

# BCCP Web Scraping Course

---

# Day 1

---

# Table of Contents

## Day 1

Very short intro to Python

Intro to Webscraping

APIs

## Day 2

HTML parsing

Text pattern matching

## Day 3

Browser automation

Own script

# Day 1

---

**Very short intro to Python**

## Your experiences

- Which tools or programming languages do you use when working with data?
- Have you used Python before?

# Why Python for Webscraping?

- Common web data structures are similar to data structures in Python.
- Many Python packages for webscraping and APIs can be found.

# Python interpreter — interactive mode

- input prompt `>>>`
- comments `#`
- operators `+`, `-`, `*` and `/`

```
>>> 2 + 2
4
>>> 8 / 5  # division always returns a floating point number
1.6
>>> 5 ** 2  # 5 squared
25
```

# Python data structures

- Lists ( value )

```
squares = [1, 4, 9, 16, 25]
```

- Dictionaries ( key: value )

```
followers = {'kevin': 15, 'julian': 9}
```



## Python data structures — Lists

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
>>> squares.append((len(squares) + 1) ** 2)  # using append() method
>>> squares
[1, 4, 9, 16, 25, 36]
```

# Python data structures — Dictionaries

- Dictionaries ( `key: value` )
- Unlike lists, dictionaries are indexed by keys not by positions.

```
>>> followers = {'kevin': 15, 'julian': 9}
>>> followers['kevin']
15
>>> followers['kevin'] = 16
```

## Common web data structures

For example, JSON nearly identical to combination of Python's dictionaries and lists.

```
[
  {
    "id": "1290837412912998347",
    "followers": "15",
    "name": "Kevin"
  },
  {
    "id": "1290837412973490803",
    "followers": "9",
    "name": "Julian"
  }
]
```

# Functions

- `def` defines a function.
- Followed by function name with parenthesized sequence of parameters.
- Body of function must be indented.

```
>>> def list_append(a, mylist=[]):  
...     """Example documentation string:  
...     Append value to list. mylist defaults to empty list."""  
...     mylist.append(a)  
...     return mylist
```

```
>>> list_append(97, [99, 98])  
[99, 98, 97]
```

```
>>> list_append(1)  
[1]
```

# Modules

- Definitions (functions and variables) can be saved in **modules**. Our example function `list_append()` can be saved in a file `list_operations.py`.
- Such modules can be imported into the interpreter, scripts, or other modules.

```
import list_operations
list_operations.list_append(1)

from list_operations import list_append
list_append(1)

import list_operations as lo
lo.list_append(1)
```

- Bad practice: `from sound.effects import *`

# Packages

- Packages structure modules namespace by using dotted module names. `A.B` designates a submodule named `B` in a package named `A`.

```
import matplotlib.pyplot as plt
```

- Packages can be installed with pip.

```
pip install matplotlib
```

- A common convention is to have a list of packages in requirements.txt:

```
pip install -r requirements.txt
```

# Files

- Reading and writing files

```
f = open('workfile', 'r') # read-only
f = open('workfile', 'w') # write-only
f = open('workfile', 'a') # appending

f = open('workfile') # read-only, as mode defaults to 'r'
```

- Closing files

```
f.close() # Manually close a file

with open('workfile') as f: # with closes file "automatically"
    read_data = f.read()
```

- Several packages offer file operations (e.g. `pandas.read_csv()`)

## Further reading

- [Official Python Docs Tutorial](#)
- [W3Schools Python Tutorial](#)
- [Cookiecutter Data Science Project Template](#)



# Day 1

---

## Intro to Webscraping

# Introduction to Webscraping

- Basic idea: Turn information on website to structured data
- Typical workflow:
  1. Look at website to decide best approach
    - Is an Application Programming Interface (API) available?
    - Do the HTML elements have fixed names?
    - Does the page load statically or dynamically?
  2. Download information from URL
  3. Turn information into structured data and save

## Some concepts

- APIs
- HTML parsing vs text matching
- Static vs dynamic websites

- If available, a convenient way to get pre-structured data (usually JSON or XML).
- Example: OpenStreetMap (OSM) (<https://www.openstreetmap.org>)
  - When searching manually, results can be shown as XML. Automating the search on OpenStreetMap and clicking on the relevant links would therefore be a way to save this data.
  - However, OSM offers several APIs that simplify this task. One API is the Nominatim API (<https://nominatim.openstreetmap.org>).

## API example: Nominatim API for OSM

- See <https://nominatim.org/release-docs/develop/api/Search/> for documentation on search syntax
- Search for 'diw berlin' and return as JSON: <https://nominatim.openstreetmap.org/search?q=diw+berlin&format=json>

- Use structure of HTML code to find needed information.
- Works best if the code is well-structured and element names are fixed.

## HTML parsing example: eBay search results

- Look at results for 'star wars blu ray' on eBay:  
`https://www.ebay.de/sch/i.html?\_nkw=star+wars+blu+ray`
- Most browsers have a feature to look at source code (e.g. in Chrome, you can right click on any website element and click on 'Inspect').
- On eBay, the HTML tags containing certain content always have the same name, this simplifies HTML parsing.
- For example, the tag `<div id="ResultSetItems">` contains all results. Inside this tag, the individual listings are saved in tags called `<li class="sresult">`. In Chrome, you can also look for elements using the XPATH syntax (e.g. for the individual listings: `//li[contains(@class,'sresult')]`). More information on XPATH here: `https://www.w3schools.com/xml/xpath\_syntax.asp`

## Text pattern matching

- If the HTML code is not well-structured or names change, text pattern matching is an alternative.
- Idea: Take text from (parts of) a page and find needed information by matching a regular expression



## Example of website without clear HTML tag names: Airbnb

- Search for homes in Berlin-Mitte: `https://www.airbnb.de/s/Berlin-Mitte--Berlin/homes?query=Berlin-Mitte%2C%20Berlin`
- Say you wanted to get the number of results for this search. The element does not have a clear name. Using HTML parsing is still possible but is prone to errors. Instead, one could match on a regular expression.

## Static vs dynamic websites

- On static websites, the entire content is loaded immediately. E.g. eBay:  
`https://www.ebay.de/sch/i.html?\_nkw=star+wars+blu+ray`
- On dynamic websites, content may not load instantaneously or only after user action, making them usually more complicated to scrape. E.g. Airbnb:  
`https://www.airbnb.de/s/Berlin-Mitte--Berlin/homes?query=Berlin-Mitte%2C%20Berlin` (Try disabling JavaScript in your browser and reloading the page).
- Getting the complete source code from a dynamic website can be done with browser automation. The idea is to open a website in an actual browser (and interacting with it if necessary) and save the source code of the content from there.

## Important Python packages

- `requests`: To load URL and recover source code (for static web pages)
- `beautifulsoup4`: To turn HTML code to navigable Python object
- `selenium`: For browser automation
- `pandas`: To create DataFrames

# Day 1

---

APIs

## Why APIs?

- Data owners want know who is using their services.
- Data owners want to limit requests.
- Data owners want to supply data in their preferred format.

- “Conduct historical research and search from Twitter’s massive archive of publicly-available Tweets posted since March 2006?”
- “Listen in real-time for Tweets of interest?”

Source: <https://developer.twitter.com/en/docs/basics/getting-started.html>

## Twitter API — Limits

All request windows are 15 minutes in length.

Endpoint	Resource family	Requests / window (user auth)
GET followers/list	followers	15
GET lists/members	lists	900
GET lists/statuses	lists	900
GET search/tweets	search	180
GET statuses/lookup	statuses	900
GET statuses/retweeters/ids	statuses	75
GET statuses/user_timeline	statuses	900
GET users/lookup	users	900

Source: <https://developer.twitter.com/en/docs/basics/rate-limits>

## Tweepy package

We use the Tweepy package to access twitter's RESTful API.

```
user = api.get_user('twitter')

# tweepy models contain the data plus and some methods.
print(user.screen_name)
print(user.followers_count)
for friend in user.friends():
    print(friend.screen_name)
```



## Twitter API — JSON Example

Packages usually also allow to access the JSON directly, which often contains more information than provided by the API.

```
import tweepy
from twitter_auth import auth

def get_tweets(api, screen_name):
    tweets_json = [status._json for status in tweepy.Cursor(
        api.user_timeline,
        screen_name=screen_name,
        tweet_mode='extended'
    ).items(2)]
    return tweets_json

api = tweepy.API(auth)
tweets = get_tweets(api, '@guardian')
```

# Twitter API — JSON Example

```
{
  'contributors': None,
  'coordinates': None,
  'created_at': 'Tue Nov 20 17:56:53 +0000 2018',
  'display_text_range': [0, 97],
  'entities': {
    'hashtags': [],
    'symbols': [],
    'urls': [
      {
        'display_url': 'trib.al/hDWAwvZ',
        'expanded_url': 'https://trib.al/hDWAwvZ',
        'indices': [74, 97],
        'url': 'https://t.co/GpWbVaZV3F'
      }
    ],
    'user_mentions': []
  },
  'favorite_count': 17,
  'favorited': False,
  'full_text': 'I was arrested at a climate change protest - it was worth it | '
    'Gavin Turk https://t.co/GpWbVaZV3F',
  'geo': None,
  'id': 1064940660942352385,
  'id_str': '1064940660942352385',
  'in_reply_to_screen_name': None,
  'in_reply_to_status_id': None,
  'in_reply_to_status_id_str': None,
  'in_reply_to_user_id': None,
  'in_reply_to_user_id_str': None,
  'is_quote_status': False,
  'lang': 'en',
  'place': None,
  'possibly_sensitive': False
}
```

# World Bank API

- World Bank APIs provide access to:
  - Indicators API
  - Data Catalog API
  - Projects API
  - Finances API
  - Climate Data API
- Access data without authentication.
- Worldbank API documentation
- `world_bank_data` package documentation

## world\_bank\_data — Example

```
import world_bank_data as wb

# Get estimates for the world population:
wb.get_series('SP.POP.TOTL', date='2017')

# Get timeseries of "Agricultural machinery, tractors" in Albania
wb.get_series('AG.AGR.TRAC.NO', country='ALB')
```

## There might be APIs without a working package

- Check more general packages. For example,  
<https://pandas-datareader.readthedocs.io/en/latest/readers/>
- Write your own functions.

- Most APIs are RESTful APIs (Representational State Transfer)
- RESTful APIs use HTTP methods:
  - GET — fetch available information
  - POST — create item
  - DELETE — delete item
  - PUT — modify an existing item

## RESTful API — Example

For web scraping we only need GET.

```
import requests

resp = requests.get('https://example.com/items/')
for item in resp.json():
    print(item['id'])
```

## Day 2

---



# Table of Contents

## Day 1

Very short intro to Python

Intro to Webscraping

APIs

## Day 2

HTML parsing

Text pattern matching

## Day 3

Browser automation

Own script

## Day 2

---

HTML parsing

- After obtaining the HTML source code, how to obtain the information required?
- If the HTML code is well-structured and its tags have (more or less) unique names, we can navigate the HTML elements to get the information we want.
- The `beautifulsoup4` package converts the HTML code into a Python object that can be navigated using properties and functions.

## Some HTML terms

- Consider

```
<a href="http://www.bccp-berlin.de" target="_blank">BCCP</a>
```
- HTML Elements
  - The entire thing is an HTML element. Specifically, it is a link leading to the BCCP website and displayed as "BCCP".
  - HTML elements usually consist of a start tag and an end tag.
- HTML Tags
  - The start tag of the element above is `<a>` and the end tag is the corresponding `</a>`
  - Start tag can and sometimes must contain attributes.
- HTML Attributes
  - The `<a>` tag contains the attribute `href` and `target`. `href` specifies the destination to which the link should lead and `target="_blank"` specifies that the link should be opened in a new window.
  - For web scraping purposes, the attributes `class` and `id` are usually useful as these are often used to identify certain (groups of) elements.

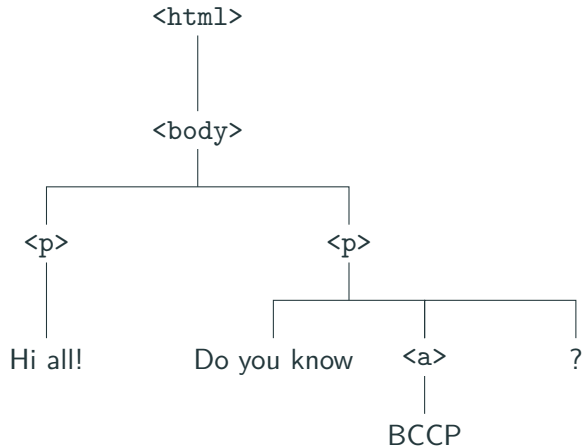
## Basic HTML documents structure

- HTML documents have a tree-like/nested structure
- Elements can contain various levels of sub-elements that in the end contain some content

## HTML document example

```
<html>
<body>
<p>
Hi all!
</p>
<p>
Do you know
<a href="http://www.bccp-berlin.de" target="_blank">BCCP</a>?
</p>
</body>
</html>
```

# Tree structure



## General steps

1. Load a web page and get the source code: Use `requests` for static and `selenium` for dynamic websites
2. Convert (“parse”) the source code into a soup object: Use `beautifulsoup4`
3. Navigate/search the soup object to get the information you want



## Example for today

- Let's scrape the details of all upcoming BCCP events:  
`http://www.bccp-berlin.de/events/all-events/`
- Steps:
  1. Analyze HTML structure
  2. Load source code
  3. Save information on events available on the front page
  4. Loop through individual event pages to get details
  5. Combine to DataFrame

## Analyzing the HTML structure i

- Open <http://www.bccp-berlin.de/events/all-events/> in a browser and inspect the source code
- Information on events saved in div elements

```
<div class="eventList">
```

```
...
```

```
<div class="event-list-item event-type1">...</div>
```

```
...
```

```
<div class="event-list-item event-type2">...</div>
```

```
...
```

```
</div>
```

## Analyzing the HTML structure ii

- Details are saved in sub-elements in each  
//div[contains(@class,'event-list-item')] element

```
<div class="event-list-item event-type1">  
  <div class="top-bar">  
    <span class="date single">June 27, 2019</span>  
    <span class="b-events__item__type">Seminar</span>  
  </div>  
  <div class="b-events__item__inner">  
    <div class="content">  
      <div class="genres">Berlin Behavioral Economics Seminar</div>  
      <h2 class="eventHeader">
```

## Analyzing the HTML structure iii

```
<a href="/events/all-events/events-detail/
felix-holzmeister-university-of-innsbruck/">
  Felix Holzmeister (University of Innsbruck)
</a>
</h2>
<div class="teaser">Delegated Decision Making in Finance</div>
<div class="location">
  <strong class="label">Location</strong>
  <div class="address">
    <span class="name">WZB</span>
    <span class="address">Reichpietschufer 50, Room B001</span>
    <span class="zip">10785</span>
```

## Analyzing the HTML structure iv

```
<span class="place">Berlin</span>
</div>
</div>
<div class="time">
  <strong class="label">Time</strong>
  <span>16:45-18:00</span>
</div>
</div>
<div class="button detail">
  <a title="Felix Holzmeister (University of Innsbruck)"
  href="/events/all-events/events-detail/
  felix-holzmeister-university-of-innsbruck/">
```

```
    Event Details  
  </a>  
</div>  
</div>  
</div>
```

## Getting the data

- Idea: Loop through listings, save details, visit details page to load more info
- See “htmlparsing.ipynb”.

## From website to Python soup

1. requests: Load website and save source code as string
2. BeautifulSoup: Take source string and parse to get soup object
  - There are three different parsers: `html.parser`, `lxml`, `html5lib`
  - Differences are discussed here: <https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser>
  - I usually use `lxml`
3. Soup object includes functions and attributes that facilitate searching and navigating HTML elements



## Some BeautifulSoup functions

- Look at the very good documentation:  
<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- You can either *search* the document:
  - `.find_all()`: Find all elements that match a certain condition. Returns a list.
  - `.find()`: Same as `find_all()` but only returns first match.
- If unique tag names are not available, *navigation* of the HTML tree rather than searching it is possible, e.g.:
  - Vertically: `.parent`, `.parents`, `.children`
  - Horizontally: `.next_sibling`, `.previous_sibling`

## Day 2

---

**Text pattern matching**

## Text pattern matching

- Regular expressions (regex) are rules for the set of possible strings to match.
- Use regex to search and extract substrings.
- Set might contain words, whole sentences, or e-mail addresses, or anything you like.
- Are there matches for the pattern anywhere in the string?

# Regular Expression Syntax

- `.` matches any character except a newline.
- `^` matches the start of the string.
- `$` matches the end of the string.
- `*` match 0 or more repetitions of the preceding RE.
- `+` match 1 or more repetitions of the preceding RE.
- `?` match 0 or 1 repetitions of the preceding RE.

## Text pattern matching — Example

```
import re

# Find all href tags with http(s) link
links_href = re.findall('href="http[s]?://.*?"', html)

# Find all href tags with http(s) link and return links only
links_only = re.findall('href="(http[s]?://.*?)"', html)

# Find all href tags with http(s) link and split
links_parts = re.findall('href="(http[s]?):/(.*?)"', html)
[x[1] for x in links_parts]

# Find all numbers
re.findall('\\d+\\.?\\d?', html)
```

## Day 3

---

# Table of Contents

## Day 1

Very short intro to Python

Intro to Webscraping

APIs

## Day 2

HTML parsing

Text pattern matching

## Day 3

Browser automation

Own script

# Day 3

---

## Browser automation



## Why browser automation?

- If the content of a page is loaded dynamically (e.g. with JavaScript), using requests could yield an “empty” source code.
- Browser automation is then a way to load the page in an actual browser and let the JavaScript load as if you actually visited the page.
- Because this uses an actual browser and a browser driver, this approach is less stable and crashes can occur. Further, loading a page in a browser usually takes more time than loading it in requests.

## Example for today

- Let us scrape all future events from the BERA website:  
`https://www.berlin-econ.de/events.`
- In order to load all events, we need to click on the bottom buttons to navigate through the results pages.
- However, these buttons do not link to a new URL but load content using JavaScript:

```
<a href="javascript:;" class="item" data-request-success="scroll(0,0)"
data-request="onEventSearch"
data-request-update="'@events-list': '#event-results'"
data-request-data="page:2">Next →</a>
```

## Some technical notes

- We will use the `selenium` package
  - It allows you to control a browser from a Python script
  - The documentation can be found here:  
`https://selenium-python.readthedocs.io/`
- Besides `selenium`, you need to have an actual browser installed that you are going to use and a compatible browser driver that `selenium` can use to control the browser
  - We will use Google's Chrome browser (`https://www.google.com/chrome/`) and the corresponding `ChromeDriver` (`http://chromedriver.chromium.org/`). Some parts of the code might have a different syntax for different browsers.
  - `selenium`'s documentation includes links to drivers for four popular browsers:  
`https://selenium-python.readthedocs.io/installation.html#drivers`
  - The documentation for the various browser driver types in `selenium` can be found here: `https://seleniumhq.github.io/selenium/docs/api/py/api.html`
  - Make sure that the driver version fits your installed browser version

## First, analyze the HTML code of <https://www.berlin-econ.de/events>

- Events are saved in a `<div class='event-results'>` element
- Inside this, events for different days are separated by a `<div class='event-date-separator'>` element
- The actual events are then saved in a `<div class='ui segments'>` elements, more specifically, in `<div class='ui segment'>` elements
- The buttons to navigate to the next results pages are saved in the last element in `<div class='event-results'>` (`<div class='ui pagination menu'>`)
- Need a mix of navigating and searching the HTML document

# Approach

1. Load events page in browser
2. Loop through elements in `<div class='event-results'>`
  - 2.1 If it is a date, save the date
  - 2.2 If it is an event, save the event details
  - 2.3 If it is the buttons, click the button for the next page, if available.
  - 2.4 Repeat until no other next page available
3. Turn to DataFrame and save
  - See `automation.ipynb`

## Interacting with the webpage

- In order to be able to click the button, we need to scroll it into view first
- For this, we need to tell selenium where the wanted element is and have it scroll there
- This can be done e.g. using XPATH syntax
- Typical steps are therefore:
  1. Find the element in the source code (e.g.  
`element = driver.find_element_by_xpath(xpath)`, other alternatives here:  
<https://selenium-python.readthedocs.io/locating-elements.html>)
  2. Scroll it into view and click, e.g.  
`ActionChains(driver).move_to_element(element).click(element).perform()`
- See [https://seleniumhq.github.io/selenium/docs/api/py/webdriver/selenium.webdriver.common.action\\_chains.html](https://seleniumhq.github.io/selenium/docs/api/py/webdriver/selenium.webdriver.common.action_chains.html) for documentation on ActionChains and things you can do with it

# Waits

- It can occur that the page is not finished loading when the script continues and converts the source code
- To prevent this, Waits can be used
- There are two main types of Waits:
  - Explicit Waits: Explicitly waits until a condition is fulfilled or a maximum time is reached
  - Implicit Waits: Usually set once and is a maximum waiting time whenever some element is looked for
- More details here: <https://selenium-python.readthedocs.io/waits.html>

## Explicit Waits with Expected Conditions

- What often comes in handy in browser automation are Explicit Waits with Expected Conditions
- Here, you can let the script pause until e.g. some element is visible on the web page
- Selenium features some methods that should be enough for most use cases: See Section 7.39 at <https://selenium-python.readthedocs.io/api.html>



## Finding the right button

- The page buttons are saved as children of the `<div class='ui pagination menu'>` tag.
- Their tags are of the form `<a class="item">`.
- Unfortunately, the “Next” button does not have a unique id/name.
- However, using `find_all()`, we can find the list of `<a class="item">` items, look at the last one, and determine if it is a “Next” button or not

## Day 3

---

Own script