Intro to Web-Scraping with Python

Tristany Armangué-Jubert

1 APIs

An application programming interface is a way for two or more computer programs to communicate with each other. It is a type of software interface, offering a service to other pieces of software.

Whenever one wants to work with online data, looking for an official API should be the first step.

1.1 Example: FRED

Suppose we want to plot the cyclical components of M1, M2, CPI and GDP for the US and we want to have the ability to update our plot every now and then.

After a brief search, we find that FRED has an API. Moreover, there is a wrapper for Python `https://github.com/mortada/fredapi`.

Every API is different but the public ones generally have good documentation. Following the documentation from the FRED API, we can quickly put together a script to extract the raw data:

Source Code 1: FRED API

```python
# Dependencies
import pandas as pd
from fredapi import Fred

# FRED API wrapper
fred = Fred(api_key='YOUR_API_KEY')

# Names and codes of variables
codes = {
        "M1":"M1SL", "M2":"M2SL",
        "GDP":"GDPC1", "CPI":"CPIAUCSL"
}

# Request first variable
df = fred.get_series(codes["M1"]).to_frame().rename(columns={0:"M1"})

# Request the rest and append
for key in list(codes.keys())[1:]:
        # Pull
        data = fred.get_series(codes[key]).to_frame().rename(columns={0:codes[key]})
        # Merge
        df = df.merge(data, left_index=True, right_index=True, how='outer')

# Show resulting dataframe
print(df)
```

We can use the resulting dataframe to compute the cyclical components using a Hodrick-Prescott filter or a similar technique, which we can then plot using *matplotlib* or *seaborn*.

In general, it would be best practice to wrap the previous script inside a function for reusability.

1.2 Example: ECB

To highlight the significant differences between available APIs, let us look at a different example. Suppose we want to also plot the cyclical components of M1, M2 and M3 for the Euro area. After searching we find the ECB also has an API (`https://data.ecb.europa.eu/help/api/overview`).

After reading through the documentation, we find that we must point requests for data at an entry point (`https://data-api.ecb.europa.eu/service/data/`) with some parameters such as the source of the data (BSI in our case), the name of the variable, and the start and end dates.

In this case, the names of the variables such as M3 for the purpose of the API calls can be found here `https://data.ecb.europa.eu/search-results?searchTerm=M3`.

With this info, we can put together a simple script that requests data from the ECB and constructs a dataframe that we can use later for plotting cyclical components:

Source Code 2: ECB API

```
# Dependencies
import pandas as pd
import requests

# Entry point
url = 'https://data-api.ecb.europa.eu/service/data/'

# Codes for data points
codes = {
        "M1" : "BSI/M.U2.Y.V.M10.X.1.U2.2300.Z01.E",
        "M2" : "BSI/M.U2.Y.V.M20.X.1.U2.2300.Z01.E",
        "M3" : "BSI/M.U2.Y.V.M30.X.1.U2.2300.Z01.E",
}

# Headers for requests
headers = {'Accept':'application/json'}

# Dates of interest
init_date = "1999-01"
end_date = "2023-11"

# Empty dataframe for results
df = pd.DataFrame(columns=["date"])

# Request data points
for key, value in codes.items():
        # List for row of table
        lst2 = [key]

        # Request data
        res = requests.get('{}{}?startPeriod={}&endPeriod={}'.format(url, value, init_date, end_date), headers=headers).json()

        # Extract values
        vals = res["dataSets"][0]["series"]['0:0:0:0:0:0:0:0:0:0:0']['observations']
        keys = list(vals.keys())
        vals = [vals[item][0] for item in keys]

        # Extract dates
        dates = [item["id"] for item in res["structure"]["dimensions"]["observation"][0]["values"]]

        # Make df
```

```
            df_ = pd.DataFrame()
            df_["date"] = dates
            df_[key] = vals

            # Merge into main df
            df = df.merge(df_,on="date",how="outer")

    # Show result
    print(df)
```

2 Unofficial APIs

Web-scraping is a term normally used for cases where there is no API. Still, some websites handle requests internally in ways that closely resemble an API. When looking to programmatically access data from a source that does not provide an official API, the second step should always be to look for an unofficial API.

2.1 A Billion Prices

Suppose that following Cavallo and Rigobon [2016] you wanted to track prices of basket of goods in Spain over time to compute high-frequency inflation numbers. For the sake of exposition let us focus on a subsection of the basket - clothing detergents - and on a single supermarket chain: Carrefour. If you wanted to compute inflation you would have to track prices for products in many other sectors and in as many outlets as possible, as well as figure out a way to weight them.

The naive approach is to navigate to

`https://www.carrefour.es/supermercado/limpieza-y-hogar/cuidado-de-la-ropa/cat20044/c`

every day and write down prices. Obviously that is unfeasible. We will try to reverse engineer the website to obtain access to an unofficial API such that we can set up a script that runs every day automatically and appends the new prices into our dataset. Opening up the url in Chrome we see something like this:

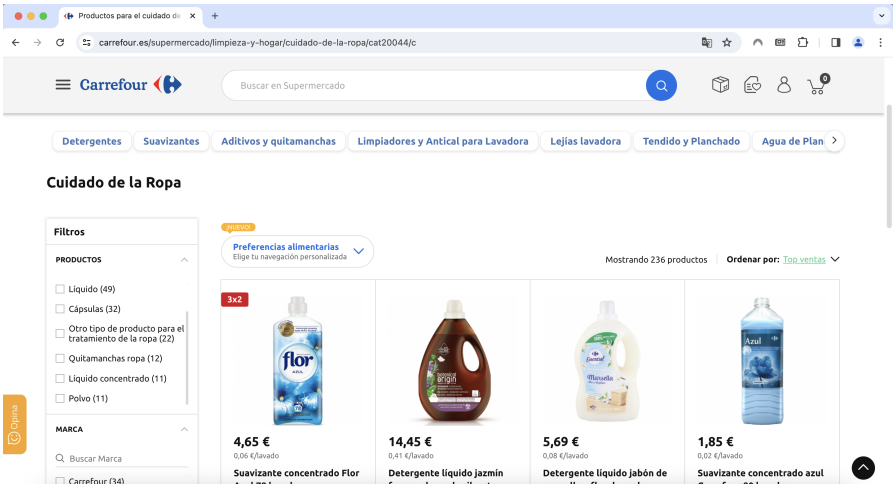

Figure 1: The landing page in Carrefour's section for clothing detergents

Right-clicking anywhere on the page and selecting inspect opens up the following dialog:

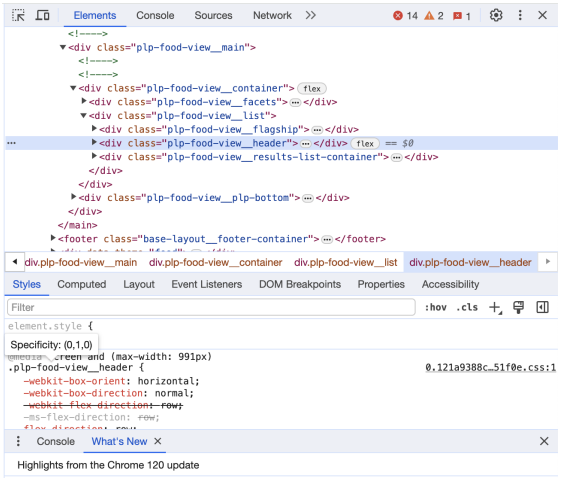

Figure 2: The inspect element in Google Chrome

If we choose the Network tab we can see how our browser is interacting with Carrefour's servers as we navigate through the website. If we now reload the page and head to the Fetch/XHR tab within the Network section we can see the requests we are sending and the responses by the server:

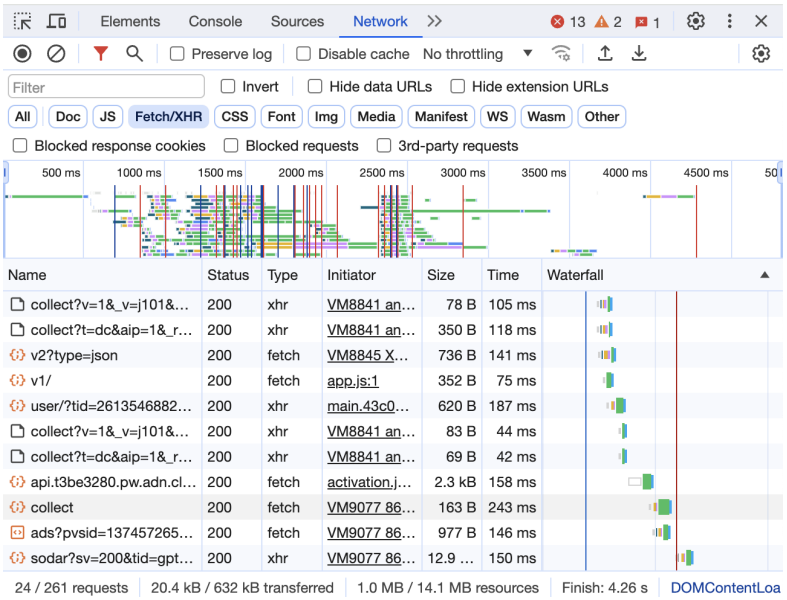

Figure 3: The requests executed when reloading the page from Figure 1

Clicking on the individual requests we can see the headers we sent and the objects the website returned. At this stage we are after a returned JSON with a list of the products available. In this current case I could not find it, so I hit sort by price and looked at the new requests:

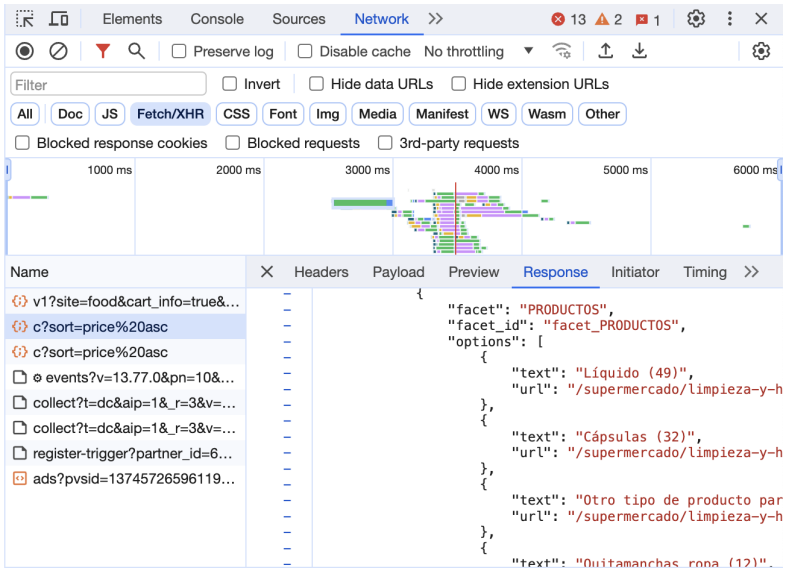

Figure 4: The requests executed when sorting the page from Figure 1 by price

Now, we can see that the second request returned a JSON with what looks like a list of products. Great! If we right-click on it we can copy the request. A good way of doing this is to copy as cURL. Once we have copied the cURL, we can navigate to `https://curlconverter.com` paste it and get back a Python requests snippet:

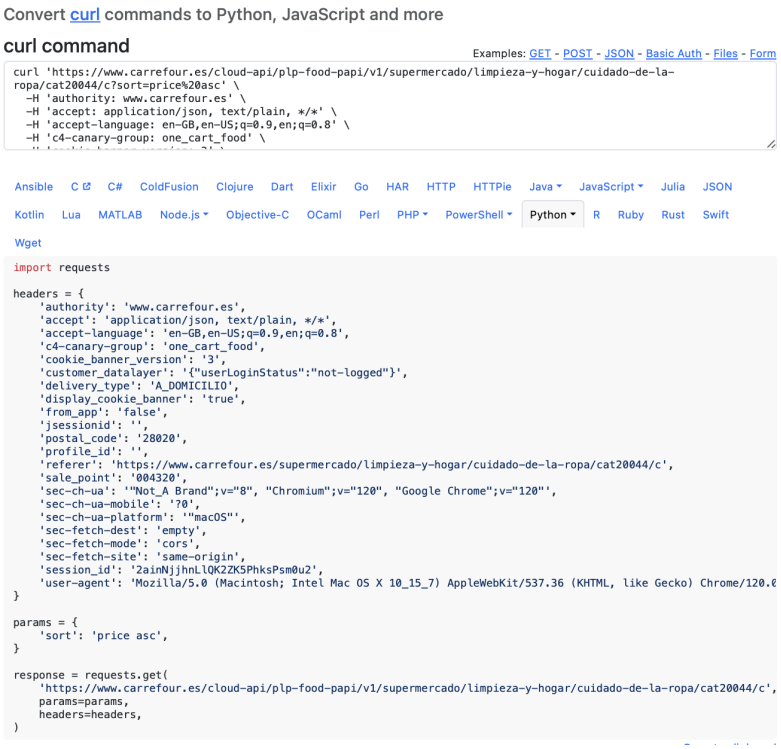

Figure 5: The requests executed when sorting the page from Figure 1 by price

Now we have what we need to write a Python script that we can run at any time and get the prices of those products:

Source Code 3: Carrefour Unofficial API

```python
# Dependencies
import pandas as pd
import requests
import json

# Paste headers we got from curlconverter
headers = {
        'authority': 'www.carrefour.es',
        'accept': 'application/json, text/plain, */*',
        'accept-language': 'en-GB,en-US;q=0.9,en;q=0.8',
        'c4-canary-group': 'one_cart_food',
        'cookie_banner_version': '3',
        'customer_datalayer': '{"userLoginStatus":"not-logged"}',
        'delivery_type': 'A_DOMICILIO',
        'display_cookie_banner': 'true',
        'from_app': 'false',
        'jsessionid': '',
        'postal_code': '28020',
        'profile_id': '',
        'referer': 'https://www.carrefour.es/supermercado/limpieza-y-hogar/cuidado-de-la-ropa/cat20044/c',
        'sale_point': '004320',
        'sec-ch-ua': '"Not_A Brand";v="8", "Chromium";v="120", "Google Chrome";v="120"',
```

```
        'sec-ch-ua-mobile': '?0',
        'sec-ch-ua-platform': '"macOS"',
        'sec-fetch-dest': 'empty',
        'sec-fetch-mode': 'cors',
        'sec-fetch-site': 'same-origin',
        'session_id': '2ainNjjhnLlQK2ZK5PhksPsm0u2',
        'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36',
}

# Request first page
response = requests.get(
        'https://www.carrefour.es/cloud-api/plp-food-papi/v1/supermercado/limpieza-y-hogar/cuidado-de-la-ropa/cat20044/c',
        params={'sort': 'price asc',},
        headers=headers,
)

# Parse html to json
res = json.loads(response.text)

# List for info we will extract
lst = []

# Iterate results
for item in res["results"]["items"]:
        # Save what we need
        lst.append(
                [item["price"], item["price_per_unit"], item["measure_unit"],
                item["name"], item["sku_id"]]
        )

# Make a dataframe from the results
df = pd.DataFrame(lst, columns=["price","price_pu","measure_unit","item_name","item_id"])

# Request subsequent pages
i = 24
while True:
        try:
                # Request
                response = requests.get(
                        'https://www.carrefour.es/cloud-api/plp-food-papi/v1/supermercado/limpieza-y-hogar/cuidado-de-la-ropa/cat20044/c',
                        params={'sort': 'price asc','offset': str(i)},
                        headers=headers,
                )
                # Parse html to json
                res = json.loads(response.text)
                # List for info we will extract
                lst = []
                # Iterate results
                for item in res["results"]["items"]:
                        # Save what we need
                        lst.append(
                                [item["price"], item["price_per_unit"], item["measure_unit"],
                                item["name"], item["sku_id"]]
                        )
                # Make a dataframe from the results
                df_ = pd.DataFrame(lst, columns=["price","price_pu","measure_unit","item_name","item_id"])
                # Concat into main dataframe
                df = pd.concat([df,df_])
                # Update offset
                i = i + 24
        except KeyError:
                break

# Show the result
print(df)
```

Let's break this script up:

- The first commented block sets up the headers as we got them from curlconverter.
- The second block requests the webpage (we got this from curlconverter too).
- The third block parses the response (which is a string JSON) into a dictionary.
- The fourth block makes an empty list for results.
- The fifth block iterates the entries in the JSON (we find these by reading the

JSON) and extracts the needed info.

- The next block makes a dataframe from the blocks.
- The rest repeats the above but changes the offset parameter to iterate pages (you find this by looking at the requests when you click on "next page"), it concatenates the resulting dataframe to the master one. This block catches the exception that rises when there are no more items and stops.

The code above returns a dataframe with 236 rows, one for each product that the URL we looked at showed. This can easily be wrapped in a function that can be run to collect prices of those products at any point in time.

3 Scrapping HTML

Sometimes we are just after HTML tables. Suppose we have found a table of metropolitan areas in Spain by population that we want to use for something (`https://en.wikipedia.org/wiki/List_of_metropolitan_areas_in_Spain`).

As before, we first navigate to the website in Chrome and inspect it. This time we look at the Elements tab rather than the Network tab. As you scroll through the elements in the inspect tab they will highlight in the actual page. We are after the elements that make up the table of interest:

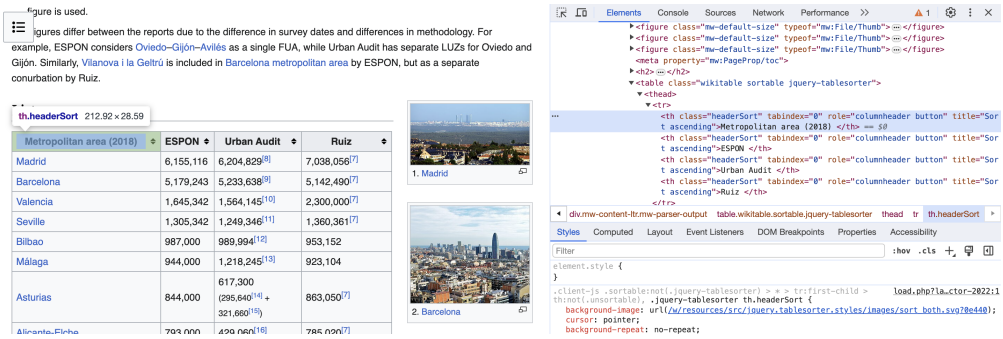

Figure 6: The inspect element and highlighted part of table

The highlighted element in the inspect panel corresponds to the highlighted element on the left.

The following script extracts this table:

Source Code 4: HTML Table

```
# Dependencies
import pandas as pd
import requests
from bs4 import BeautifulSoup

# Request the webpage
res = requests.get("https://en.wikipedia.org/wiki/List_of_metropolitan_areas_in_Spain")
```

```
# Parse data from the html into a beautifulsoup object
soup = BeautifulSoup(res.text, 'html.parser')
table = soup.find('table', {'class':"wikitable"})

# Convert to pandas
df = pd.DataFrame(pd.read_html(str(table))[0])
print(df.head())
```

Let's break this script up:

- The first commented block requests the webpage where the HTML table is located.
- The second block parses the HTML that the server returned using BeautifulSoup, and finds the element table within it.
- The final block makes a dataframe from the HTML table that we found using BeautifulSoup.

BeautifulSoup and similar packages such as lxml can be used in this fashion to extract any info contained in HTML pages.

4 Selenium

Sometimes things are organized in stupid ways. Selenium is a way to automate an actual browser, it can simulate clicks and keystrokes and allows you to navigate very quickly as if you were manually navigating a website.

Suppose you wanted to download rainfall raster data for Africa from CHIRPS (`https://www.chc.ucsb.edu/data`) to then compute a daily-village-level rainfall panel. If you navigate to the data repository, at `https://data.chc.ucsb.edu/products/CHIRPS-2.0/africa_daily/tifs/p05/`, you will see that the .tif files are organized by year and in individual files. If you had to download daily data from 1980 to 2023 manually you would have to click on $\approx$ 15700 links. You can easily automate this using selenium. Here is a function that downloads all days for a given year:

Source Code 5: Selenium

```
# Dependencies
import selenium
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from time import sleep

# Function to download daily data
def download_raw(year):
        # Options - Download directory
        chrome_options = webdriver.ChromeOptions()
        prefs = {'download.default_directory' :
                '/Users/tristany/Library/CloudStorage/OneDrive-UAB/IDEA/Year 2/Python Brush Up/WEB SCRAPING/TIFs'}
        chrome_options.add_experimental_option('prefs', prefs)
        chrome_options.add_argument("--headless=new")
        chrome_options.binary_location = "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome"

        # Start selenium
        driver = webdriver.Chrome(service=Service(ChromeDriverManager().install()), options=chrome_options)

        # URL
        url = "https://data.chc.ucsb.edu/products/CHIRPS-2.0/africa_daily/tifs/p05/{}".format(year)

        # Open URL
```

```
        driver.get(url)

        # Iterate files
        idx = 4
        ok = True
        while ok:
                try:
                        link = driver.find_element(By.XPATH, '//*[@id="indexlist"]/tbody/tr[{}]/td[2]/a'.format(idx))
                        link.click()
                        idx += 1
                except selenium.common.exceptions.NoSuchElementException:
                        ok = False
                        sleep(10)

                sleep(0.4)

# Run the function
download_raw(1981)
```

Let's break this function up:

- The first commented section sets the download directory for Selenium, if this isn't done the downloaded files will be in your Downloads folder. It also sets Selenium to run headless, so that a Chrome window does not open. Finally it points to the Chrome executable.
- The second section starts up selenium, automatically installing/updating Chromedriver.
- The third one sets the URL.
- The fourth requests the URL.
- The final block iterates the elements by XPATH (this is found by inspecting elements as in the previous sections).

References

Alberto Cavallo and Roberto Rigobon. The billion prices project: Using online prices for measurement and research. *Journal of Economic Perspectives*, 30(2):151–178, 2016.