Web Scraping mit Python und BeautifulSoup

Jonathan Helgert

4. November 2021

`https://github.com/jhelgert/web_scraping_einfuehrung`

Intro

Beispielprojekte

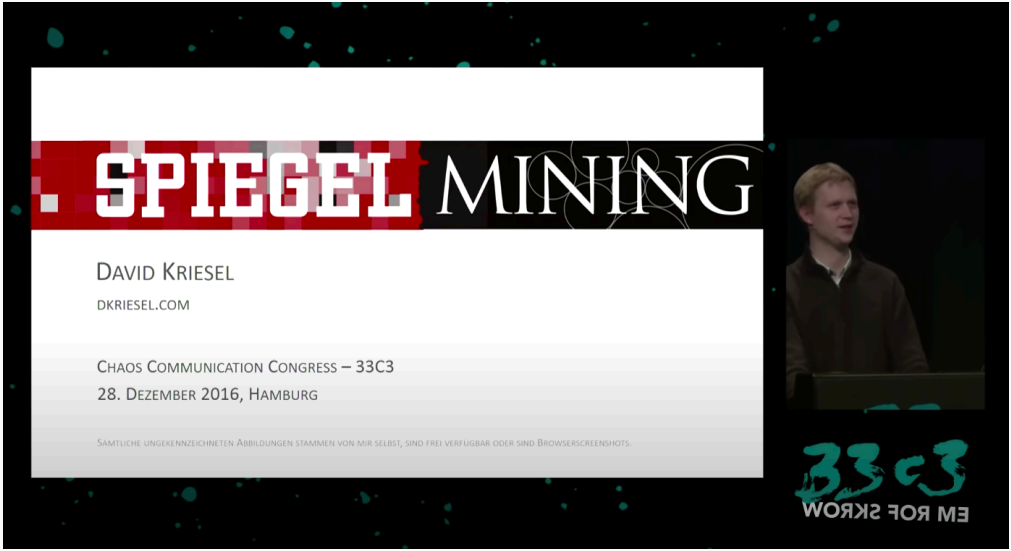

Abbildung 1: `https://youtu.be/-YpwsdRKt8Q`

Abbildung 2: `https://github.com/iamnotturner/vaccipy`

Ilias Downloader UniMA

code quality A+ | build passing | python 3 | pypi v0.5.0 | downloads 53/month

A simple python package to download files from https://ilias.uni-mannheim.de.

- Automatically synchronizes all files for each download. Only new or updated files and videos will be downloaded.
- Uses the BeautifulSoup package for scraping and the multiprocessing package to accelerate the download.

Abbildung 3: `https://github.com/jhelgert/IliasDownloaderUniMA`

Web Scraping Libraries

Es werden hauptsächlich drei Libraries verwendet:

- **Selenium**: Eine Automatisierungslibrary, mit der man das Browserfenster beliebig steuern kann. Flexibel, aber mMn sehr umständlich.
- **Scrapy**: Für Massenscraping von mehreren Seiten ausgelegt. Mächtig, aber zu Beginn etwas unübersichtlich.
- **BeautifulSoup**: Eine elegante und extrem verbreitete Library, welche einfache Funktionen bietet um innerhalb eines HTML Dokuments zu Suchen und Extrahieren.

Let's get started

Wir verwenden zwei Libraries: Requests und Beautifulsoup.

- „**Requests** is an elegant and simple HTTP library for Python, built for human beings. [...] allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your POST data."
- „**BeautifulSoup** is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree".
- **Hinweis**: Requests unterstützt kein Javascript. Also sind dynamische Daten, die seitens des Servers durch ein Javascript-Framework erzeugt werden, nicht sichtbar.

requests.get()

Mit Hilfe von `requests.get()` Methode können wir extrem leicht über das HTTP(S) Protokoll eine `GET`-Anfrage stellen und damit einen Seitenaufruf im Browser simulieren.

- Jede HTTP-Anfrage liefert einen Statuscode, z.B: 200 (*ok*), 403 (*forbidden*), 404 (*not found*).

Beispiel:

```
from requests import get


res = get("https://www.google.de")
print(res.ok)          # True (GET-Anfrage erfolgreich)
print(res.status_code) # 200 = OK
print(res.content)     # HTML-Code
print(res.headers)     # Headers unsere GET-Anfrage
```

- Mittels der *headers* bekommt der HTTP-Server Metadaten, um seine Antwort entsprechend anzupassen oder gar zu verweigern.
- Deshalb sollte unsere `GET`-Anfrage möglichst wie die eines Browsers aussehen.
- Der `user-agent`-Schlüssel enthält dabei z.B. Infos über unser Betriebssystem und den Browser.

Beispiel:

```
from requests import get


res = get("https://www.the-future-of-commerce.com/")
print(res.status_code) # 403 Forbidden :(
```

Was nun?

- Bisher wurde kein `user-agent` verwendet $\Longrightarrow$ Der Server weiß, dass wir kein Browser sind.
- **Tipp**: Falls nötig, die kompletten *request headers* vom eigenen Browser übernehmen (siehe Chrome Developertools).
- Damit die `GET`-Anfrage wie die eines Browsers aussieht, sollten wir einen verwenden:

```
headers = {'user-agent': ("Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) "
                          "AppleWebKit/537.36 (KHTML, like Gecko) "
                          "Chrome/95.0.4638.54 Safari/537.36")}
res = get("https://www.the-future-of-commerce.com/", headers=headers)
print(res.status_code) # 200 OK :)
```

Absicherung gegen Web-Scraping

Es gibt insgesamt mehrere mögliche Hürden, die Web-Scraping erschweren.

- Blockieren von Anfragen mit verdächtigen *requests headers*.
- Limitierung der erlaubten Anfragen je IP-Adresse innerhalb eines bestimmten Zeitraums.
- Verschlüsseln sensibler Daten oder Darstellung als Bild statt als Text (z.B. Emails).
- Authentifizierung durch Captchas.

HTML Basics

HTML Basics

Allgemeine Syntax `<tagname attribut1='value1' attribut2='value2'>...</tagname>`
Beispiel:

```
<!DOCTYPE html>
<html>
<body>
<a href="bla.html">Klick mich1</a>
<a href="bla.html" class="css_class1">Klick mich2</a>
<a href="bla.html" id="irgendeine_id1" class="css_class1">Klick mich3</a>
<img src="https://foo.com/image.mypng" alt="DummyText">
</body>
</html>
```

- Häufig vorkommende Tags sind z.B. `p`, `a`, `div`, `table`, `img`.
- Häufig verwendete Tagattribute wären z.B. `class`, `id`, `href`, `src`.

BeautifulSoup

BeautifulSoup

Wichtigste Klassen

bs4.BeautifulSoup

Ausgangspunkt ist immer eine Instanz der `bs4.BeautifulSoup` Klasse, deren Konstruktor den HTML Code als `str` und optional einen HTML-Parser als Argument erwartet:

```
from bs4 import BeautifulSoup
soup1 = BeautifulSoup(html_str)           # Verwendet den Standardparser
soup2 = BeautifulSoup(html_str, "lxml")   # Wähle "lxml"-Parser (Cython)
```

Die beiden wichtigsten Klassen:

- `bs4.BeautifulSoup` enthält das komplette HTML Dokument, das wir verarbeiten möchten.
- `bs4.element.Tag` entspricht einem Tag des HTML Dokuments. Bietet Methoden und Iteratoren zum Zugriff auf Tagattribute oder ineinander verschachtelte Tags.
- Die meisten Methoden und Iteratoren sind in beiden Klassen vertreten.

Wir können innerhalb des HTML Dokuments navigieren, indem wir den Tagnamen als Objektattribut verwenden:

```
soup = BeautifulSoup("""<html><body><head><h1>Überschrift</h1></head>
                                    <p>Dummytext.</p>
                                </body></html>""", "lxml")

print(soup.body)          # <body><h1>Überschrift</h1><p>Dummytext.</p></body>
print(soup.body.h1)       # <h1>Überschrift</h1>
print(soup.body.h1.text) # Überschrift

print(type(soup.body))    # bs4.element.Tag
print(type(soup.footer)) # None, da kein footer-Tag vorhanden
```

Beachte: Es wird lediglich der erste Tag gewählt, der zutrifft.

Die drei häufigst benötigten Klassenatribute sind `.name`, `.attrs` und `.text`. Beispiele:

```
soup = BeautifulSoup("<a class='c1 c2' href='http://example.de'>Klick mich!</a>")
tag = soup.a     # type(tag) = bs4.element.Tag

# Tagobjektattribute
print(tag.name)  # "a"
print(tag.attrs) # {'class' : ['c1', 'c2'], 'href' : 'http://example.de'}
print(tag.text)  # "Klick mich!"
# Check auf Tagattribute
print(tag.has_attr("class")) # True
print(tag.has_attr("foo"))   # False
# Zugriff auf Tagattribute
print(tag["class"]) # ['c1', 'c2']
print(tag["href"])  # "http://example.de"
```

BeautifulSoup

Filter

`find()` und `find_all()`

Mit Hilfe von `find_all()` und `find()` können wir Tags basierend auf einem Filter finden.

- Die Methode `find_all()` sucht rekursiv innerhalb der Nachfolgertags und gibt eine Liste aller gefunden Tags zurück.
- Dagegen gibt `find()` nur den ersten gefunden Tag zurück.

Signatur der Methode:

```
def find_all(name=None, attrs={}, recursive=True, text=None, limit=None, **kwargs)
# name: A filter on tag name.
# attrs: A dictionary of filters on attribute values.
# recursive: perform a recursive search of this PageElement's children?
# limit: Stop looking after finding this many results.
# kwargs: A dictionary of filters on attribute values.
```

Das `name`-Argument sucht alle Tags basierend auf den Tagnamen:

```
soup = BeautifulSoup("""<div><a class='c1'>A</a>"
                            <a class='c2'><span>S</span></a>
                        </div>""")

# Simpler String "a" als Filter
print(soup.find_all("a")) # [<a class="c1">A</a>, <a class="c2"><span>S</span></a>]

# Liste von Strings als Filter (logisches ODER)
print(soup.find_all(["a", "span"])) # [das gleiche wie oben, <span>S</span>]
```

find_all()

Mit Hilfe des `attrs`-Arguments können wir nach Tags suchen, die bestimmte Attributwerte erfüllen müssen:

```
soup = BeautifulSoup("""<div>
        <a class='c1'    id="myid1">A</a>"                    <!-- (1) -->
        <a class='c2'    id="myid2"><span>S</span></a>        <!-- (2) -->
        <a class='c1 c2' id="myid4"><span>ACHTUNG</span></a>  <!-- (3) -->
    </div>""", "lxml")

soup.find_all("a", {'class': 'c2'})                # (2), (3)
soup.find_all("a", {'class': 'c2', 'id': 'myid2'}) # (2)
soup.find_all("a", {'class': 'c1 c2'})             # (3)
soup.find_all("a", {'class': ['c2']})              # (2), (3)
```

Wichtig: `class` kann mehrere Attributwerte besitzen. Min. einer muss passen.

find_all()

Durch `**kwargs` können wir Attribute und Attributwerte auch direkt als Filter verwenden.

- Wichtig: Der Attributname selbst muss ein gültiger Pythonvariablennamen sein.
- Es gibt daher einige Fälle, in denen man stattdessen `attrs` verwenden muss.

Beispiele:

```
#soup.find_all(class = "c1")     # 'class' ist reserviertes Python keyword
#soup.find_all(name = "hallo")   # 'name' ist bereits Argument von find_all
#soup.find_all(data-odd = "123") # "data-odd" kein gültiger Variablenname

# Kein Problem:
soup.find_all("a", id="myid3")  # Alle a-tags mit id-Attribut und Wert "myid3"
soup.find_all("a", id=True)     # Alle a-tags mit einem id-Attribut und bel. Wert.
soup.find_all("a", class_="c2", id=True) # Klar
```

Filter im Detail: Lambdas

Für komplexere Filter bieten sich eigene Funktionen als Filter an.

- Die Funktion muss als einziges Argument ein `bs4.element.Tag` Objekt erwarten und `True` bzw `False` zurückgeben.
- Übergibt man die Funktion als Filter für ein bestimmtes Attribut, ist das erwartete Funktionsargument stattdessen der Attributwert.

```
def my_filter(tag):
    return tag.has_attr("class") and tag["class"] == ["c1"]


soup.find_all(my_filter)
soup.find_all(lambda tag: tag.has_attr("href") and ".pdf" in tag["href"])
soup.find_all(lambda tag: tag.has_attr("foo") and not tag.has_attr("foo2"))
soup.find_all(lambda tag: len(tag.attrs) == 2)
soup.find_all(href=lambda h: h is not None and ".pdf" in h) # h ist ein str
```

Filter im Detail: Reguläre Ausdrücke

Reguläre Ausdrücke sind ein extrem mächtiges Werkzeug um Strings nach bestimmten Eigenschaften zu filtern.

Beispiele:

```
import re
#  *    matched das vorherige Zeichen bzw. den Teilausdruck 0 or mehrmals
#  .    matched jedes beliebige Zeichen
# [A-Z] matched jedes jedes Zeichen von A bis Z
#  $    matched das Ende des Strings
#  ^    matched den Anfang des Strings
soup.find_all(re.compile(r"h[0-9]"))                    # z.B. h1bla, aah2bla Tags
soup.find_all(re.compile(r"^h[0-9]"))                   # h1, ..., h9 Tags
soup.find_all("a", class_=re.compile(r"c"))             # CSS Klassen mit "c"
soup.find_all("a", href=re.compile(r"pdf$"))            # Alle .pdf Links
soup.find_all("a", href=re.compile(r".*uebung.*pdf$"))  # ...
```

BeautifulSoup

Iteratoren

bs4.element.Tag.children und .descendants und .parents

- `.children` liefert die direkten Nachfolgertags innerhalb inseres Tags.
- `.descendants` liefert die rekursiven Nachfolgertags innerhalb inseres Tags.
- `.parents` liefert die rekursiven Vorgängertags unseres Tags.

```
soup = BeautifulSoup("<html><body><div><a class='c1'><span>S</span></a></div></body></html"
tag1, tag2 = soup.div, soup.div.a  # type = bs4.element.Tag

l1 = [c for c in tag1.children]
#  = [<a class="c1"><span>S</span></a>]
l2 = [d for d in tag1.descendants]
#  = [<a class="c1"><span>S</span></a>, <span>S</span>, "S"]
l3 = [p for p in tag2.parents]
#  = [<div><a class='c1'><span>S</span></a></div>, <body><div>...</div></body>, ...]
```

bs4.element.Tag.next_siblings und .next_siblings

- `.next_siblings` enthält alle nachfolgenden Nachbartags des gleichen Levels ($\Longrightarrow$ alle Tags mit denselben `.parents`)
- `.next_elements` ähnlich zu `.next_siblings`, enthält allerdings auch die rekursiven *children* jedes Nachbartags.

Beispiel:

```
soup = BeautifulSoup("<html><body><p><i>text2</i><b>text1</b></p></body></html>")
tag = soup.i

l1 = [s for s in tag.next_siblings]  # [<b>text1</b>]
l2 = [s for s in tag.next_elements]  # ['text2', <b>text1</b>, 'text1']
```

bs4.element.Tag.previous_siblings und .previous_elements

- `.previous_siblings` analog zu `.next_siblings`, allerdings der vorherfolgenden Nachbartags.
- `.previous_elements` analog zu `.next_elements`, allerdings mit vorherfolgenden Nachbartags und rekursiven children.

Beispiel:

```
soup = BeautifulSoup("<html><body><p><i>text2</i><b>text1</b></p></body></html>")
tag = soup.b

l1 = [s for s in tag.previous_siblings] # [<i>text2</i>]
l2 = [s for s in tag.previous_elements] # ['text2', <i>text2</i>,
                                        #  <p><i>text2</i><b>text1</b></p>, ...]
```

Logins und Cookies

Logins und Cookies

HTML-Forms

HTML-Forms

```
<form action="https://form_action_url.com/bla/api/blub" method="GET|POST">
<input type="text" name="first_name"/>
<input type="text" name="last_name"/>
<input type="hidden" value="abc1234"/> <!-- ggf. weitere Formularelemente -->
<input type="submit" name="submit" value="Anmelden"/>
</form>
```

- Die Formdaten werden an die URL des `action`-Attributwertes gesendet.
- Das `type`-Attribut legt den Typ fest, z.B. `text`, `email`, `number`, `hidden`.
- Das `name`-Attribut legt den Namen der Eingabe des Eingabeelements fest.
- Die Eingebwarte der Eingabefelder sind nach Ausfüllen des Formulars die `value`-Attributwerte. Der Wert kann auch bereits vorgegeben sein.

HTML-Forms

Das Ausfüllen einer HTML-Form ist ziemlich bequem:

```
# Namen der Eingabefelder inklusive der Werte, die wir abschicken bzw. eingeben.
form_data = { 'first_name' : 'Walter',
              'last_name': 'Heisenberg',
              'hidden': 'abc1234',
              'submit': 'Anmelden'}

# Falls method="GET"
req1 = requests.get(action_url, data=form_data, headers=headers)
# Falls method="POST"
req2 = requests.post(action_url, data=form_data, headers=headers)
```

requests.session

Eine Instanz der Klasse `requests.session` speichert alle Cookies, Headers und sonstige Metainformationen einer *Session* für jede `GET`- oder `POST`-Anfrage.

```
with requests.session() as s:
    # POST-Anfrage. Analog s.get(...) für GET
    req = s.post(url, data=form_data, headers=headers)
    # Jetzt sind wir eingeloggt:
    soup = BeautifulSoup(s.get(another_url).text, "lxml")
    # ... verarbeite soup
# am Ende des Blocks wird die Session automatisch via s.close() geschlossen
```

- Bietet sich insbesondere für Webseiten an, in welche man sich einloggen muss

Logins und Cookies

Dateidownload

Dateidownload

Prototyp einer Lösung mit Hilfe von `shutil`:

```python
import shutil


def download_file(s: requests.session, url: str, filename: str) -> None:
    req = s.get(url, stream=True)
    if req.status_code == 200:
        # Öffne Datei im Binärmodus ('b') zum Schreiben ('w')
        with open(filename, "wb") as f:
            # Schreibe die rohen Bytes in ein Fileobjekt
            shutil.copyfileobj(req.raw, f)
```

- `req.raw` enthält die unkodierten Antwortbytes der `GET`-Anfrage.
- `stream=True` verhindert, dass die Datei zuerst in den Arbeitsspeicher geladen wird.