SKILL SHARING SESSION

Web Scraping in Python

Beginner Tutorial

Paul Balluff
Department of Communication, University of Vienna
R 6.40, Kolingasse 14-16, 1090 Wien
paul.balluff@univie.ac.at

21st March 2022

Contents

Contents 2

1 Overview 3
- 1.1 What is covered . . . 3
- 1.2 What is not covered . . . 3
- 1.3 Requirements . . . 3
- 1.4 At the end of this session, you will 3

2 Planning your web scraping project 4

3 Legal considerations 5

4 Hyper Text Transfer Protocol 5
- 4.1 HTTP Basics . . . 5
- 4.2 The Anatomy of HTTP Requests & Responses . . . 6

5 Hyper Text Markup Language 8
- 5.1 HTML Basics . . . 8
- 5.2 Classes, IDs and other Attributes . . . 9
- 5.3 Head & Body . . . 9

6 Programming your first scraper 11
- 6.1 Installing required modules . . . 11
- 6.2 Importing the modules . . . 11
- 6.3 Making our first request . . . 12
- 6.4 Parsing HTML . . . 12
- 6.5 Getting a list of links . . . 15
- 6.6 Scraping one article . . . 16
- 6.7 Scraping every Article . . . 18
- 6.8 Saving scrape output to disk . . . 19

7 Web Scraping Toolbox 20
- 7.1 Inspect feature of browsers . . . 20
- 7.2 Machine Readable Resources . . . 21
- 7.3 Best Practices & Summary . . . 21

Useful Resources 22

1 Overview

Web scraping is as old as the internet—or at least as old as search engines. Or, how do you think a search engine such as Google knows the content of each page on every website? They have large scale, automated web crawling processes in place that extract important keywords from each page and add them to their index. This is just one example of how web scraping is commonly used in the internet. Knowing that scraping is a common and old practice, you can see that it is in fact something not mysterious at all.

1.1 What is covered

- Scraping websites
- Basics of the Hyper Text Transfer Protocol (HTTP)
- Basics of the Hyper Text Markup Language (HTML)
- Some notes on legal concerns
- Most commonly used python modules for web scraping
- Toolbox and tips
- Traps and common problems (if time allows)

1.2 What is not covered

- Using APIs
- Scraping specific social media platforms
- More specific or advanced python modules (e.g., `requests_html` or `selenium`)

1.3 Requirements

- Basic Python skills

1.4 At the end of this session, you will ...

- Be able to plan a web scraping project confidently
- Be informed about legal boundaries for scraping
- Understand the basics of HTTP Requests (status codes, anatomy of requests)
- Be able to parse simple HTML Documents
- Be able to use the python modules `requests` and `bs4`
- Have a sample script for a fully working scraper
- Know about useful tools such as the inspect mode in browsers

- Know about web standards such as sitemaps, news feeds, or `robots.txt`

2 Planning your web scraping project

Before you begin scraping, there are some considerations to make which inform many design decisions for your web scraper.

- Which website do you plan to scrape? Is it only one, a few, or many?
- Do you want to scrape the website(s) regularly or is it a one-off project?
- Is the content protected in some form ?
- Which elements of the content are you interested in? Are you seeking to scrape entire news articles, or user comments? Do you only need the headlines? Or is it something entirely different, such as the lottery numbers or the weather forecast?
- Do you need the raw data or is the parsed output sufficient?
- What do you need the data for? Are other people going to work with it?

Answering as many of these questions as possible, will save you time later, because you have a clear idea of your needs and how to design your scraper.

If you plan to scrape a website regularly, you need to think of an automated setup where human intervention is generally not needed. If you run a scraper on one website once, automation is not a big concern. Scraping one website is easier than scraping many different websites, because each website requires a different configuration for scraping and parsing.

If the content can only be accessed with a user login and password, you need to figure out how you can get a computer to perform the login process for you. As scraping is a common practice, some websites try everything they can to prevent users from automated access (for various reasons). But for every protection that they use, there is (at least) one way around it. But it will definitely make it harder for you to scrape their content.

The more content you need to scrape, the more space you will require on your hard disk. It is imperative to make some simple calculations on space requirements when you plan a long-term scraping project, or even intend to scrape large files such as raw HTML, images, or even videos. This also relates to the question what you need the data for. If you have no clear idea what exactly you are going to do, it is better to store the data as raw as possible (e.g., HTML files) and parse it at a later stage. If you collaborate

with others on the project you need to think of a simple way to share the data, and also documentation becomes more important.

3 Legal considerations

As mentioned in the introduction, web scraping is a common practice and therefore, it is not as illegal as you might have heard. Most importantly for us, "text and data mining" is granted for research purposes by the EU Directive 2019/790[1]. But there are important rules to follow:

- Your data collection activities may not interfere with the functionality of the website. This means, you should not overload the server and make too many requests.
- Copyright still applies, so you are generally not allowed to share the data with others (there are exceptions of course).
- You are not allowed to scrape where you do not have access. This means, you are not allowed to "hack" into a server and download any data. But you are allowed to collect data that is behind a paywall, as long as you are paying for the access to the content.
- You should also always respect the privacy of other users. If you collect usernames, email addresses, images of users, etc, be very careful with the data, or even consider not collecting it in the first place.

4 Hyper Text Transfer Protocol

4.1 HTTP basics

When computers communicate with each other, they use protocols. For each purpose there are several protocols, and the most relevant for us is the *Hyper Text Transfer Protocol* in short: **HTTP**. It is one of the fundamentals of the internet and more than 30 years old. Understanding some basics about it can greatly help you in programming a scraper. You will be able to understand what is happening in the background when you are scraping a website, which in turn helps you to write better scrapers and to be better prepared for potential problems and errors.

Ok, so what is this HTTP anyways? When you visit a website you typically use a something like Chrome, Firefox, or Safari. In these cases, the **client** that you use is a browser (we learn about other clients later). The **client** makes a **request** to the **server** and it uses HTTP to perform the request. The client asks the server for a specific **URL** and wants to see the content of that URL. With HTTP the client and the server "speak" a common language

[1] https://eur-lex.europa.eu/eli/dir/2019/790/oj

and therefore, "know" what to expect from each other. Then the **server** sends a **response** back to the **client**.

Of course not every request and response is successful, and thus, it is important to understand more about that in order to deal with such situations. More on unsuccessful requests later.

4.2 THE ANATOMY OF HTTP REQUESTS & RESPONSES

Requests and responses have typically two parts: **header** and the **content**. The header is sent or received at first and contains important information about the request/response. Then the content follows, which is what you would typically see in your browsers. In other words, in your daily life, the headers is what the computers need, and the content is what the human gets to see and what you would normally care about. But since we are programming scripts where computers supposed to communicate with each other, we need to understand some stuff about the non-human component.

Request Header

Here is an excerpt of a real request header that I made with Chrome:

```
GET /news-events/ HTTP/1.1
Host: compcommlab.univie.ac.at
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="99", "Google
↪  Chrome";v="99"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
↪  (KHTML, like Gecko) Chrome/99.0.4844.51 Safari/537.36
Accept: text/html,application/xhtml+xml,...
Accept-Encoding: gzip, deflate, br
Accept-Language: en-AT,en-GB;q=0.9,en;q=0.8,zh-HK;...
Cookie: amplitude_id_408774472b1245a7df5814f20e7484d0univie.ac...
```

The first line is the request line and is an integral part of HTTP. It consists of three elements: the **method**, the route, and the version of the protocol. In this case the method is `GET`. HTTP has eight different methods (or also called "verbs") that determine how the data is transmitted or what is expected from the server. For us, the most important HTTP methods are `GET`, `POST`, and `HEAD`[2]. With the `GET` method the client tells the server that it would like to *get* a specific page (or *route*). In the example above, the page

[2] the others are `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, and `PATCH`.

is right after the `GET` method and called `/news-events/`. The `POST` method is used to *post* data to the server. You typically find that in login forms, registration forms, or uploading an image. All these actions *post* data to the server. By using the `POST` method, the server knows that it should expect some data to follow after the header (unlike in a `GET` request). The `HEAD` method tells the server that the client only wants to get the header of the response and does not need the content (we look at a use case for that later).

The second line shows the host of the content, or the address of the server. This is because, the client's request is forwarded and redirected through the internet until it has reached the right server. The following lines are not of vital importance to us, but I would like to highlight here that they contain information about the operating system I used for making the request. The next relevant line is the `User-Agent` which in this case is Chrome. When we use a different client (e.g., python) this line changes accordingly. Sometimes websites block certain user agents which are not browsers. The following lines describe what content the client will `Accept` from the server and also which languages are preferred. The final line contains the cookies, which are a whole new chapter on its own (and not covered today).

Response Header

My request was successful and the response has the following header:

```
HTTP/1.1 200 OK
Date: Fri, 11 Mar 2022 17:21:29 GMT
Server: Apache/2.4.6 (Red Hat Enterprise Linux)
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-xss-protection: 1; mode=block
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: interest-cohort=()
Content-Security-Policy: frame-ancestors 'self'
Content-Length: 16334
Keep-Alive: timeout=5, max=96
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
Vary: Accept-Encoding
Content-Encoding: gzip
```

Analogous to the request, the first line is the most import and and called the *status line*. Right after the rather uninteresting version number, it shows the value `200 OK`. HTTP has a range of **status codes** that tell the client about the request's success. All status codes starting with `2` mean that the request was successful. Status codes starting with `4` tell the client that their request was invalid. Probably you have encountered the code `404` before, which means that the requested route does not exist. `403` means that your client

does not have the right permissions to access the page. Other status codes start with `3` to indicate redirects and starting with `5` to show that the request was unsuccessful, but it was the server's fault.

After the status line we have a variety of other information that we will not cover in detail here. The most relevant line for us is the `Content-Type` which states that the client will receive HTML content back from the server. Of course there are other content types possible, such as JSON or similar. Which leads us to the next topic, which is the content of a response.

5 Hyper Text Markup Language

5.1 HTML BASICS

HTML is not a programming language, it is a markup that is partially human-readable and that tells the browser how to render the received content. HTML uses so-called **tags** which fence off blocks of code from each other. Here is an example of a code block that shows a simple headline followed by a paragraph of text:

```
<h1>Simple Headline</h1>
<p>Here is some paragraph</p>
```

The first `<h1>` tag declares that a header element of level 1 follows. Then we see what the user will see: `Simple Headline`, and finally the tag is closed with `</h1>`. That's pretty straightforward, tags are opened like this `<tag>`, then there is some content and finally the tag is closed: `</tag>`. A HTML page is a large collection of such tags, which are typically nested within each other and form a hierarchy:

```
<article>
    <h1>Simple Headline</h1>
    <p>Here is some paragraph, and this <b>is a bold text</b> and
    ↪  this is in <em>italics</em>.</p>
</article>
```

The `<article>` tag encloses the entire section. Within the paragraph we can see that there are some words which are enclosed in tags such as `<b>` or `<em>`. These tags tell the browser that it should render the words inside in bold or italics.

In reality, the nested structure of HTML files has become rather complex, which often makes it difficult to navigate the raw code. And moreover, there is also an abundance of different tags: `<a>`, `<script>`, `<div>`, just to name a few important ones. But for web scraping we need to understand HTML, because we want to extract information and texts from the HTML pages that we requested.

5.2 CLASSES, IDS AND OTHER ATTRIBUTES

HTML tags can contain much more information that is hidden from the user and these are called attributes. These attributes are used to give the browser even more instructions. For example, how specifically it should render the contents of a tag, or where a hyperlink should lead to. Building on the example from above, it could look like this:

```
<article id="story-1" class="preview">
    <h1 class="underline">Simple Headline</h1>
    <p class="teaser-text">Here is some paragraph,
        and this <b>is a bold text</b>
        and this is in <em>italics</em>.</p>
    <p class="read-more">
        <a href="/link/to/news-article.html">Read More</a>
    </p>
</article>
```

This looks already much more cluttered than the plain example, but let's inspect some of the attributes. The `<article>` tag has the attributes `id` and `class`. The HTML specification states that each `id` has to be unique and only one element can have only one `id`. IDs are used by other parts of the browser to identify this specific element in the document. While we are not going into details, why web developers use IDs, they are *extremely* useful for us, because they allow us to find specific elements when we are parsing HTML files. `class`es are used to assign specific styles to a tag. How that works, is not important to us, but we have to remember here that several elements can all have the same classes. Unlike ids, they are not unique to each element. Furthermore, each tag can also have several classes that are separated with spaces: `<p class="teaser-text muted mx-3">`. Finally, it is important to know that the names of the classes are defined by the developers of the website you're browsing. This means that they are probably very different on every website!

Moving down, in the second `<p>`, we can see an `<a>` tag. "A" stands for "anchor" and these tags represent hyperlinks to other pages. The `href` attribute contains the path or URL to the destination. In this example, the `href` points at an internal location, we know that because it starts with an `/` and not with an URL of another address. This attribute is useful for scraping, because it leads our scraper to other pages that we might want to scrape as well.

5.3 HEAD & BODY

HTML pages are typically divided into to major blocks: `<head>` and `<body>`. Similar to HTTP requests and responses, the `<head>` section of a HTML file is invisible to the user, but holds important information for the browser. It also often contains meta information which are in a machine readable

format—perfect for web scraping! Meta data is used by a variety of services that you know, such as search engines. But maybe you have also seen before, that when you post a link in a chat (e.g., WhatsApp, Slack) or on Twitter, that a small preview of the link appears in the chat. This preview is also generated with a standardized set of meta information.

Typically meta information are inside `<meta>` tags and they also have some attributes that are useful. Here an example:

```
<head>
    <meta name="keywords"
    ↪  content="travel,ARTICLE,story,food-drink">
    <meta name="description" content="Serving up a rarely changing
    ↪  menu of quirky Canto-European [...]">
    <meta name="author" content="Gary Jones">
    <meta property="“article:published_time“
    ↪  content="“2022-03-11T09:21:19+00:00">
    <meta property="og:title" content="Hong Kong's 'greasy spoon'
    ↪  cafes">
    <meta property="og:type" content="article">
    <meta property="og:url"
    ↪  content="https://www.bbc.com/travel/article/20220310...">
    <meta property="og:description" content="Serving up a rarely
    ↪  changing menu of quirky Canto-European [...]">
    <meta property="og:author" content="Gary Jones">
    <meta property="og:image"
    ↪  content="https://ychef.files.bbci.co.uk/live/624x351/p0bsgd6g.jpg">
    <meta name="twitter:card" content="summary_large_image">
    <meta name="twitter:site" content="@bbc_travel">
    <meta name="twitter:title" content="Hong Kong's 'greasy spoon'
    ↪  cafes">
    <meta name="twitter:description" content="Serving up a rarely
    ↪  changing menu of quirky Canto-European [...]">
</head>
```

This example uses several standards of meta tags at once: the generic one, `og:` and `twitter`. We can see that these use the attributes `name` or `property` to declare the kind of information contained. The meta tag in the example contains `keywords`, which are useful for search engines. The second line has a short preview text, and the third line has the name of the author, fourth line states the date of publication and so forth. When we scrape large amounts of data, it can be very useful to leverage this meta

information to store our scraped data.

6 Programming your first scraper

In this section we develop a fully functioning web scraper using the python modules Requests and Beautiful Soup. As an example, we are going to scrape the latest articles from Wikinews. We want to include the following information:

- URL of the article
- Headline of the article
- Publication Date
- Full text of the article

To give an overview of the required steps:

1. Request the main page
2. Get a list of latest news
3. Request each URL leading to a news item
4. Parse each news article
5. Finally store the data in a file for later use

6.1 Installing required modules

First, we need to make sure that we have some non-standard modules installed:

```
python3 -m pip install requests beautifulsoup4 lxml python-dateutil
↪  pyarrow
```

The `requests` module makes it easy to perform HTTP requests. `beautifulsoup4` allows us to parse HTML documents. The other packages are helper modules that are explained later.

6.2 Importing the modules

We make our imports at the beginning of our scraper:

```
import requests
from bs4 import BeautifulSoup

# Module for sleep timers
from time import sleep
```

```
# Helper module for pretty printing output
from pprint import pprint
```

The only thing to note here is that Beautiful Soup is imported via `bs4` and we only need the `BeautifulSoup` class from the module.

6.3 MAKING OUR FIRST REQUEST

When programming a scraper it is easier to have the targeted website opened in a browser. So first we open the English edition of Wikinews in Firefox or Chrome.

Then we copy the URL from the browser's address bar. This way we can make sure that we have the correct address and also the right protocoll without any typos. We store the address in an object:

```
website = "https://en.wikinews.org"
```

To perform the `GET` request it is as simple as follows:

```
r = requests.get(website)
```

We can now check whether the request was successful, by accessing the status code:

```
r.status_code
# 200
```

`200` tells us that the request was successful. We can also explore other properties of the response object `r`. As mentioned in the previous sections, each HTTP response has a header and a content part. In the `requests` module they are just simple properties and returned as strings.

```
r.headers
r.content
```

6.4 PARSING HTML

We can proceed and pass the content of the response into `BeautifulSoup`. The first argument is the HTML we want to parse, the second argument is the parsing engine: `lxml`.

```
soup = BeautifulSoup(r.content, 'lxml')
```

Using "Inspect"

We have a large pile of HTML code and it is very difficult to read all of that in the Python console. Instead, we are going to use the "Inspect" feature of our browser. We are interested in the section labelled "Latest news" (Figure 1).

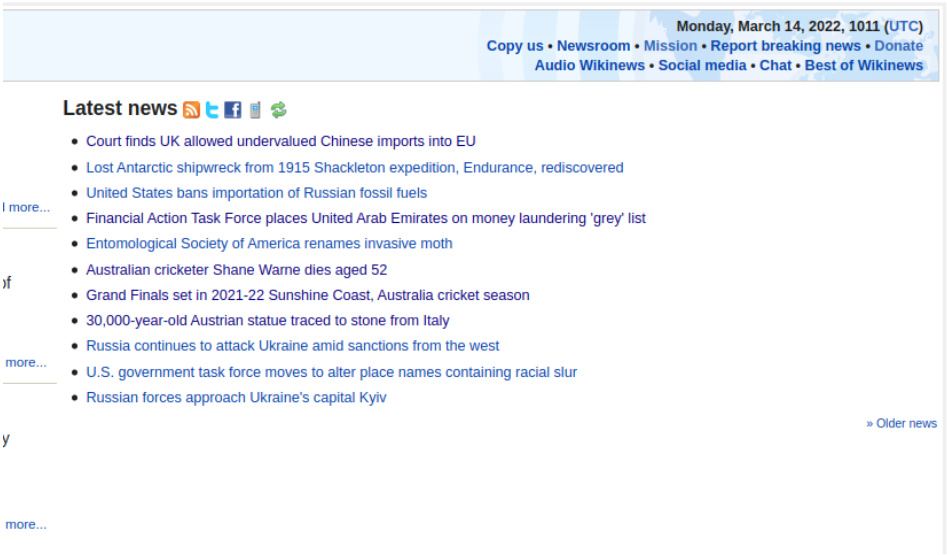

Figure 1: Wikinews Latest News

If we right-click on the headline, we see the option "Inspect" inside the context menu (Figure 2).

Now the browser window is split into two parts: the normal view and the HTML code inspection. It depends on your settings, I prefer having the inspection at the bottom of the screen (Figure 3).

If you look carefully, you can see that one HTML tag is already highlighted. This is the very element that you selected when you righ-clicked "Inspect" on it. When you hover over the HTML tags in the code view, the corresponding elements are also highlighted in the browser main window. That makes it rather easy to identify the element that we are looking for.

We want to get the links to the latest news, therefore, we look for their direct parent element that enclosese all of them (Figure 4).

Navigating in Beautiful Soup

To select the element with the ID `MainPage_latest_news_text` We use the `find()` method: it finds the first tag meeting the criteria we specify. Remember that ids are unique, therefore, the first hit is always the correct one

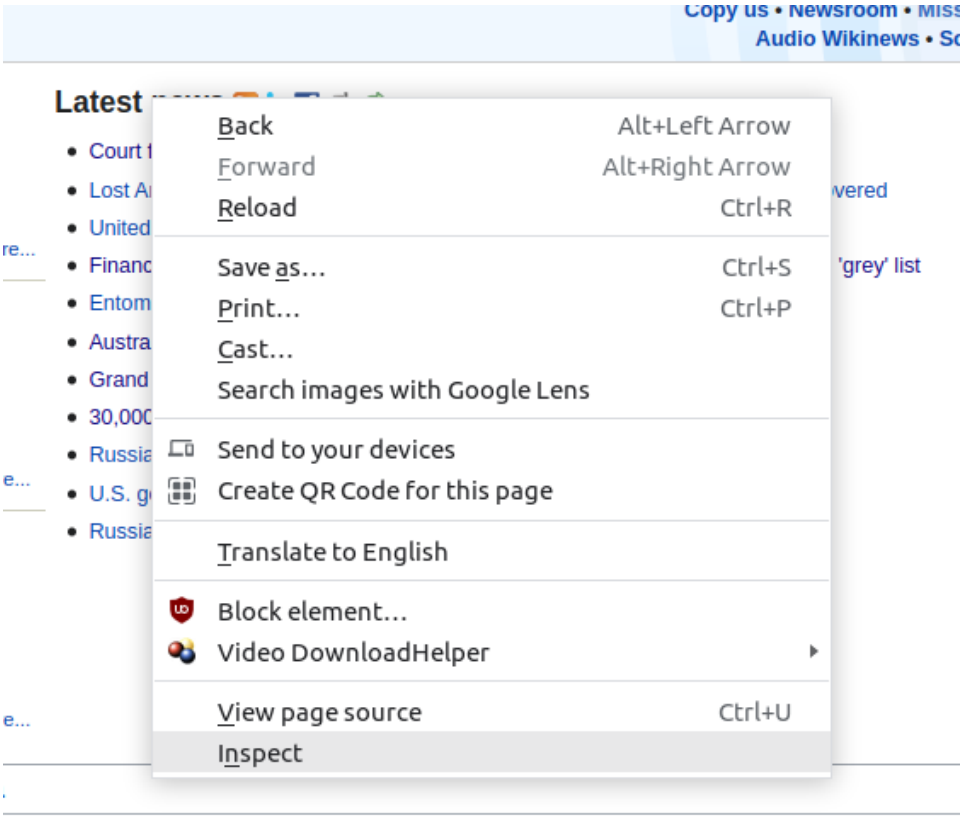

Figure 2: Inspect Context Menu

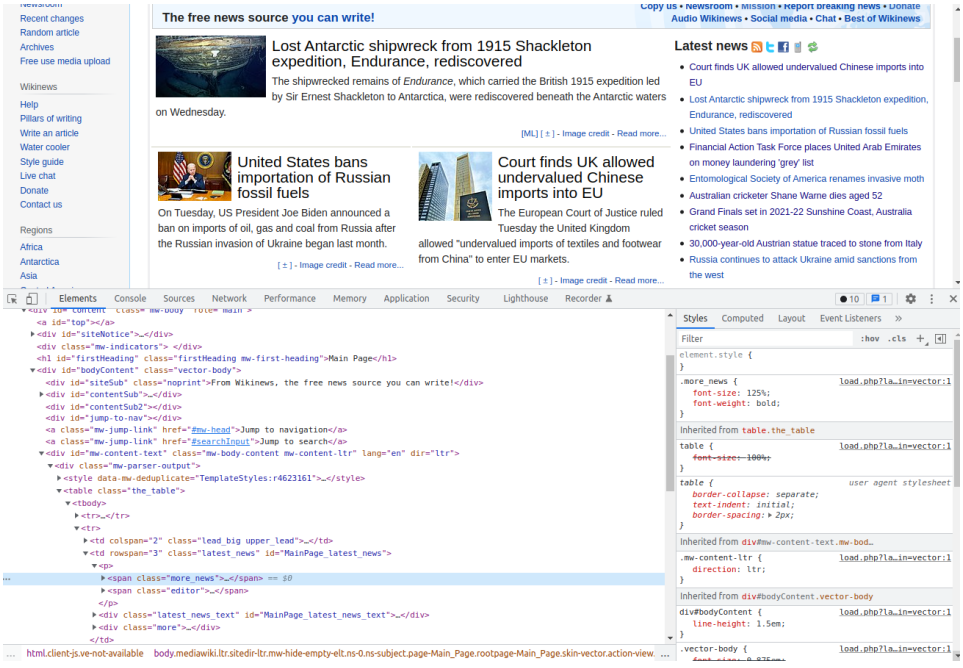

Figure 3: Inspect Code View

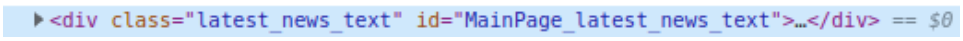

Figure 4: HTML code of of Latest news

```
latest_news = soup.find(id="MainPage_latest_news_text")
```

Now we want to get all links inside the element. Remember that links are inside `<a>` tags. We use the `find_all()` method, it returns all elements that meet our specified criteria. This means we get a list of `<a>` tags.

```
a_tags = latest_news.find_all('a')
```

In order to look at the first element we can simply access it like this:

```
a_tags[0]
```

Now we want to extract the URL to the full article and also the headline of the article. Beautiful Soup allows us to access HTML attributes like a Python dictionary. Links are inside the `href` attribute:

```
a_tags[0]['href']
# '/wiki/Court_finds_UK_allowed_undervalued_Chinese_imports_into_EU'
```

In this particular example, the `href` contains a *relative* link. As you can see, the URL starts with `/wiki/`. In order to get the complete URL, we have to add the main website:

```
pprint(website + a_tags[0]['href'])
```

It is a good idea to manually test, if the link really works by copying it into our browser's address bar.

The headline is the bare text, we can access it with the `get_text()` method

```
pprint(a_tags[0].get_text())
```

6.5 GETTING A LIST OF LINKS

Now we want to do extract the link for all `<a>` and store them inside a list So we create an empty list first, where each element is a news article

```
articles = []
```

There are different ways of approaching this. I prefer using dictionaries to represent news articles. So an article would look like this:

```
article = {'url': 'https://en.wikinews.org/wiki/Court_finds_UK_...',
           'headline': 'Court finds UK allowed ...',
           'publication_date': "2022-03-11",
           'text': 'blablablabla'}
```

Now we iterate through a_tags and each time we extract the url and the headline. And then we append the article dictionary to the articles list

```
for a in a_tags:
    url = a['href']
    headline = a.get_text()
    article = {'url': website + url,
               'headline': headline}
    articles.append(article)
```

Let's check the output:

```
pprint(articles)
```

6.6 SCRAPING ONE ARTICLE

Next we want to get the full text and publication date for each article. Therefore, we have to request each link and parse the output again. Unlike the first time, we have to iterate over a list and make a request each time. We run an example first with a single article, to see how it works:

```
article = articles[0]
```

Make a `GET` request:

```
r = requests.get(article['url'])
```

Check the status code:

```
r.status_code
# 200
```

And pass the content into Beautiful Soup:

```
soup = BeautifulSoup(r.content, 'lxml')
```

First, we try to get the publish date at the top of the article. Again, we use the inspect feature of the browser to identify the element. It is inside the `<strong>` with the class `"published"`. Note that we need an underscore when we address an HTML class in python, because `class` is a reserved keyword in Python:

```
published_date = soup.find(class_="published")
```

We can now get the text using the 'get_text()' method:

```
published_date_string = published_date.get_text()
```

However, this is not a very nice format, because it is not machine-readable. We can improve this by using the `python-dateutil`, which is a non-standard module:

```
from dateutil import parser as dateparser

parsed_date = dateparser.parse(published_date_string)
```

This way, we get a nice Python `datetime` object. However, this kind of parsing is not very robust and prone to error. It is recommended to only use it if there is no other way of retrieving the date.

Another way is to look closer at the HTML attributes. The `<span>` tag inside the `<strong>` tag has a `title` attribute, which contains the date as machine readable string. We can access children tags as a property and then get the attribute like a dictionary:

```
published_date_title = published_date.span['title']
```

Next we get the text of the article. For that, we identify the parent element of the article text using the inspect feature. All text content is inside the `<div>` with the class `mw-parser-output`:

```
content = soup.find(class_="mw-parser-output")
```

We want to get the text of every `<p>` tag inside the content box. So, first we select all `<p>` tags:

```
p_tags = content.find_all('p')
```

Then we iterate over all of them:

```
for p in p_tags:
    pprint(p.get_text())
```

Almost perfect, but we do not want to get the first `<p>` as this is the date that we already parsed. We also do not need the last `<p>` as it is always the same text in every article. We can simply skip them like this:

```
for p in p_tags[1:-2]:
    pprint(p.get_text())
```

Now we just have to turn the single strings into a long one, by appending each `p.get_text()` to a list and then join it to one large string:

```
text = []

for p in p_tags[1:-2]:
    text.append(p.get_text())

text = "\n\n".join(text)
```

For the experts: here is the one liner solution

```
text = "\n\n".join([p.get_text() for p in p_tags[1:-2]])
```

Finally, we add our parsed results to the article dictionary

```
article['published_date'] = published_date_title
article['text'] = text
```

6.7 SCRAPING EVERY ARTICLE

Now we do the same we did before but for every article, inside in a loop. The code is almost the same, but we will add some more extras:

```
for article in articles:
    r = requests.get(article['url'])

    article['status_code'] = r.status_code

    if r.status_code != 200:
        continue

    soup = BeautifulSoup(r.content, 'lxml')
```

```
    published_date = soup.find(class_="published")
    published_date_title = published_date.span['title']

    content = soup.find(class_="mw-parser-output")
    p_tags = content.find_all('p')
    text = []

    for p in p_tags[1:-2]:
        text.append(p.get_text())

    text = "\n\n".join(text)

    article['published_date'] = published_date_title
    article['text'] = text

    sleep(0.2)
```

Let's go through this loop step-by-step. We have a list `articles` that contains dictionaries that represent single news articles. When the loop starts, each dictionary only has the keys `url` and `headline`.

First, we request the URL for the article and check the status code for every request in the loop. If the request was unsuccessful, we skip it and continue.

Then the request content is passed into Beautiful Soup where we first extract the date of article publication. Next we extract the text of the article by getting all `<p>` inside the element with the HTML class `mw-parser-output`.

Next, these two extracted elements are then added to the `article` dictionary as new keys: `published_date` and `text`.

Finally, we add a short sleep timer at the end of each iteration to prevent overloading the server. 200 milliseconds (0.2 seconds) are enough.

6.8 SAVING SCRAPE OUTPUT TO DISK

Finally, we are going to store the output in a separate file. There are many ways to achieve this, and here we look at two of them. First, we define a file name that also has a timestamp in it. This prevents overwriting previous scraper output.

```
from datetime import datetime

output_file_name = "wikinews_scraper_output_" +
↪  datetime.now().strftime('%Y%m%d_%H.%M.%S')
```

Side note: the expression `%Y%m%d_%H.%M.%S` tells how to format the current date time. You can look up the syntax in the Python documentation.

Now we can use the JSON module to export the list of dictionaries. To store to a JSON file, we use a context manager. The `open()` function takes two positional arguments: `filename` and `mode`. We pass `'w'` as mode for "write".

```
import json

with open(output_file_name + '.json', 'w') as f:
    json.dump(articles, f)
```

Another method is to use pandas. We can save the output in any format that pandas supports. I strongly advice against using csv files, as these are not suitable for text data. Instead we are using the feather format, which is designed to exchange large amounts of data between python, R and other programming languages.

```
import pandas as pd

df = pd.DataFrame(articles)

df.to_feather(output_file_name + '.feather')
```

Now we have our output ready and we are done!

7 Web Scraping Toolbox

In this section, I briefly discuss some tools that we used, as well as some parts of web standards that are useful for us.

7.1 Inspect feature of browsers

We saw that it is extremely helpful for navigating the HTML jungle. Browsers have become a central tool in web development and the inspect feature has a lot more to offer than that what we used so far. I would really recommend you to get familiar with the different tabs in the inspect screen. The most important one besides the code browsing is the network tab. Try it out by reloading a page and watch what happens. It shows you every request that the browser made to retrieve the page that you are looking at. This is useful for examining special settings inside the request headers. Or is also useful for quickly getting the User-Agent of your browser.

7.2 MACHINE READABLE RESOURCES

Websites want to be found. They want to make it easy for Google & others to parse their content as efficiently as possible. Therefore, there are some standards that we can leverage for our advantage. The most important ones are the `robots.txt` file and sitemaps.

Almost every website has a `robots.txt` file. You can access it by just adding it to the base URL of a website. For example, the `robots.txt` of the BBC website is located here: `https://www.bbc.co.uk/robots.txt`. It contains a list of pages where bots and scrapers are allowed to go and where not. Of course, you do not have to respect that. But sometimes you should. More importantly it contains a list of sitemaps. These are XML files that contain a list of every page in the website. The exact content of the sitemap is up to the web developers and may differ to a great extent. Some news sites maintain a sitemap only for the most recent articles. Others have a sitemap for every month since they began publishing! Therefore, it is imperative to check these things before you start programming your scraper. It can safe you a headache along the way.

Related to sitemaps are news feeds, they contain links to the most recent articles. These tend to be outdated by now, but many websites still offer that service. RSS is the most common format for news feeds. The great thing about RSS feeds and Sitemaps is that they follow the same syntax as HTML. That means, if you can parse HTML, you can also parse RSS and XML sitemaps. And Beautiful Soup is the tool that you just learned to use.

7.3 BEST PRACTICES & SUMMARY

Here a quick summary of things to keep in mind when you design and program a web scraper. To follow the legal guidelines you should definitely think of ways to ease the load on the server. Therefore, you should always add a sleep timer of at least 0.2 seconds. More than 1 second is only necessary in extreme cases.

You should also think of where your script might fail and throw an error at you. Thus, insert fail safes into your script. What to do if status code is not 200? How to deal with problems in parsing? Sometimes it can happen that an HTML element is not found or does not have the attribute that you expected. It is a good idea to add `try:` and `except:` to sections of your code where you anticipate problems.

I would strongly recommend storing the data in a format that is suitable to hold text. Recommended file formats are feather, arrow, xlsx, SQL, NoSQL, or JSON. I would advise against using CSV files for storing text data. This format is not really suitable, since longer texts with lots of quotation marks and new lines can confuse even the best CSV parser. Moreover, CSVs are not really efficient in storing text data and tend to get quite large.

You might also want to consider storing the raw HTML and parse it later (e.g., for long-term scraping projects). This way you can keep the scraping and parsing separate and address problems in a more targeted fashion.

Useful Resources

Niemeyer, G. & Ganssle, P. (2022). *Dateutil - powerful extensions to datetime*. Version 2.8.2. https://dateutil.readthedocs.io/

Reitz, K. (2022). *Requests: Http for humans™*. Version 2.27.1. https://docs.python-requests.org/

Richardson, L. (2022). *Beautiful soup documentation*. Version 4.8.1. https://beautiful-soup-4.readthedocs.io/

Schafer, C. (2017, November 8). *Python tutorial: Web scraping with beautifulsoup and requests*. Youtube. https://www.youtube.com/watch?v=ng2o98k983k
Note: Highly recommended to watch this video.

Schafer, C. (2019, March 11). *Python tutorial: Web scraping with requests-html*. Youtube. https://www.youtube.com/watch?v=a6fIbtFB46g