WEB SCRAPING FOR PRACTITIONERS

Selected problems with solutions in Python, v0.9

Damian Stelmasiak

Pawel Macias

FOREWORD

Web scraping, which provides an attractive way of obtaining the data, is becoming more and more popular. Especially for statistics, web scraping offers an alternative source of information to survey and scanner data.

New technologies are bringing opportunities but also challenges. Therefore, dear reader, we call for responsibility in all your actions. Web scraping users are obliged to abide by domestic legal situation of web scraping activities. If applicable, we strongly suggest to comply with web-scraping policy guidelines, ie. given by Office for National Statistics [1]. Among others, they are: minimising burden on websites, avoiding scraping areas of websites specified by robots.txt and including your personal information in user-agent string for identification purposes.

Authors of this techbook shall not be liable for any damages exerted by unlawfull or inaccurate applying of the presented codes. Any resemblance to actual websites codes is purely coincidental. The tech presented in this document is a collection of exemplary problems and solutions and does not reflect any specific methodology.

Listings

CHAPTER 1

BASIC APPROACHES AND THEIR COMBINATIONS

First, solve the problem. Then, write the code

—Mohammed Javeed

1.1 Description of basic approaches

To keep it fairly simple, we distinguish 3 web scraping approaches:

1. raw web page source parsing,
2. interacting with DOM objects in web browser,
3. direct downloading already structured data (with or without API).

Every method has some prons and cons. Parsing with HTML tags is pretty fast and it does not usually pose serious difficulties. However, web page source may not contain all of data you are interested in. Web browser offers much more possibilities, but it is mostly slower and may be more error-prone solution than parsing already fetched page source. In the context of the third approach, it should be noted that

most of APIs are non-public. More and more often web sites are built in the way[1] they fetch the data on products on the fly (at the moment when customer accesses the given subpage). You might be able to find appropriate urls that return structured data in JSON format. The third approach may be the most convenient way of obtaining data from web pages, but it requires more effort in locating ”magic” urls. This may fail if a website employs pregenerated pages.

1.2 Page source example

Assume that data of our interest is present in web page source. We download the page source within headless session of Requests library [3] and parse using BeautifulSoup [4].

```
from bs4 import BeautifulSoup as soup
import requests
s = requests.session()

requests_obj = s.get(url_mainpage)
PS = soup(requests_obj.text)

categories = PS.find("nav", {"class": "navigation-wrapper"})
a_tags = categories.findAll("a")
categories_links = [x["href"] for x in a_tags]

results = []
for category_link in categories_links:
     requests_obj = s.get(category_link)
     PS = soup(requests_obj.text)
     Products = PS.findAll("div", {"class": "product-item"})
     for product in Products:
          try:
               id = product.find("div", {"class": "product-id"}).text
               name = product.find("div", {"class": "name"}).text
               price = product.find("span", {"class": "price"}).text
               results.append([id, name, price])
          except:
               print('Problems with parsing encountered')
```

Listing 1.1 Simple example of page source approach

1.3 Web browser example

Similar example of the code above is provided for web browser case. Web browser automation software in general offers much more for advanced users, but for such a simple example the code is straightforward. This aproach is more general as even

[1]This is perhaps not particularly strange as websites are computer programs. They are always executed, but differ in share of pregenerated code and dynamic JS scripts.

very dynamic JavaScript web pages are handled properly. It is the most similar method to the way users experience a website.

For documentation of Selenium library see: [5].

```
from selenium import webdriver
options = webdriver.ChromeOptions()
options.add_argument("--start-maximized")
driver = webdriver.Chrome(chrome_options=options)

driver.get(url_mainpage)
categories = driver.find_element_by_class_name("navigation-wrapper")
a_tags = categories.find_elements_by_tag_name("a")
categories_links = [x.get_attribute('href') for x in a_tags]

results = []
for category_link in categories_links:
     driver.get(category_link)
     Products = driver.find_elements_by_class_name("product-item")
     for product in Products:
          try:
               id = product.find_element_by_class_name("product-id").
    text
               name = product.find_element_by_class_name("name").text
               price = product.find_element_by_class_name("price").
    text
               results.append([id, name, price])
          except:
               print('Problems with parsing encountered')
```

Listing 1.2 Simple example of web browser approach

1.4 Direct approach example

We provide here a simple example of direct approach that employs JSON. Fully functional API that enable to download all of the data at once are very rare. Therefore we show a common scheme, where firstly we find an url to categories JSON, which contains links to all categories in webshop. Then, we iterate over categories and retrieve JSONs with data on product details.`'magic_string'` is a characteristic string, which is often applied, but there is no standard for url construction for querying like that. We refer to Network tab in pupular web browsers, which helps to understand querying system for a particular website.

```
from bs4 import BeautifulSoup as soup
import requests
s = requests.session()
s.headers.update({'accept': 'application/json'})

categories = s.get(urls_categories).json()
categories_links = list(categories["url"])
```

```
results = []
for category_link in categories_links:
     Products_dict = s.get(category_link+"magic_string").json()
     for id, product in Products_dict.items():
          name = product["name"]
          price = product["price"]
          results.append([id, name, price])
```

Listing 1.3 Simple example of direct approach

1.5 Basic combination

If content of your interest is not present in page source returned by requests session, you may use web browser. To save time on searching for tags in DOM, though, it is reasonable to download page source of the rendered site in browser and proceed with ordinary parser.

```
from selenium import webdriver
from bs4 import BeautifulSoup as soup

options = webdriver.ChromeOptions()
options.add_argument("--start-maximized")
driver = webdriver.Chrome(chrome_options=options)

driver.get(url_mainpage)
PS = soup(driver.page_source)

categories = PS.find("nav", {"class": "navigation-wrapper"})
a_tags = categories.findAll("a")
categories_links = [x["href"] for x in a_tags]

results = []
for category_link in categories_links:
     driver.get(category_link)
     PS = soup(driver.page_source)
     Products = PS.findAll("div", {"class": "product-item"})
     for product in Products:
          try:
               id = product.find("div", {"class": "product-id"}).text
               name = product.find("div", {"class": "name"}).text
               price = product.find("span", {"class": "price"}).text
               results.append([id, name, price])
          except:
               print('Problems with parsing encountered')
```

Listing 1.4 Simple example of approach that employs browser combined with bs parser

CHAPTER 2

SELECTED REAL-WORLD PROBLEMS

2.1 Setting products per page

An effective way to scrape is to max out the the number of products per page. Assume that the number of products is being set by visting a specific url, then cookies are generated. For effectiveness we transfer these cookies to headless session of requests library. The last step is optional - we add 'application/json' header, which quite often enables us to obtain JSON output instead of HTML files if web page offers this feature.

```
import requests
from selenium import webdriver
driver = webdriver.Chrome()
s = requests.session()

url_settings = "https://www.XYZXYZXYZX.com/ppp/100"
driver.get(url_settings) # 100 products per page are set now

cookies = driver.get_cookies()
for cookie in cookies:
    s.cookies.set(cookie['name'], cookie['value'])
```

```
driver.quit()

s.headers.update({'accept': 'application/json'})
```

Listing 2.1 Setting products per page and transfering cookies

2.2 Hard-coded JSONs

It is not rare to see a JSON file in web page source code. JSON are mostly embedded in JS script and the data of interest follows var statement. One may easily find, extract and load the JSON as a Python dictionary, as below.

```
import re
import json

pattern = re.compile("var jProductData = ({.*?});", re.DOTALL)
code = pattern.search(page_source_asstring).group(1)
dictionary = json.loads(code)
```

Listing 2.2 Hard-coded JSON extraction

Dictionaries are often multiply nested - simplifying them is, in general, ambiguous. If dictionary is relatively simple one may obtain satisfactory results by flattening with `json_normalize` from pandas, imported as follow: `from pandas.io.json import json_normalize`.

One may come across JSON delimited with non-pythonic brackets [...]. In this case just use `demjson.decode(your_nonpythonic_dict)` and proceed as always.

Please notice that JSONs might include more details about products, i.e. features that are not presented for customers. However, this data may be out-of-date or inaccurate depending on a website.

2.3 Load more products case

Simple solution for web pages where number of products is limited and button for loading more of them is present. The following script simulates scrolling to the bottom of the page and clicking the button for loading more products.

```
while True:
     driver.execute_script("window.scrollTo(0, document.body.
    scrollHeight);")
     time.sleep(2.0)
     next_page_button = driver.find_element_by_xpath("//button[@class
    ='button load-more']")
     try:
          next_page_button.click()
          time.sleep(2.0)
     except:
```

```
            time.sleep(0.5)
            break
```

Listing 2.3 Scroll to bottom and load more products scheme

2.4 Scroll into view case

This is rarely seen - categories tree is generated only if customer scrolled the page to have this tree in view. We may use button for sorting the categories as a waypoint. After all, the page may require scrolling up a bit (by 250 pixels).

```
button = driver.find_element_by_id('sort-categories')
driver.execute_script("arguments[0].scrollIntoView(true);", button)
time.sleep(1)
driver.execute_script("window.scrollBy(0,-250);")
```

Listing 2.4 Scroll into view to have categories tree generated

2.5 "Hidden" objects

Nowadays, a lot of web page content is rendered on the fly not being present in page source. Moreover, some of the elements are hidden from DOM objects inspectors built in popular web browsers. In this case, Network tab is your friend. This built-in feature tracks thoroughly web page scripts execution and may be perceived as debugging tool.

Assume that we are looking for a particular url, which returns shop's JSON categories tree. We know it is variable, but contains a constant substring `'<magic_name>'`. This url does not appear in parsers or Selenium search results. The following function gives a list of all urls generated during web page loading according to the network tab.

```
def network_tab_get(driver):
    scriptToExecute = "var performance = window.performance || window
    .mozPerformance || window.msPerformance || window.
    webkitPerformance || {}; var network = performance.getEntries()
    || {}; return network;"

    network_tab = driver.execute_script(scriptToExecute)
    all_urls =[x['name'] for x in network_tab]

    return all_urls
```

Listing 2.5 Getting a list of all urls from network tab

If you have a full list of urls, you are able to find a desired one, for example: `url_of_your_interest = [x for x in all_urls if '<magic_name>' in x]`.

2.6 Logging in to earn access to more goods

Retailers limit access to some goods, eg. alcoholic beverages. In order to scrape them one needs to create an account and log in. The code below logs in with help of selenium webdriver.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException

def log_in(driver, login, password, wait):
    try:
        element = WebDriverWait(driver, wait).until(
            EC.presence_of_element_located((By.CLASS_NAME, 'login')))
        try:
            driver.find_element(By.CLASS_NAME, "login").click()
        except:
            print("Problem with clicking to log in")
        try:
            element = WebDriverWait(driver, wait).until(
                EC.presence_of_element_located((By.ID, 'Loginform-
    username')))
            try:
                driver.find_element(By.ID, 'Loginform-username').
    send_keys(login)
            except:
                print("Problem with typing the login")
        except TimeoutException:
            print("No login field?")
        try:
            element = WebDriverWait(driver, wait).until(
                EC.presence_of_element_located((By.ID, 'Loginform-
    password')))
            try:
                driver.find_element(By.ID, 'Loginform-password').
    send_keys(password)
            except:
                print("Problem with typing the password")
        except TimeoutException:
            print("No password field?")
        try:
            element = WebDriverWait(driver, wait).until(
                EC.presence_of_element_located((By.XPATH, '//*[@id="
    login-form"]/div/button[@class="button"]')))
            try:
                driver.find_element(By.XPATH, '//*[@id="login-form"]/
    div/button[@class="button"]').click()
            except:
                print("Problem with clicking the button")
        except TimeoutException:
            print("Problem with login button")
    except TimeoutException:
```

```
        print("Timeout")
```

Listing 2.6 Logging in (selenium)

CHAPTER 3

MISCELLANEOUS

Proposition of a setup for Chrome with website-specific data directory.

```
from selenium import webdriver
chromeOptions = webdriver.ChromeOptions()
chromeOptions.add_argument("--user-data-dir="+"your_path"+"/
    profile_chrome/"+"website_name")
chromeOptions.add_argument("--start-maximized")
#chromeOptions.add_argument("--headless")

driver = webdriver.Chrome(executable_path="/chromedriver.exe",
    chrome_options=chromeOptions)
```

Listing 3.1 Chrome driver setup

Proposition of a setup for Firefox in Selenium that doesn't save excessive logs into disk.

```
from selenium import webdriver
import os
```

```
options = webdriver.FirefoxOptions()
options.add_argument("-start-maximized")
fp = webdriver.FirefoxProfile(
        r"location_of_your_profile")
#options.add_argument('-headless')

driver = webdriver.Firefox(executable_path="/geckodriver.exe",
    firefox_options=options,
                               firefox_profile=fp, log_path=os.
    devnull)
```

Listing 3.2 Firefox driver setup

```
import re
price = re.findall("[\d.,]+", prices_string)
```

Listing 3.3 Search for (possible) prices

```
import re
def clean_price(price_string):
     price_number = re.findall("[-+]?(\d+([.,]\d*)?|[.,]\d+)([eE
    ][-+]?\d+)?", price_string)[0][0]
     return price_number
```

Listing 3.4 Cleaning price strings from currencies

There are no ready-made recipes for handling categories. We provide a few examples of structures that are processed with simple nested functions.

```
from collections import OrderedDict
from bs4 import BeautifulSoup as soup

def categories_tree_v1(soup_menu_obj):
    result = OrderedDict()
    for li in soup_menu_obj.find_all("li"):
        key = next(li.stripped_strings)
        HasUL = li.find("ul")
        result[key] = {li.find("a")["href"]} if HasUL is None else
    categories_tree_v1(HasUL)
    return result

def categories_tree_v2(list_of_categories_dicts):
    result = OrderedDict()
    for category_dict in list_of_categories_dicts:
        if 'name' in category_dict:
            name = category_dict['name']
            if 'children' in category_dict and category_dict['
    children']:
                haschild = category_dict['children']
            else:
                haschild = False
            result[name] = {category_dict['url_key']} if not haschild
     else categories_tree_v2(haschild)
```

```
    return result
```

Listing 3.5 Processing categories tree examples

REFERENCES

[1] Greenaway, M. (2018), *ONS Web-scraping policy*, Office for National Statistics.

[2] Macias, P. and Stelmasiak, D. (2018), Food inflation nowcasting with web scraped data.

[3] Requests library documentation, LINK

[4] BeautifulSoup library documentation, LINK

[5] Selenium library documentation, LINK