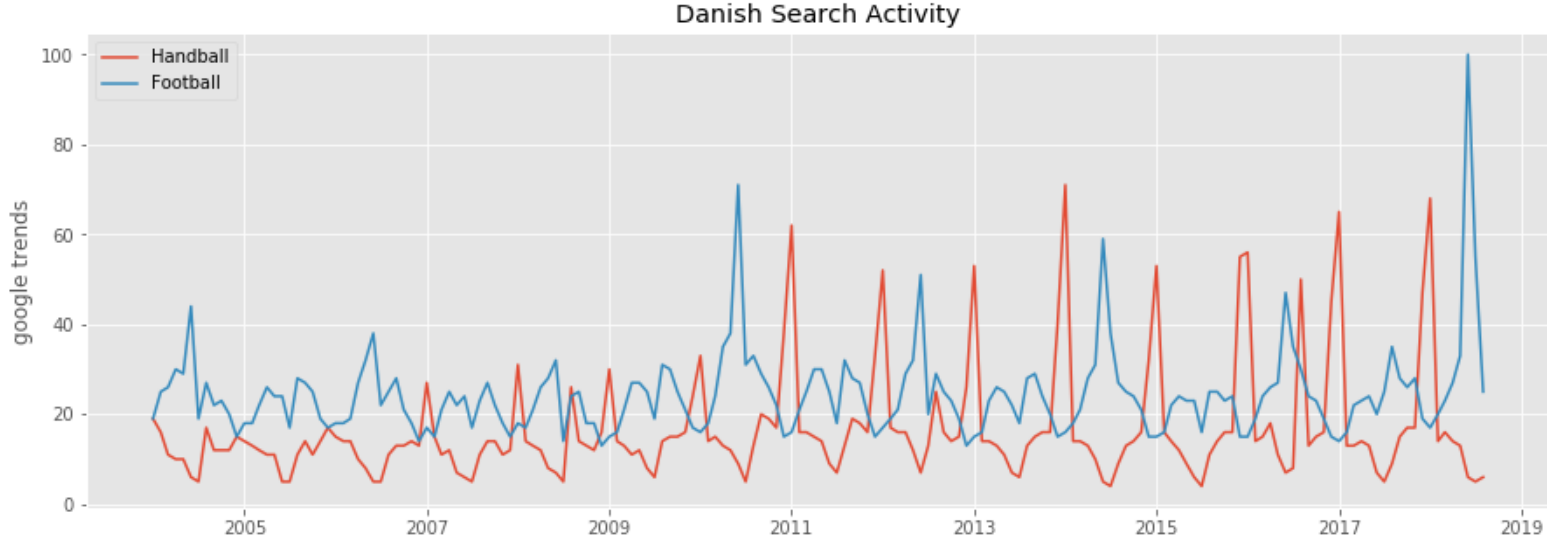
now we unpack and plot the data

In [80]:
```
d.keys()
```

Out[80]:
```
dict_keys(['market_cap_by_available_supply', 'price_btc', 'price_usd', 'volume_usd'])
```

In [33]:
```
import json # we use the json module to parse the json string.
import matplotlib.pyplot as plt
import datetime # datetime module is used to handle time information in python
%matplotlib inline
plt.style.use('ggplot')
data = json.loads(response.text.split(',',1)[1].strip())
t,y,y1 = zip(*[(i['time'],i['value'][0],i['value'][1]) for i in data['default']['timelin
eData']])
t = [datetime.datetime.fromtimestamp(int(i)) for i in t]
plt.figure(figsize=(15,5))
plt.plot(t,y,label='Handball')
plt.plot(t,y1,label='Football')
plt.legend()
plt.title('Danish Search Activity')
plt.ylabel('google trends')
```

Out[33]:
```
Text(0,0.5,'google trends')
```



In-class exercise 1

- Click on the following link (https://coinmarketcap.com/currencies/bitcoin/#charts (https://coinmarketcap.com/currencies/bitcoin/#charts))
- open the **>Network Monitor<** of your browser (refresh the page) and figure which request is collecting the data behind the chart.
- Collect the data using requests.
- Plot the "price_usd" data against time. Data comes as a nested dictionary.
  - First you need to unpack this.
  - Each datapoint is a unix timestamp (https://en.wikipedia.org/wiki/Unix_time) in miliseconds and a price.
  - To plot against time in python, you need to convert it to a datetime object using the `datetime` module.

```
t = datetime.datetime.fromtimestamp(unixtime_in_seconds)
```

In [53]:
```
# solution goes here
```

collecting unstructured data

Step 1. Finding the pages to collect. Often times we need to crawl a set of pages, from a website, this means finding all the links we need to collect. Here is a simple example of doing that.

Say we wanted to investigate the difference in how many hours of lectures and exercises the students on different universities and different study programs gets, or even if the ressources to this area are dropping. To answer this question we decide to scrape the Course description and the university webpages.

First we look at the courses on UCPH:

- *Click on this link: https://kurser.ku.dk (https://kurser.ku.dk)*
- *Navigate to a page where links to courses are displayed.*
- *Figure out a way to fetch those links. Here we look for the **"a"**-tag and the **"href"** attribute.*

Use your browsers **>Inspector<** to see the raw html that we want to parse.

In [81]:
```
# here I have found a list of courses at Anthropology UCPH
url = 'https://kurser.ku.dk/archive/2016-2017/STUDYBOARD_0010'
response = requests.get(url)
```

In [85]:
```
#response.text
```

After inspecting the html using our browsers Inspector tool, we can see that links occur after a href= pattern. Employing the python you already know we can use the `string.split` method to fetch the links.

In [118]:
```
# we split by the pattern 'href="' and
# skip the first element that was before the first occurrece of href
link_locations = response.text.split('href="')[1:]
###

links = [] #define container for the links
import random # good practice is to shuffle our data to inspect different data points ea
ch time.
# we do this with the random.sample function.
for link in random.sample(link_locations,len(link_locations)):
    #print(link)
    link = link.split('"')[0]
    links.append(link)
links[0:2]
```

Out[118]:
```
['/archive/2016-2017/course/AANB11002U', 'https://jobportal.ku.dk/']
```

In [122]:
```
# links are relative to the domain: https://kurser.ku.dk/
links = ['https://kurser.ku.dk'+ i for i in links]
print(len(links))
# only links with /archive/ in the name is a relevant course
links = [link for link in links if '/archive/' in link]
print(len(links))
```

```
142
50
```

Building URLS using a recognizable pattern.

A nice trick is to understand how urls are constructed to communicate with a server.

Lets look at how jobindex.dk (https://www.jobindex.dk/) does it. We simply click around and take note at how the addressline changes.

This will allow us to navigate the page, without having to parse information from the html or click any buttons.

- / is like folders on your computer.
- ? entails the start of a query with parameters
- = defines a variable: e.g. page=1000 or offset = 100 or showNumber=20
- & separates different parameters.
- + is html for whitespace

Good practices

- Transparency: send your email and name in the header so webmasters will know you are not a malicious actor.
- Ratelimiting: Make sure you don't hit their servers to hard.
- Reliability:
  - Make sure the scraper can handle exceptions (e.g. bad connection) without crashing.
  - Keep a log.
  - Store raw data.

In [5]:
```
# Transparent scraping
import requests
#response = requests.get('https://www.google.com')
session = requests.session()
session.headers['email'] = 'youremail'
session.headers['name'] = 'name'
#session.headers['User-Agent'] = '' # sometimes you need to pose as another agent...
session.headers
```

Out[5]:
```
{'User-Agent': 'python-requests/2.18.4', 'Accept-Encoding': 'gzip, deflate', 'Accep
t': '*/*', 'Connection': 'keep-alive', 'email': 'youremail', 'name': 'name'}
```

A quick tip is that you can change the user agent to a cellphone to obtain more simple formatting of the html.

Reliability!

When using found data, you are the curator and you are **responsible** for enscribing **trust** in the datacompilation.

Reliability is ensured by an interative process, of inspection, error detection and error handling.

Build your scrape around making this process easy by:

- logging information about the collection (e.g. servertime, size of response to plot weird behavior, size of response over time, number of calls pr day, detection of holes in your data).
- Storing raw data (before parsing it) to be able to backtrack problems, without having to wait for the error to come up.

In [4]:
```
# Control the pace of your calls
import time
def ratelimit():
    "A function that handles the rate of your calls."
    time.sleep(1) # sleep one second.
# Reliable requests
def get(url,iterations=10,check_function=lambda x: x.ok):
    """This module ensures that your script does not crash from connection errors.
        that you limit the rate of your calls
        that you have some reliability check
        iterations : Define number of iterations before giving up.
        exceptions: Define which exceptions you accept, default is all.
    """
    for iteration in range(iterations):
        try:
            # add ratelimit function call here
            ratelimit() # !!
            response = session.get(url)
            if check_function(response):
                return response # if succesful it will end the iterations here
        except exceptions as e: #  find exceptions in the request library requests.excep
tions
            print(e) # print or log the exception message.
    return None # your code will purposely crash if you don't create a check function la
ter.
```

Logs

Example of a logging function

```
done = set()# define a container for the links you have already collected
count = 0
def log_function(url,response,error_check=lambda x: x.ok,separator=','):
    global last_t
    if os.path.isfile(logfilepath)
        f_log = open(logfilepath,'w') # define logfile, remember not to overwrite it.
        # write columns to be used, basic ones are, servertime, deltaT since last call,
 url, success of request,
        header = ['serverTime','deltaT','url','success','length','path']
        f_log.write(','.join(header)+'\n')
    else:
        f_log = open(logfilepath,'a')
    #### Update timing info ####
    t = time.time()
    delta_t = t-last_t # calculate time since last call
    last_t = t# update last call time
    #### meta data ###
    success = error_check(response)
    if success: # if call is successfull we add it to the done container
        done.add(url)
    if response.ok:
        length = len(response.text)
    else:
        length = 0
    row = [t,delta_t,url,success,length,path]
    f_log.write(separator.join(map(str,row))+'\n')
```