BCCP Web Scraping Course

Day 1

Table of Contents

Day 1

very short intro to Python

Very short intro to Python

-

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

APIs

- 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`
- The JSON format has a similar structure as dictionaries in Python and can easily be transformed to DataFrames.

HTML parsing

- 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.
- Foe 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

Application Programming Interface

-

Twitter API

- "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?"

Day 2

Table of Contents

Day 2

HTML parsing

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.

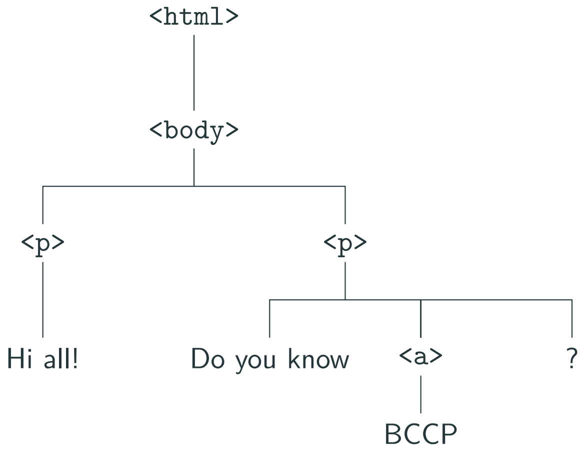
Basic HTML documents structure

- 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



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>16:4518:00</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/">
```

Analyzing the HTML structure v

```
      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

Day 3

Table of Contents

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 then 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` 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