6 Web Scraping II: Advanced Web Scraping Techniques

Lab Objective: *Gathering data from the internet often requires information from several web pages. In this lab, we present two methods for crawling through multiple web pages without violating copyright laws or straining the load a server. We also demonstrate how to scrape data from asynchronously loaded web pages, and how to interact programmatically with web pages when needed.*

Scraping Etiquette

There are two main ways that web scraping can be problematic for a website owner. First, if the scraper doesn't respect the website's terms and conditions or gathers private or proprietary data. Second, if the scraper imposes too much extra server load by making requests too often or in quick succession. These are extremely important considerations in any web scraping program.

Achtung!

Scraping copyrighted information without the consent of the copyright owner can have severe legal consequences. Many websites, in their terms and conditions, prohibit scraping parts or all of the site. Websites that do allow scraping usually have a file called `robots.txt` (for example, `www.google.com/robots.txt`) that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.[a]

Be careful and considerate when doing any sort of scraping, and take care when writing and testing code to avoid unintended behavior. It is up to the programmer to create a scraper that respects the rules found in the terms and conditions and in `robots.txt`.[b]

[a] See `www.robotstxt.org/orig.html` and `en.wikipedia.org/wiki/Robots_exclusion_standard`.

[b] Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. For more information, see `https://docs.python.org/3/library/urllib.robotparser.html`.

The standard way to get the HTML source code of a website using Python is via the `requests` library.[1] Calling `requests.get()` sends an HTTP GET request to a specified website; the result is an object with a response code to indicate whether or not the request was responded to, and access to the website source code.

[1] Though `requests` is not part of the standard library, it is recognized as a standard tool in the data science community. See `http://docs.python-requests.org/`.

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.example.com")
>>> print(response.status_code, response.ok, response.reason)
200 True OK

# The HTML of the website is stored in the 'text' attribute.
>>> print(response.text)
<!doctype html>
<html>
<head>
    <title>Example Domain</title>

    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
# ...
```

ACHTUNG!

Consecutive GET requests without pauses can strain a website's server and provoke retaliation. Most servers are designed to identify such scrapers, block their access, and sometimes even blacklist the user. This is especially common in smaller websites that aren't built to handle enormous amounts of traffic. To briefly pause the program between requests, use `time.sleep()`.

```
>>> import time
>>> time.sleep(3)                    # Pause execution for 3 seconds.
```

The amount of necessary wait time depends on the website. Sometimes, `robots.txt` contains a `Request-rate` directive which gives a ratio of the form `<requests>/<seconds>`. If this doesn't exist, pausing for a half-second to a second between requests is typically sufficient. An email to the site's webmaster is always the safest approach, and may be necessary for large scraping operations.

Crawling Through Multiple Pages

While web *scraping* refers to the actual gathering of web-based data, web *crawling* refers to the navigation of a program between webpages. Web crawling allows a program to gather related data from multiple web pages and websites.

Consider `www.wunderground.com`, a site that records weather data in various cities. The page `https://www.wunderground.com/history/airport/KSAN/2012/7/1/DailyHistory.html` records the weather in San Diego on July 1, 2012. Data for previous or subsequent days can be accessed by clicking on the `Previous Day` and `Next Day` links, so gathering data for an entire month or year requires crawling through several pages. The following example gathers temperature data for July 1 through July 4 of 2012.

```
import re
import requests
from bs4 import BeautifulSoup


def wunder_temp(day="/history/airport/KSAN/2012/7/1/DailyHistory.html"):
    """Crawl through Weather Underground and extract temperature data."""

    # Initialize variables, including a regex for finding the 'Next Day' link.
    actual_mean_temp = []
    next_day_finder = re.compile(r"Next Day")
    base_url = "https://www.wunderground.com"       # Web page base URL.
    page = base_url + day                           # Complete page URL.
    current = None

    for _ in range(4):
        while current is None:  # Try downloading until it works.
            # Download the page source and PAUSE before continuing.
            page_source = requests.get(page).text
            time.sleep(1)           # PAUSE before continuing.
            soup = BeautifulSoup(page_source, "html.parser")
            current = soup.find(string="Mean Temperature")

        # Navigate to the relevant tag, then extract the temperature data.
        temp_tag = current.parent.parent.next_sibling.next_sibling.span.span
        actual_mean_temp.append(int(temp_tag.string))

        # Find the URL for the page with the next day's data.
        new_day = soup.find(string=next_day_finder).parent["href"]
        page = base_url + new_day                   # New complete page URL.
        current = None

    return actual_mean_temp
```

In this example, the `for` loop cycles through the days, and the `while` loop ensures that each website page loads properly: if the downloaded `page_source` doesn't have a tag whose string is "Mean Temperature", the request is sent again. Later, after locating and recording the Actual Mean Temperature, the function locates the link to the next day's page. This link is stored in the HTML as a relative website path (`/history/airport/...`); the complete URL to the next day's page is the concatenation of the base URL `https://www.wunderground.com` with this relative link.

Problem 1. Modify `wunder_temp()` so that it gathers the Actual Mean Temperature, Actual Max Temperature, and Actual Min Temperature for every day in July of 2012. Plot these three measurements against time on the same plot.

Consider printing information at each iteration of the outer loop to keep track of the program's progress.

An alternative approach that is often useful is to first identify the links to relevant pages, then scrape each of these page in succession. For example, the Federal Reserve releases quarterly data on large banks in the United States at `http://www.federalreserve.gov/releases/lbr`. The following function extracts the four measurements of total consolidated assets for JPMorgan Chase during 2004.

```
def bank_data():
    """Crawl through the Federal Reserve site and extract bank data."""
    # Compile regular expressions for finding certain tags.
    link_finder = re.compile(r"2004$")
    chase_bank_finder = re.compile(r"^JPMORGAN CHASE BK")

    # Get the base page and find the URLs to all other relevant pages.
    base_url="https://www.federalreserve.gov/releases/lbr/"
    base_page_source = requests.get(base_url).text
    base_soup = BeautifulSoup(base_page_source, "html.parser")
    link_tags = base_soup.find_all(name='a', href=True, string=link_finder)
    pages = [base_url + tag.attrs["href"] for tag in link_tags]

    # Crawl through the individual pages and record the data.
    chase_assets = []
    for page in pages:
        time.sleep(1)               # PAUSE, then request the page.
        soup = BeautifulSoup(requests.get(page).text, "html.parser")

        # Find the tag corresponding to Chase Banks's consolidated assets.
        temp_tag = soup.find(name="td", string=chase_bank_finder)
        for _ in range(10):
            temp_tag = temp_tag.next_sibling
        # Extract the data, removing commas.
        chase_assets.append(int(temp_tag.string.replace(',', '')))

    return chase_assets
```

Problem 2. Modify `bank_data()` so that it extracts the total consolidated assets ("Consol Assets") for JPMorgan Chase, Bank of America, and Wells Fargo recorded each December from 2004 to the present. In a single figure, plot each bank's assets against time. Be careful to keep the data sorted by date.

Problem 3. ESPN hosts data on NBA athletes at `http://www.espn.go.com/nba/statistics`. Each player has their own page with detailed performance statistics. For each of the five offensive leaders in points and each of the five defensive leaders in rebounds, extract the player's career minutes per game (MPG) and career points per game (PPG). Make a scatter plot of MPG against PPG for these ten players.

Asynchronously Loaded Content and User Interaction

Web crawling with the methods presented in the previous section fails under a few circumstances. First, many webpages use *JavaScript*, the standard client-side scripting language for the web, to load portions of their content *asynchronously*. This means that at least some of the content isn't initially accessible through the page's source code. Second, some pages require user interaction, such as clicking buttons which aren't links (`<a>` tags which contain a URL that can be loaded) or entering text into form fields (like search bars).

The *Selenium* framework provides a solution to both of these problems. Originally developed for writing unit tests for web applications, Selenium allows a program to open a web browser and interact with it in the same way that a human user would, including clicking and typing. It also has BeautifulSoup-esque tools for searching the HTML source of the current page.

> **Note**
>
> Selenium requires an executable *driver* file for each kind of browser. The following examples use Google Chrome, but Selenium supports Firefox, Internet Explorer, Safari, Opera, and PhantomJS (a special browser without a user interface). See `https://seleniumhq.github.io/selenium/docs/api/py` or `http://selenium-python.readthedocs.io/installation.html` for installation instructions.

To use Selenium, start up a browser using one of the drivers in `selenium.webdriver`. The browser has a `get()` method for going to different web pages, a `page_source` attribute containing the HTML source of the current page, and a `close()` method to exit the browser.

```
>>> from selenium import webdriver

# Start up a browser and go to example.com.
>>> browser = webdriver.Chrome()
>>> browser.get("https://www.example.com")

# Feed the HTML source code for the page into BeautifulSoup for processing.
>>> soup = BeautifulSoup(browser.page_source, "html.parser")
>>> print(soup.prettify())
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>
   Example Domain
  </title>
  <meta charset="utf-8"/>
  <meta content="text/html; charset=utf-8" http-equiv="Content-type"/>
# ...

>>> browser.close()                # Close the browser.
```

Selenium can deliver the HTML page source to BeautifulSoup, but it also has its own tools for finding tags in the HTML.

Method	Returns
`find_element_by_tag_name()`	The first tag with the given name
`find_element_by_name()`	The first tag with the specified `name` attribute
`find_element_by_class_name()`	The first tag with the given `class` attribute
`find_element_by_id()`	The first tag with the given `id` attribute
`find_element_by_link_text()`	The first tag with a matching `href` attribute
`find_element_by_partial_link_text()`	The first tag with a partially matching `href` attribute

Table 6.1: Methods of the `selenium.webdriver.Chrome` class.

Each of the `find_element_by_*()` methods returns a single object representing a *web element* (of type `selenium.webdriver.remote.webelement.WebElement`), much like a BeautifulSoup tag (of type `bs4.element.Tag`). If no such element can be found, a Selenium `NoSuchElementException` is raised. Each webdriver also has several `find_elements_by_*()` methods (elements, plural) that return a list of all matching elements, or an empty list if there are no matches.

Web element objects have methods that allow the program to interact with them: `click()` sends a click, `send_keys()` enters in text, and `clear()` deletes existing text. This functionality makes it possible for Selenium to interact with a website in the same way that a human would. For example, the following code opens up `https://www.google.com`, types "Python Selenium Docs" into the search bar, and hits enter.

```
>>> from selenium.webdriver.common.keys import Keys
>>> from selenium.common.exceptions import NoSuchElementException

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://www.google.com")
...     try:
...         # Get the search bar, type in some text, and press Enter.
...         search_bar = browser.find_element_by_name('q')
...         search_bar.clear()                  # Clear any pre-set text.
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN)   # Press Enter.
...     except NoSuchElementException:
...         print("Could not find the search bar!")
...         raise
... finally:
...     browser.close()
...
```

Problem 4. The arXiv (pronounced "archive") is an online repository of scientific publications, hosted by Cornell University. Write a function that accepts a string to serve as a search query. Use Selenium to enter the query into the search bar of `https://arxiv.org` and press Enter. The resulting page has up to 25 links to the PDFs of technical papers that match the query. Gather these URLs, then continue to the next page (if there are more results) and continue gathering links until obtaining at most 100 URLs. Return the list of URLs.

NOTE

Using Selenium to access a page's source code is typically much safer, though slower, than using `requests.get()`, since Selenium waits for each web page to load before proceeding. For instance, the arXiv is a somewhat defensive about scrapers (`https://arxiv.org/help/robots`), but Selenium makes it possible to gather info from the website without offending the administrators.

Problem 5. *Project Euler* (`https://projecteuler.net`) is a collection of mathematical computing problems. Each problem is listed with an ID, a description/title, and the number of users that have solved the problem.

Using Selenium, BeautifulSoup, or both, for each of the (at least) 600 problems in the archive at `https://projecteuler.net/archives`, record the problem ID and the number of people who have solved it. Return a list of IDs, sorted from largest to smallest by the number of people who have solved them. That is, the first entry in the list should be the ID of the **most solved** problem, and the last entry in the list should be the ID of the **least solved** problem. (Hint: start by identifying the URLs to each archive page.)