

Scrapy

A Fast and Powerful Scraping and Web Crawling Framework

LdPx, lammbraten, foobar999

12. Juni 2017

Übersicht

Allgemeines

Installation

Projekt anlegen

Spider

Extraktion von Daten

Links folgen

Übung

- Emails - Skynet kann das!

- Blog-Posts

- Link-counter

Fazit

Quellen

Allgemeines

- ▶ Definition Scraping...
- ▶ Was ist Web-Scraping?
- ▶ Framework für Python
- ▶ Crawlen von Websites
- ▶ Extraktion strukturierter Daten

Installation

- ▶ mit pip (erfordert vorhandene Python-Installation):

```
pip3 install scrapy
```

- ▶ mit anaconda (in Windows einfacher, da numpy, etc... bereits enthalten):

```
conda install -c conda-forge scrapy
```

Projekt anlegen

- ▶ Verzeichnisstruktur erzeugen:

```
scrapy startproject mynewproject
```

- ▶ erzeugt ein gleichnamiges Verzeichnis `./mynewproject`
- ▶ künftige Spider werden in `./mynewproject/spiders` angelegt
- ▶ Konfiguration erfolgt über `./mynewproject/settings.py`

Spider

- ▶ spezielle Klassen in Scrapy-Projekten
- ▶ von Klasse `scrapy.Spider` abgeleitet
- ▶ führen Crawling durch, spezifisch für Websites programmierbar
- ▶ Ablauf:
 1. Spider schickt Requests an initiale URLs
 2. scrapy ruft je Response Callback-Methode auf, mit Inhalt als Parameter
 3. Callback-Methode startet ggf. Requests an weitere URLs
 4. Callback-Methode extrahiert Daten des Response und gibt sie zurück
 5. scrapy sammelt alle zurückgegebenen Daten ein, und speichert sie z.B. in einer Datei

einfacher Beispielspider (1)

```
import scrapy

class SimpleSpider(scrapy.Spider):
    name = 'simplespider'
    start_urls = ['http://supersimpleloremipsum.com/']

    def parse(self, response):
        self.logger.info('parsing {}'.format(response))
        yield {'status': response.status}
```

einfacher Beispelspider (2)

- ▶ `name` identifiziert den Spider eindeutig im Projekt
- ▶ `start_urls` beinhaltet die URLs für die initialen Requests
- ▶ `parse()` wird als Callback aufgerufen, der Parameter `response` enthält das Ergebnis des Response
- ▶ die Methode erzeugt eine Logging-Ausgabe von `response`
- ▶ die Methode übergibt scrapy das Parsingergebnis den HTTP-Statuscode als `dict`

einfacher Beispelspider (3)

- ▶ das Crawling mit Spider `simplespider` wird über einen eigenen Befehl in der Kommandozeile gestartet:

```
scrapy crawl simplespider -o res.json
```

- ▶ scrapy speichert die per `yield` zurückgegebenen Daten in der Datei `res.json` (das Dateiformat ergibt sich aus Suffix)
 - ▶ scrapy unterstützt neben json weitere Formate: xml, csv, ...
 - ▶ beachte: scrapy leert bei mehrfachem Start des Crawlingbefehls die Datei nicht, sondern hängt die Daten hinten dran
- ▶ Befehl auch über Pythonskript aufrufbar:

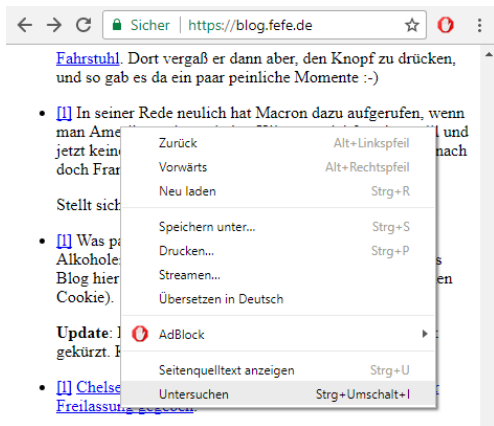
```
def main():  
    cmd = 'scrapy crawl iter -o res.json'  
    scrapy.cmdline.execute(cmd.split())  
main()
```

- ▶ API zum Crawlen: Fehlanzeige ?

Extraktion von Daten (1)

```
def parse(self, resp):
    res1 = resp.css('div small')
    #[<Selector data='<small...">'>, <Selector data='<
        small...">'>, ...]
    res2 = resp.css('div small::text')
    #[<Selector data='Albert Einstein'>, <Selector data
        ='J.K. Rowling'>, ...]
    res3 = resp.css('div small::text').extract()
    #['Albert Einstein', 'J.K. Rowling', ...]
    res4 = resp.css('div small::text').extract_first()
    #Albert Einstein
    ...
```

Extraktion von Daten (2)



Bla Bla Bla Mr. Freeman, Caption.

Extraktion von Daten (3)

- ▶ `resp.css('div small')` selektiert mehrere Elemente im DOM mit dem CSS-Selektor `'div small'` (d.h. es findet alle `<small>`-Elemente innerhalb von `<div>`-Elementen)
- ▶ `resp.css('div small::text')` selektiert den Text innerhalb jedes gefundenen `<small>...</small>`-Tags
- ▶ `resp.css('div small::text').extract()` liefert die selektierten Texte als Liste von Python-Strings
- ▶ `resp.css('div small::text').extract_first()` liefert den 1. Treffer davon

Extraktion von Daten (4)

- ▶ das Ergebnis einer Selektion kann selbst wieder zum Selektieren genutzt werden
- ▶ folgendes Beispiel selektiert alle Zitate einer Website, und gibt je Zitat als Datensatz Autor und Wortlaut zurück:

```
def parse(self, response):  
    quotes = response.css('div.quote')  
    for q in quotes:  
        yield {  
            'author': q.css('.author::text').extract_first(),  
            'text': q.css('.text::text').extract_first()  
        }
```

Links folgen (1)

```
def parse(self, response):
    for q in response.css('div.quote'):
        yield {
            'text': q.css('.text::text').extract_first(),
            'author': q.css('.author::text').extract_first()
        }
    a_selector = 'li.next a::attr(href)'
    hrefs = response.css(a_selector).extract()
    for href in hrefs:
        yield response.follow(href, callback=self.parse)
```

- ▶ die Methode liefert zunächst von den Zitaten Autor und Wortlaut
- ▶ danach sucht sie alle passenden -Elemente und selektiert je Treffer das href-Attribut mit `'a::attr(href)'`

Links folgen (2)

- ▶ zuletzt extrahiert die Methode die enthaltenen URLs und untersucht sie rekursiv mit
`response.follow(href, callback=self.parse)`
- ▶ scrapy merkt sich außerdem intern schon besuchte URLs und vermeidet so mögliche Endlosrekursionen durch wechselseitige URL-Verweise

Übung: Emails - Skynet kann das!

→ *TO.BE.DONE!*

Übung: Blog-Posts

→ *Textebefreien! und ... jawaseigentlich :?*

Übung: Link-counter

→ *ZählenZählenZählen!*

Fazit

Ist Scrapy toll?
Vielleicht!

Quellen

[1] <https://doc.scrapy.org/en/latest/intro/tutorial.html>

[2] <https://doc.scrapy.org/en/latest/intro/install.html>

[3] <https://doc.scrapy.org/en/latest/topics/spiders.html>