8 Webscraping

8.1 Introduction

This document demonstrates the use of the `BeautifulSoup` library in Python to do webscraping.

Web scraping is the process of gathering information from the Internet. Even copying and pasting the lyrics of your favorite song is a form of web scraping! However, the words "web scraping" usually refer to a process that involves automation. Some websites don't like it when automatic scrapers gather their data, while others don't mind.

If you're scraping a page respectfully for educational purposes, then you're unlikely to have any problems. Sill, it's a good idea to do some research on your own and make sure that you're not violating any Terms of Service before you start a large-scale project.

ⓘ Note

If you do not have the `beautifulsoup4` library installed then you will need to run

`pip install beautifulsoup4`

in the Jupyter/Colab terminal to install. **Remember:** you only need to install once per machine (or Colab session).

```
import pandas as pd
```

8.2 An Alternative to Web Scraping: APIs

Some website providers offer application programming interfaces (APIs) that allow you to access their data in a predefined manner. With APIs, you can avoid parsing HTML. Instead, you can access the data directly using formats like **JSON** and **XML**. **HTML** is primarily a way to present content to users visually.

When you use an API, the process is generally more stable than gathering the data through web scraping. That's because developers create APIs to be consumed by programs rather than by human eyes.

8.2.1 The Tasty API

Check In

Tasty.co is a website and app that offers food recipes. They have made these recipes available through a REST API. You do need authentication, though.

Specifically, you will need to sign up for free account. You will then be provided with an API key that will need to be supplied with every request you make. This is used to track and limit usage.

For our example, we'll us the recipes/list endpoint.

Make sure you are logged into the account you are created, and select the recipes/list endpoint from the menu at the left of the documentation here. We'll use the `requests` Python library to make use of this.

Let's search for recipes containing "daikon" (an Asian radish). Which one is the cheapest per portion?

```
import requests

url = "https://tasty.p.rapidapi.com/recipes/list"

querystring = {"from":"0","size":"20","q":"daikon"}

headers = {
    "X-RapidAPI-Key": <your key here>,
    "X-RapidAPI-Host": "tasty.p.rapidapi.com"
}

response = requests.get(url, headers=headers, params=querystring)

print(response.json())
```

Notice that there are two elements to this object we got back, and we only really want the `results` piece.

```
daikon_recipes = pd.json_normalize(response.json(), "results")
daikon_recipes
```

Check In

With the 2-3 people around you, look up what the JSON format is and what it looks like. Then look up the `json_normalize` function and discuss the differences between the results of this and the JSON format.

The JSON format is extremely common, but can be somewhat easily worked with because it's highly structured.

Before we move on from APIs it's again important to note that gathering data from websites often comes with constraints. For example, the Tasty API only returns 20 results by default and only 40 results maximum, even if you specify the `size=` parameter. So, to gather more than 40 results we might need to use some form of iteration while still respecting the API's rate limits.

8.3 HTML and Web Scraping

HTML, which stands for "hypertext markup language", is an XML-like language for specifying the appearance of web pages. Each tag in HTML corresponds to a specific page element. For example:

- `<img>` specifies an image. The path to the image file is specified in the `src=` attribute.
- `<a>` specifies a hyperlink. The text enclosed between `<a>` and `</a>` is the text of the link that appears, while the URL is specified in the `href=` attribute of the tag.
- `<table>` specifies a table. The rows of the table are specified by `<tr>` tags nested inside the `<table>` tag, while the cells in each row are specified by `<td>` tages nested inside each `<tr>` tag.

Our goal is not to teach you HTML to make a web page. You will learn just enough HTML to be able to scrape data programmatically from a web page.

8.3.1 Inspecting HTML Source Code

Suppose we want to scrape faculty information from the Cal Poly Statistics Department directory. Once we have identified a web page that we want to scrape, the next step is to study the HTML source code. All web browsers have a “View Source” or “Page Source” feature that will display the HTML source of a web page.

Check In

Visit the web page above, and view the HTML source of that page. (You may have to search online to figure out how to view the page source in your favorite browser.) Scroll down until you find the HTML code for the table containing information about the name, office, phone, e-mail, and office hours of the faculty members.

Notice how difficult it can be to find a page element in the HTML source. Many browsers allow you to right-click on a page element and jump to the part of the HTML source corresponding to that element.

8.3.2 Scraping an HTML Table with `pandas`

The `pandas` command `read_html` can be used to scrape information from an HTML table on a webpage.

We can call `read_html` on the URL.

```
pd.read_html("https://en.wikipedia.org/wiki/List_of_United_States_cities_by_popul
```

However, this scrapes all the tables on the webpage, not just the one we want. As we will see with Beautiful Soup, we can narrow the search by specifying the table attributes.

```
pd.read_html("https://en.wikipedia.org/wiki/List_of_United_States_cities_by_popul
```

Check In

Where did the `attrs` details in the code above come from? With the 2-3 people around you, inspect the HTML source code for this page and see if you can identify this.

This still returns 3 tables. The table that we want is the first one!

```
df_cities2 = pd.read_html("https://en.wikipedia.org/wiki/List_of_United_States_ci
df_cities2
```

This is a good first pass at scraping information from a webpage and it returns it to us well in the form of a data frame. This works well for HTML *tables*. Unfortunately, you often want to scrape information from a webpage that isn't conveniently stored in an HTML table, in which case `read_html` won't work. (It only searches for `<table>`, `<th>`, `<tr>`, and `<td>` tags, but there are many other HTML tags.)

8.3.3 Web Scraping Using `BeautifulSoup`

`BeautifulSoup` is a Python library that makes it easy to navigate an HTML document. Like with XML, we can query tags by name or attribute, and we can narrow our search to the ancestors and descendants of specific tags. Also, many web sites have malformed HTML, which `BeautifulSoup` is able to handle gracefully.

First we issue an HTTP request to the URL to get the HTML source code.

```
import requests
response = requests.get("https://statistics.calpoly.edu/content/directory")
```

The HTML source is stored in the `.content` attribute of the response object. We pass this HTML source into `BeautifulSoup` to obtain a tree-like representation of the HTML document.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(response.content, "html.parser")
```

8.3.3.1 Find Elements by ID

In an HTML web page, every element can have an `id` attribute assigned. As the name already suggests, that `id` attribute makes the element uniquely identifiable on the page. You can begin to parse your page by selecting a specific element by its ID.

Check In

Visit the web page above, and view the HTML source of that page. Locate an element of interest and identify its `id` attribute. Then run the following to try extracting it

```
results = soup.find(id="your id here")
print(results.prettify())
```

8.3.3.2 Find Elements by HTML Class Name

You will often see that every similar piece of a web page is wrapped in the same HTML element, like `<div>` with a particular class. We're able to extract all of the parts of the HTML of interest to us, also, by specifying a *containing* HTML element and the specific classes we want with code like the following:

```
results.find_all("element name", class_="class name")
```

8.3.3.3 Find Elements by Class Name and Text Content

It's often the case that we only want pieces of a web page's content that match certain criteria. We can refine our search using the `string` option of `.find_all()`.

```
refined_results = results.find_all("class name", string="search text")
```

This code finds all `class name` elements where the contained string matches `"search text"` **exactly**. We'll discuss later how to be more robust in the specification of string matches like this!

8.3.3.4 Find Elements by Tag

Now we can search for tags within this HTML document, using functions like `.find_all()`. For example, we can find all tables on this page.

```
tables = soup.find_all("table")
len(tables)
```

As a visual inspection of the web page would confirm, there are 3 tables on the page (chair and staff, faculty, emeritus faculty), and we are interested in the second one (for faculty).

```
table = tables[1]
table
```

There is one faculty member per row (`<tr>`), except for the first row, which is the header. We iterate over all rows except for the first, extracting the information about each faculty to append to `rows`, which we will eventually turn into a `DataFrame`. As you read the code below, refer to the HTML source above, so that you understand what each line is doing.

> **Note**
>
> You are encouraged to add `print()` statements inside the `for` loop to check your understanding of each line of code.

```
# initialize an empty list
rows = []

# iterate over all rows in the faculty table
for faculty in table.find_all("tr")[1:]:

    # Get all the cells (<td>) in the row.
    cells = faculty.find_all("td")

    # The information we need is the text between tags.

    # Find the the name of the faculty in cell[0]
    # which for most faculty is contained in the <strong> tag
    name_tag = cells[0].find("strong") or cells[0]
    name = name_tag.text

    # Find the office of the faculty in cell[1]
    # which for most faculty is contained in the <a> tag
    link = cells[1].find("a") or cells[1]
    office = link.text

    # Find the email of the faculty in cell[3]
    # which for most faculty is contained in the <a> tag
    email_tag = cells[3].find("a") or cells[3]
    email = email_tag.text

    # Append this data.
    rows.append({
        "name": name,
        "office": office,
        "email": email
    })
```

Check In

With the 2-3 people around you do the following referring to the code above:

1. What does `cells` look like for the first faculty (i.e. first iteration of the loop)?
2. Why do we need to call `.find("strong")` within `cells[0]` (i.e. a `<td>` tag) to get the name of the faculty member?
3. Could you extract the phone information for each faculty as well? If so, then add the code necessary to do this.

In the code above, observe that `.find_all()` returns a list with all matching tags, while `.find()` returns only the first matching tag. If no matching tags are found, then `.find_all()` will return an empty list `[]`, while `.find()` will return `None`.

Finally, we turn `rows` into a `DataFrame`.

```
pd.DataFrame(rows)
```

Now this data is ready for further processing.

Practice Activity

Practice Activity notebook here.

Edit this page