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.

Scraping skills allow you to:

- singlehandedly create powerful datasets.
- Pose new questions that was very expensive or impossible with existing data sources.

Agenda

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

- The basics of webscraping
    - Connecting, Crawling, Parsing,Storing, Logging.
- Hands-on examples.
- 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. Combine them to reach new potentials.
- 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.

WITH GREAT POWER
COMES GREAT RESPONSIBILITY
imgflip.com

</a>

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.

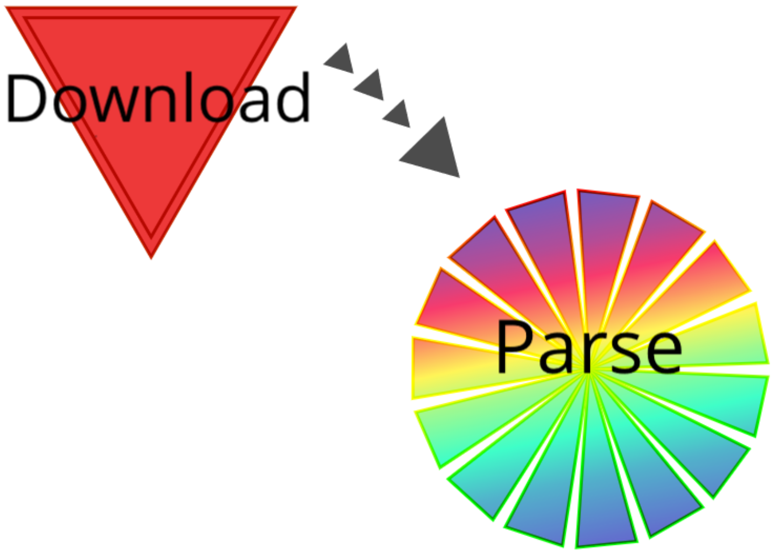
- Det er rigtigt, at der er lukket for den eksterne adgang til Folketingets hjemmeside. Vi er under et såkaldt 'Denial of service"-angreb, og det har vi været siden klokken ti i formiddags. Det fungerer på den måde, at vi får så mange opkald til vores hjemmeside, at systemet bliver overbelastet. Derfor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

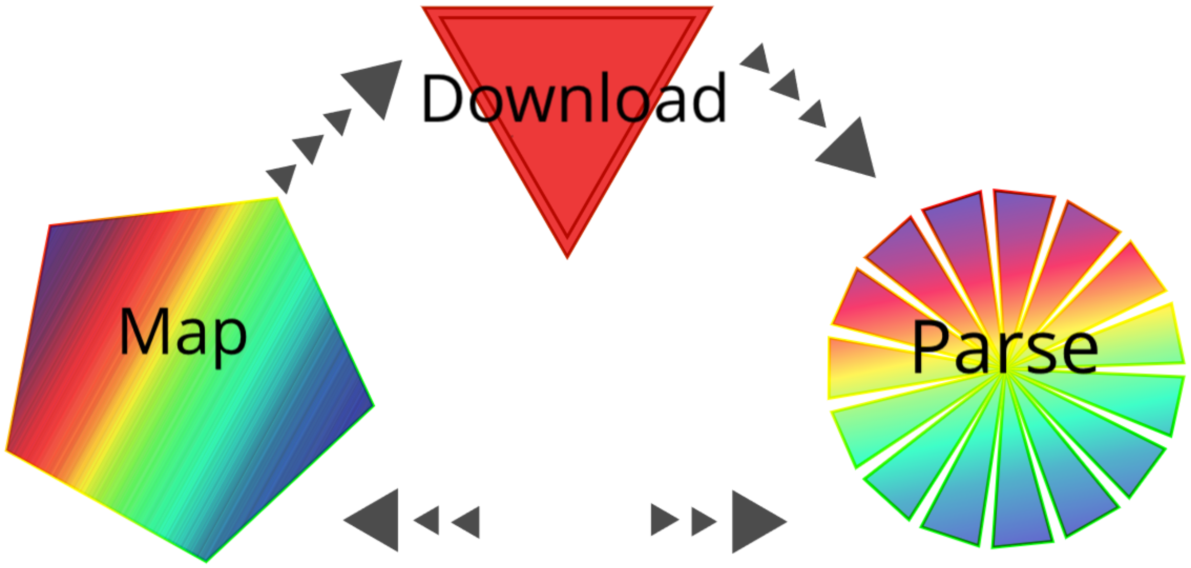
/ritzau/

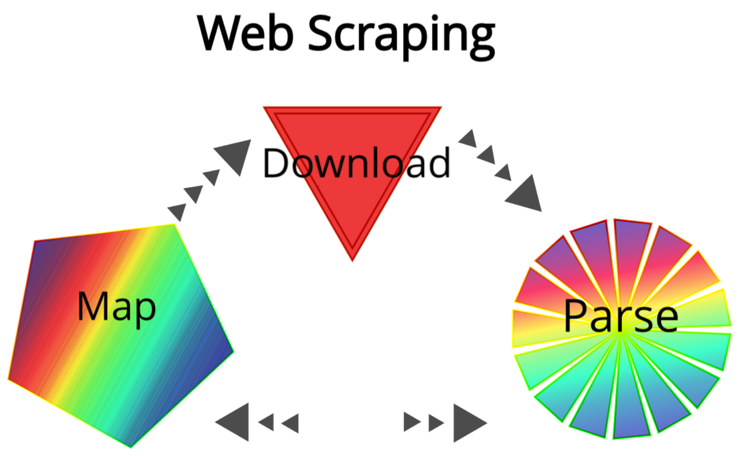
quick 101

Web Scraping



Web Scraping





To scrape information from the web is:

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

Crawling is when your collection spans unknown domains and is generally much harder.

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

Connecting to the Internet

```
In [ ]: import requests
        response = requests.get('https://www.google.com')
        response.text
```

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 / Domain: 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.

Reliability and data quality issues: A first primer

- You are in charge of the data quality!
- **Everything** can go wrong.
    - **Program** errors.
    - Connection errors
    - Domain wants to avoid scrapers and change the behavior of the site.
    - Parsing errors.

You will be graded based on your ability to document your data collection and your **critical** and **informed** inspection of data quality based on logs.

In [6]:
```
## SOLUTION

import requests,os,time
def ratelimit():
    "A function that handles the rate of your calls."
    time.sleep(1) # sleep one second.

class Connector():
  def __init__(self,logfile,overwrite_log=False,connector_type='requests',session=False,
path2selenium='',n_tries = 5,timeout=30):
    """This Class implements a method for reliable connection to the internet and monito
ring.
    It handles simple errors due to connection problems, and logs a range of information
for basic quality assessments

    Keyword arguments:
    logfile -- path to the logfile
    overwrite_log -- bool, defining if logfile should be cleared (rarely the case).
    connector_type -- use the 'requests' module or the 'selenium'. Will have different s
ince the selenium webdriver does not have a similar response object when using the get m
ethod, and monitoring the behavior cannot be automated in the same way.
    session -- requests.session object. For defining custom headers and proxies.
    path2selenium -- str, sets the path to the geckodriver needed when using selenium.
    n_tries -- int, defines the number of retries the *get* method will try to avoid ran
dom connection errors.
    timeout -- int, seconds the get request will wait for the server to respond, again t
o avoid connection errors.
    """

    ## Initialization function defining parameters.
    self.n_tries = n_tries # For avoiding triviel error e.g. connection errors, this def
ines how many times it will retry.
    self.timeout = timeout # Defining the maximum time to wait for a server to response.
    ## not implemented here, if you use selenium.
    if connector_type=='selenium':
      assert path2selenium!='', "You need to specify the path to you geckodriver if you
```

```
 want to use Selenium"
      from selenium import webdriver
      ## HIN download the latest geckodriver here: https://github.com/mozilla/geckodrive
r/releases

      assert os.path.isfile(path2selenium),'You need to insert a valid path2selenium the
path to your geckodriver. You can download the latest geckodriver here: https://github.c
om/mozilla/geckodriver/releases'
      self.browser = webdriver.Firefox(executable_path=path2selenium) # start the browse
r with a path to the geckodriver.

    self.connector_type = connector_type # set the connector_type

    if session: # set the custom session
      self.session = session
    else:
      self.session = requests.session()
    self.logfilename = logfile # set the logfile path
    ## define header for the logfile
    header = ['id','project','connector_type','t', 'delta_t', 'url', 'redirect_url','res
ponse_size', 'response_code','success','error']
    if os.path.isfile(logfile):
      if overwrite_log==True:
        self.log = open(logfile,'w')
        self.log.write(';'.join(header))
      else:
        self.log = open(logfile,'a')
    else:
      self.log = open(logfile,'w')
      self.log.write(';'.join(header))
    ## load log
    with open(logfile,'r') as f: # open file

      l = f.read().split('\n') # read and split file by newlines.
      ## set id
      if len(l)<=1:
        self.id = 0
      else:
```

```
    self.id = int(l[-1][0])+1

 def get(self,url,project_name):
  """Method for connector reliably to the internet, with multiple tries and simple error handling, as well as default logging function.
  Input url and the project name for the log (i.e. is it part of mapping the domain, or is it the part of the final stage in the data collection).

  Keyword arguments:
  url -- str, url
  project_name -- str, Name used for analyzing the log. Use case could be the 'Mapping of domain','Meta_data_collection','main data collection'.
  """

  project_name = project_name.replace(';','-') # make sure the default csv seperator is not in the project_name.
  if self.connector_type=='requests': # Determine connector method.
   for _ in range(self.n_tries): # for loop defining number of retries with the requests method.
    ratelimit()
    t = time.time()
    try: # error handling
     response = self.session.get(url,timeout = self.timeout) # make get call

     err = '' # define python error variable as empty assumming success.
     success = True # define success variable
     redirect_url = response.url # log current url, after potential redirects
     dt = t - time.time() # define delta-time waiting for the server and downloading content.
     size = len(response.text) # define variable for size of html content of the response.
     response_code = response.status_code # log status code.
     ## log...
     call_id = self.id # get current unique identifier for the call
     self.id+=1 # increment call id
     #['id','project_name','connector_type','t', 'delta_t', 'url', 'redirect_url','response_size', 'response_code','success','error']
     row = [call_id,project_name,self.connector_type,t,dt,url,redirect_url,size,res
```

```
ponse_code,success,err] # define row to be written in the log.
          self.log.write('\n'+';'.join(map(str,row))) # write log.
          return response,call_id # return response and unique identifier.

        except Exception as e: # define error condition
          err = str(e) # python error
          response_code = '' # blank response code
          success = False # call success = False
          size = 0 # content is empty.
          redirect_url = '' # redirect url empty
          dt = t - time.time() # define delta t

          ## log...
          call_id = self.id # define unique identifier
          self.id+=1 # increment call_id

          row = [call_id,project_name,self.connector_type,t,dt,url,redirect_url,size,res
ponse_code,success,err] # define row
          self.log.write('\n'+';'.join(map(str,row))) # write row to log.
    else:
      t = time.time()
      ratelimit()
      self.browser.get(url) # use selenium get method
      ## log
      call_id = self.id # define unique identifier for the call.
      self.id+=1 # increment the call_id
      err = '' # blank error message
      success = '' # success blank
      redirect_url = self.browser.current_url # redirect url.
      dt = t - time.time() # get time for get method ... NOTE: not necessarily the compl
ete load time.
      size = len(self.browser.page_source) # get size of content ... NOTE: not necessari
ly correct, since selenium works in the background, and could still be loading.
      response_code = '' # empty response code.
      row = [call_id,project_name,self.connector_type,t,dt,url,redirect_url,size,respons
e_code,success,err] # define row
      self.log.write('\n'+';'.join(map(str,row))) # write row to log file.
    # Using selenium it will not return a response object, instead you should call the b
```

```
rowser object of the connector.
    ## connector.browser.page_source will give you the html.
      return call_id
```

In [5]:

```
### Here we define our connection to the internet
connector = Connector('logfile_sds_lecture8.csv')
## Test the get method. It takes a url, and name for the log. In this case test call.
## Ideally you should name it after the project, and subpart of that project.
## E.g. mapping_jobposting referring to the initial mapping of the domain.
response,call_id = connector.get('https://www.google.com','test_call')
```

Now we are ready to get some data

3 basic examples

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]:

```python
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. Open your browsers **>Network Monitor<** tool and search for the request that contains the data.

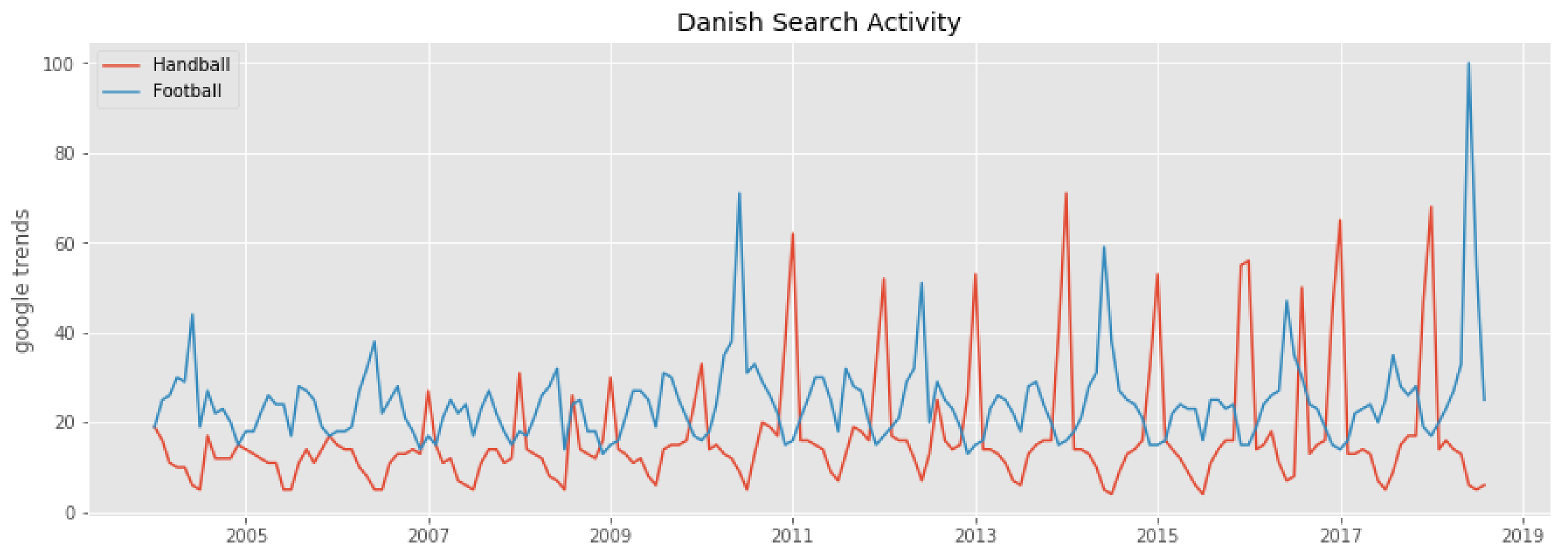
```
In [10]: import requests
         url = 'https://trends.google.com/trends/api/widgetdata/multiline?hl=da&tz=-120&req={"tim
         e":"2004-01-01+2018-08-07","resolution":"MONTH","locale":"da","comparisonItem":[{"geo":
         {"country":"DK"},"complexKeywordsRestriction":{"keyword":[{"type":"BROAD","value":"Håndb
         old"}]}},{"geo":{"country":"DK"},"complexKeywordsRestriction":{"keyword":[{"type":"BROA
         D","value":"Fodbold"}]}}],"requestOptions":{"property":"","backend":"IZG","category":0}}
         &token=APP6_UEAAAAW2s2Bpioky7lfJMWt4LDyhLInpC0jFzC&tz=-120'
         response,call_id = connector.get(url,'google_trends')
         response.ok
```

```
Out[10]: False
```

now we unpack and plot the data

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
```

Navigating websites to collect links

Now I will show you a few common ways of finding the links to the pages you want to scrape.

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

collecting unstructured data

Step 1. **Mapping** - 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 - if the ressources to exercises have been cut over the years, and whether this might affect dropout. To answer this question we decide to scrape the Course description of 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]:
```python
# here I have found a list of courses at Anthropology UCPH
url = 'https://kurser.ku.dk/archive/2016-2017/STUDYBOARD_0010'
response = connector.get(url,'course_mapping')
```

```
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
```

```
In [139]: print(random.sample(links,5))

['https://kurser.ku.dk/archive/2016-2017/course/AANB11075U', 'https://kurser.ku.dk/ar
chive/2016-2017/course/AANK16102U', 'https://kurser.ku.dk/archive/2016-2017/course/AA
NB05070U', 'https://kurser.ku.dk/archive/2016-2017/course/AANB05023U', 'https://kurse
r.ku.dk/archive/2016-2017/course/AANB11046U']
```

In-class exercise 2 Now it your turn to practice collecting links using the simple split method.

The above example only collected links to courses in Anthropology. Now I want you to build a script that

- first collects links to all the different studyboards here: https://kurser.ku.dk/archive/2016-2017 (https://kurser.ku.dk/archive/2016-2017)
- And next run through those to collect the links to all the courses at UCPH 2016-2017.
- figure out how to get links from the other years (hint look at the urls).

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

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 (many + randomly sampled), 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.