Python Web Scraping

Jonathan Helgert
jhelgert@mail.uni-mannheim.de

27. Mai 2021

https://github.com/jhelgert/web_scraping_einfuehrung

Intro

Web Scraping Libraries

Es werden hauptsächlich drei Libraries verwendet:

- **Selenium**: Eine Automatisierungslibrary, mit der man das Browserfenster beliebig steuern kann. Extrem mächtig. Für WebScraping mMn aber zu mühselig.
- **Scrapy**: Ähnlich zu BeautifulSoup, aber eher für Massenscraping von mehreren Seiten ausgelegt. Etwas mächtiger, aber umständlicher.
- **BeautifulSoup**: Eine elegante und extrem verbreitete Library, welche einfache Funktionen bietet um innerhalb eines HTML Dokuments zu Suchen und Extrahieren.

Beispielprojekte in der Wildnis:

- SpiegelMining – Reverse Engineering von Spiegel-Online (33c3)
  `https://youtu.be/-YpwsdRKt8Q`
- `https://github.com/iamnotturner/vaccipy`
- `https://github.com/jhelgert/iliasDownloaderUniMA`

Let’s get started

- BeautifulSoup selbst kann keine Dateien herunterladen oder ähnliches, es kann lediglich HTML Code durchsuchen.
- Für den Webzugriff verwenden wir die `requests` library.
- BeautifulSoup selbst benötigt nur einen XML/HTML Parser (wir verwenden `lxml`)
- BeautifulSoup und requests können lediglich statische Webseiten laden, d.h. Daten, welche z.B. mit Javascript erzeugt werden, können **nicht** *geparsed* werden.

Requests

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

- Wir verwenden die Library (vorerst) ausschließlich um den HTML Code einer Webseite zu erhalten.
- Jede HTTP-Anfrage gibt vom Server einen Statuscode zurück. Häufige Statuscodes: `200` OK, `401` UNAUTHORIZED, `403` FORBIDDEN, `404` NOT FOUND.
- Zahlreiche weitere Features: *Sessions with Cookie Persistence*, *Basic/Digest Authentication*, *HTTP(S) Proxy Support*, *POST Multiple Multipart-Encoded Files*.

Beispiel:

```
from requests import get

url = "https://www.google.de"

if (res := get(url)):
    print(res.text)     # Alles geklappt. Gib das HTML-Dokument als text aus
else:
    print(res.status_code) # Falls es Probleme gab, schaue den Status Code an
```

Requests

- Da Web Scraping nicht immer erwünscht bzw. geduldet ist, gibt es gelegentlich einige (absichtliche) Hürden.
- Bei jeder HTTP-Anfrage werden auch Metadaten (die *headers*) abgefragt, anhand derer es für einen Admin möglich ist, Tools zu erkennen und die Anfrage zu blocken.

Beispiel:

```
from requests import get

url = "https://www.the-future-of-commerce.com/2020/03/20/brands-with-the-best-customer-service/"
print(get(url).status_code) # 403 Forbidden :(
```

Was nun?

- Die `GET`-Anfrage sollte möglichst wie die eines Browsers *aussehen*.
- Das kann unter anderem durch die *headers* erreicht werden.

HTML Basics

HTML Basics

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

Wichtige Tags

- `p` erzeugt einen Textabsatz mit Zeilenumbruch
- `a` Link bzw. Anker, wobei Text ohne Zeilenumbruch (mit `href` Attribut)
- `div` ist ein *block tag*. Fasst mehrere Elemente zu Gruppen zusammen.
- `table` HTML-Tabelle.
- `img` Bild.

Wichtige Attribute

- `class` CSS-Eigenschaften aus der CSS-Datei oder einem style-Tag.
- `id` identifziert ein HTML-Tag eindeutig.
- `style` Inline CSS-Code für das entsprechende HTML-Element.
- `href` enthält den Ziellink.
- `src` enthält den Link zu einer Bilddatei (`img`-Tag)

BeautifulSoup

BeautifulSoup

Ausgangspunkt ist immer eine Instanz der BeautifulSoup Klasse, deren Konstruktor den HTML Code als String und einen HTML Parser als Argument erwartet:

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

Wichtige Klassenobjekte:

- `bs4.BeautifulSoup` enthält das komplette HTML Dokument, das wir verarbeiten möchten. Verhält sich ähnlich zu `bs4.element.Tag` und unterstützt viele der Methoden.
- `bs4.element.Tag` entspricht einem HTML Tag des HTML Dokuments. Bietet u.A. Methoden, die direkt auf Attribute des Tags zuzugreifen und auch einige nützliche Iteratoren für den Zugriff auf ineinander verschachtelte Tags.
- Es gibt noch `bs4.element.NavigableString` und `bs4.element.Comment`, welche wir aber nicht behandeln.

bs4.element.BeautifulSoup

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

Beispiel:

```
soup = BeautifulSoup("""<html><body>
  <head><h1>Überschrift</h1></head>
  <p>Das hier ist ein Absatz.</p></body>
  </html>""", "lxml")

# Man kann direkt auf die entsprechenden Tags zugreifen:
print(soup.body)         # <body><h1>Überschrift</h1><p>Das hier ist ein Absatz.</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: Hierbei wird stets nur der erste Tag gewählt, der zutrifft.

bs4.element.Tag

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

```
soup = BeautifulSoup("<a class='class1 class2' href='http://example.de'>Klick mich!</a>", "lxml")
tag = soup.a     # type(tag) = bs4.element.Tag

# Tagobjektattribute
print(tag.name)  # "a"
print(tag.attrs) # {'class' : ['class1', 'class2'], 'href' : 'http://example.de'}
print(tag.text)  # "Klick mich!"

# Check auf Tagattribute
print(tag.has_attr("class")) # True
print(tag.has_attr("href"))  # True
print(tag.has_attr("foo"))   # False

# Zugriff auf Tagattribute
print(tag["class"]) # ['class1', 'class2']
print(tag["href"])  # "http://example.de"
```

`bs4.element.Tag`

Die Klasse bietet verschiedene Iteratoren um innerhalb des HTML Dokuments zu navigieren:

- `.children` liefert alle direkten Nachfolgertags innerhalb inseres Tags
- `.descendants` enthält auch die (rekursiven) Nachfolgertags der children.
- `.parents` enthält die (rekursiven) Vorgängertags, in welchen der aktuelle Tag enthalten ist.
- `.next_siblings` enthält alle nachfolgenden Nachbartags des gleichen Levels innerhalb des HTML Dokuments, d.h. alle Tags mit den selben `.parents`.
- `.previous_siblings` analog zu `.next_siblings`, allerdings der vorherfolgenden Nachbartags.
- `.next_elements` ähnlich zu `.next_siblings`, enthält allerdings auch die rekursiven children jedes Nachbartags!
- `.previous_elements` analog zu `.next_elements`, allerdings mit vorherfolgenden Nachbartags und rekursiven children.

Analog zu den Iteratoren gibt es auch `.parent`, `.next_sibling`, `.previous_sibling`, `.next_element`, `.previous_element`.

`bs4.element.Tag.children` und `bs4.element.Tag.descendants`

Beispiel:

```
soup = BeautifulSoup("""<div><a class='c1'>A</a>
<a class='c2'><span>S</span></a>
</div>""", "lxml")
tag = soup.div  # type(tag) = bs4.element.Tag

# Liefert alle direkten (!) Nachfolgertags/"children" unseres Tags
for child in tag.children:
  print(child)            # [<a class="c1">A</a>, <a class="c2"><span>S</span></a>]

print(list(tag.children)) # [<a class="c1">A</a>, <a class="c2"><span>S</span></a>]
print(tag.contents)       # äquivalent zu list(tag.children)

# descendants = Enthält auch die (rekursiven) children der children:
for d in tag.descendants:
  print(d) # [<a class="c1">A</a>, "A", <a class="c2"><span>S</span></a>, <span>S</span>, "S"]

# Analog zu list(tag.children)
print(list(tag.descendants))
```

bs4.element.Tag.parents, siblings und elements

Beispiel:

```
soup = BeautifulSoup("<html><body><a><b>text1</b><c>text2</c></a></body></html>", 'lxml')
print(soup.prettify())
#  <a>
#   <b>
#    text1
#   </b>
#   <c>
#    text2
#   </c>
#  </a>

print(soub.b.parent)                  # <a><b>text1</b><c>text2</c></b></a>
print(list(soup.b.next_siblings))     # [<c>text2</c>]
print(soup.b.next_sibling)            # <c>text2</c>
print(list(soup.c.previous_siblings)) # [<b>text1</b>]
print(soup.c.previous_sibling)        # <b>text1</b>
print(list(soup.b.next_elements))     # ['text1', <c>text2</c>, 'text2']
print(list(soup.c.previous_elements)) # ['text1', <b>text1</b>, <a><b>text1</b><c>text2</c></a>, ...]
```

`find()` und `find_all()`

Mit Hilfe von `find_all(name, attrs={}, recursive=True, string=None, limit=None, **kwargs)` können wir eine Liste von Tags basierend auf einem Filter für bestimmte Eigenschaften bzw. Attribute finden.

- Die Methode `find_all()` sucht rekursiv innerhalb der Nachkommen (*descendants*) eines Tags und gibt alle zurück, die unseren Filter erfüllen.
- Die Methode gibt stets eine Liste von Tags zurück.
- Die Methode `find()` ist äquivalent, liefert allerdings nur den ersten gefundenen Tag.

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>""", "lxml")

# 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 Attribute und Attributwerte erfüllen müssen. Beispiele:

```
soup = BeautifulSoup("""<div>
<a class='c1' id="myid1">A</a>"                          <!-- (1) -->
<a class='c2' id="myid2"><span>S</span></a>              <!-- (2) -->
<a class='c2' id="myid3"><span>S2</span></a>             <!-- (3) -->
<a class='c1 c2' id="myid4"><span>ACHTUNG</span></a>     <!-- (4) -->
</div>""", "lxml")


soup.find_all("a", {'class': 'c2'})                 # (2), (3), (4)
soup.find_all("a", {'class': 'c2', 'id': 'myid2'}) # (2)
soup.find_all("a", {'class': 'c1 c2'})              # (4)
soup.find_all("a", {'class': ['c2']})               # (2), (3), (4)
```

Beachte, dass das **class**-Attribut eines HTML Tags mehrere Werte besitzen kann und somit nur einer dieser Werte passen muss.

find_all()

Mit Hilfe von **kwargs können wir Attribute und Attributwerte auch direkt als Filter verwenden, statt als dictionary an attrs.

- Wichtig: Voraussetzung ist, dass der Attributname selbst ein gültiger Pythonvariablenname ist, welcher noch nicht reserviert ist.
- Es gibt daher einige Fälle, in denen man stattdessen attrs verwenden muss.

Beispiele:

```
# Negativbeispiele:
soup.find_all(class = "c1")     # Nicht gültig, da 'class' reserviertes Python keyword
soup.find_all(name = "hallo")   # 'name' ist bereits Argument von find_all
soup.find_all(data-odd = "123") # Nicht gültig, da "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") # Alle a-tags mit "c2" CSS Class
soup.find_all("a", class_="c2", id=True) # Klar
```

Filter im Detail: Lambdas

Wir haben bisher die Argumente der `find_all()` Methode verwendet und simple Strings bzw. Listen übergeben. Zudem können eigene Funktionen und reguläre Ausdrücke als Filter verwendet werden.

- Eine eigene Funktion, welche als einziges Argument ein `bs4.element.Tag` Objekt erwartet und `True` bzw `False` zurückgibt.
- Übergibt man die Funktion als Filter für ein bestimmtes Attribut, ist das erwartete Funktionsargument stattdessen der Attributwert.

```
def exact_class_c1_match(tag):
    return tag.has_attr("class") and tag["class"] == ["c1"]

soup.find_all(exact_class_c1_match)

soup.find_all(lambda tag: tag.has_attr("class") and tag["class"] == ["c1"])
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(lambda tag: len(tag.contents) == 1)
soup.find_all(href = lambda h: h is not None and h.endswith("pdf")) # h ist Attributwert (str)!
```

Filter im Detail: Reguläre Ausdrücke

Reguläre Ausdrücke sind ein extrem mächtiges Werkzeug um Strings nach bestimmten Eigenschaften zu filtern. Vorteil: Die Syntax ist unabhängig von der Programmiersprache. Nachteil: Zu Beginn etwas verwirrend. Beispiele:

```
import re # regular expressions

#  *    matched das vorherige Zeichen bzw. den Teilausdruck 0 or mehrmals
#  +    matched das vorherige Zeichen bzw. den Teilausdrick min. einmal
#  .    matched jedes beliebige Zeichen
# [A-Z] matched jedes jedes Zeichen von A bis Z
# .*    matched jedes beliebige Zeichen
#  $    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]"))                   # Alle h1, h2, h3, ..., h9 Tags
soup.find_all("a", class_=re.compile(r"c"))             # Jede CSS Klasse, die c enthält
soup.find_all("a", href=re.compile(r"pdf$"))            # Alle .pdf Links
soup.find_all("a", href=re.compile(r".*uebung.*.pdf$")) # Alle .pdf links, mit "uebung" im Namen
```

CSS Selektoren

Sowohl `bs4.element.BeautifulSoup` als auch `bs4.element.Tag` besitzen eine `.select()` Methode um sogenannte *CSS Selektoren* zu verwenden.

- Die `.select_one()` Methode funktioniert gleich, gibt aber nur den ersten Treffer zurück.
- Der entscheidende Vorteil der CSS Selektoren ist, dass diese bequem vom Browser bereitgestellt werden.

Beispiele:

```
# CSS Klassen
soup.select(".c1")      # = soup.find_all(class_="c1")
soup.select(".c1, .c2") # = soup.find_all(class_=["c1", "c2"])

# Attribute
soup.select('a[href]')         # = soup.find_all("a", href=True)
soup.select('a[href^="https"') # = soup.find_all("a", href=lambda h: h.startswith("https"))
soup.select('a[href$="pdf"]')  # = soup.find_all("a", href=lambda h: h.endswith("pdf"))

# Verschachtelter Zugriff
soup.select_one("#css-selectors > div:nth-child(6) > div")
```

HTML-Tabellen

HTML-Tabellen einlesen

HTML Tabellen lassen sich mit Hilfe von **pandas** `.read_html()` Methode sehr elegant und einfach einlesen.

- Die Funktion akzeptiert u.a. das HTML Dokument als String.
- Gibt eine Liste von DataFrames zurück, in welchen alle HTML `table` eingelesen werden.

Beispiel:

```
import pandas as pd

soup = BeautifulSoup(get("https://www.kicker.de/bundesliga/tabelle").content, "lxml")
table = soup.find("table")
df = pd.read_html(str(table))[0] # Extrahiere die erste (und einzige) Tabelle
```

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"/>
<!-- weitere Formularelemente -->
<input type="submit" name="submit" value="Anmelden"/>
</form>
```

- Das `action`-Attribut des `form`-Tags gibt die URL zum *backend script*, an welche die Formdaten via `GET` oder `POST` gesendet werden müssen. Fehlt das Attribut, wird die aktuelle Seite als URL verwendet.
- Die `input`-Tags sind die jeweiligen Eingabefelder.
- Das `type`-Attribut legt den Typ fest, d.h. ob Text, Zahl, Telefonnummer, Emailadresse oder z.b. unsichtbar (`hidden`).
- Das `name`-Attribut legt den Namen der Eingabe des Eingabeelements fest.
- Das `value`-Attribut nach Ausfüllen des Formulars die Eingabewerte jedes Eingabefeldes (kann auch bereits ausgefüllt sein).

HTML-Forms

Das Ausfüllen einer HTML Form ist dank der **`requests`** Library extrem simpel:

```
import requests

# 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)
```

- Man betrachtet die HTML-Form und alle Formfelder und deren Werte, die abgeschickt werden.
- **`action_url`** ist gerade der Wert des **`action`**-Attributs des **`form`**-Tags.
- **`form_data`** ist ein *dictionary*, das die Namen der Eingabewerte und die entsprechenden Eingaben enthält:

  ```
  form_data = { 'first_name' : 'Walter', 'last_name': 'HEISENBERG'}
  ```

- **`headers`** ist ebenfalls ein *dictionary*, das die Metadaten enthält (*User-Agent* usw.)

Logins und Cookies

`requests.session`

Die **`requests`** Library bietet eine **`session`**-Klasse an, welche alle Cookies, Headers und sonstige Metainformationen einer *Session* speichert.

```
from requests import session

with session() as sess:
    # GET bzw. POST
    req1 = sess.get(url, data=form_data, headers=headers)
    req2 = sess.post(url, data=form_data, headers=headers)
    # Jetzt sind wir eingeloggt:
    soup = BeautifulSoup(sess.get(another_url).text, "lxml")
    # ... verarbeite soup
# am Ende des Blocks wird die Session automatisch via sess.close() geschlossen
```

- Cookies sind nichts anderes als ein seitens des Webservers generierter Token, welcher Metadaten enthält (z.B. ob man bereits eingeloggt ist).
- Das sollte stets verwendet werden, sobald man sich in einer Form o.ä. einloggt, damit man nicht die Cookies und andere Metadaten für jede neue HTTP-GET-Anfrage mitschleppen muss.

Dateidownload

Dateidownload

Sobald wir dank BeautifulSoup leicht Links zu Dateien *geparsed* haben, möchten wir diese natürlich herunterladen. Dafür können wir `requests` und `shutil` verwenden:

```
import requests
import shutil

def download_file(sess: requests.session, url: str, filename: str) -> None:
    req = sess.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)
```

- `stream=True` verhindert, dass die Datei zuerst in den Arbeitsspeicher geladen wird.
- `req.raw` enthält die *rohen*, d.h. unkodierten Antwortbytes der HTTP-`GET`-Anfrage. Das ist gerade unsere Datei.