8 Web Crawling

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 on 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.

1. The scraper doesn't respect the website's terms and conditions or gathers private or proprietary data.
2. 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. 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*.[1]

> ACHTUNG!
>
> 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`. Make sure to scrape websites legally.

Recall that consecutive 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()`.

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

```
>>> 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 `Crawl-delay` directive which gives a number of seconds to wait between successive requests. 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.

Python provides a parsing library called `urllib.robotparser` for reading `robot.txt` files. Below is an example of using this library to check where robots are allowed on arxiv.org. A website's `robots.txt` file will often include different instructions for specific crawlers. These crawlers are identified by a `User-agent` string. For example, Google's webcrawler, `User-agent` Googlebot, may be directed to index only the pages the website wants to have listed on a Google search. We will use the default `User-agent`, `"*"`.

```
>>> from urllib import robotparser
>>> rp = robotparser.RobotFileParser()
# Set the URL for the robots.txt file. Note that the URL contains `robots.txt'
>>> rp.set_url("https://arxiv.org/robots.txt")
>>> rp.read()
# Request the crawl-delay time for the default User-agent
>>> rp.crawl_delay("*")
15
# Check if User-agent "*" can access the page
>>> rp.can_fetch("*", "https://arxiv.org/archive/math/")
True
>>> rp.can_fetch("*", "https://arxiv.org/IgnoreMe/")
False
```

Problem 1. Write a program that accepts a web address defaulting to the site `http://example.webscraping.com` and a list of pages defaulting to `["/", "/trap", "/places/default/search"]`. For each page, check if the `robots.txt` file permits access. Return a list of boolean values corresponding to each page. Also return the crawl delay time.

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 `books.toscrape.com`, a site to practice web scraping that mimics a bookstore. The page `http://books.toscrape.com/catalogue/category/books/mystery_3/index.html` lists mystery books with overall ratings and review. More mystery books can be accessed by clicking on the `next` link. The following example demonstrates how to navigate between webpages to collect all of the mystery book titles.

```
def scrape_books(start_page = "index.html"):
    """ Crawl through http://books.toscrape.com and extract mystery titles"""

    # Initialize variables, including a regex for finding the 'next' link.
    base_url="http://books.toscrape.com/catalogue/category/books/mystery_3/"
    titles = []
    page = base_url + start_page                # Complete page URL.
    next_page_finder = re.compile(r"next")      # We need this button.

    current = None

    for _ in range(4):
        while current == 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_all(class_="product_pod")

        # Navigate to the correct tag and extract title
        for book in current:
            titles.append(book.h3.a["title"])

        # Find the URL for the page with the next data.
        if "page-4" not in page:
            new_page = soup.find(string=next_page_finder).parent["href"]
            page = base_url + new_page      # New complete page URL.
            current = None
    return titles
```

In this example, the `for` loop cycles through the pages of books, and the `while` loop ensures that each website page loads properly: if the downloaded `page_source` doesn't have a tag whose class is `product_pod`, the request is sent again. After recording all of the titles, the function locates the link to the next page. This link is stored in the HTML as a relative website path (`page-2.html`); the complete URL to the next day's page is the concatenation of the base URL `http://books.toscrape.com/catalogue/category/books/mystery_3/` with this relative link.

Problem 2. Modify `scrape_books()` so that it gathers the price for each fiction book and returns the mean price, in £, of a fiction book.

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 (for example, if you have to scroll down to load more results). 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 and driver download instructions.
>
> If your program still can't find the driver after you've downloaded it, add the argument `executable_path = "path/to/driver/file"` when you call `webdriver`. If this doesn't work, you may need to add the location to your system PATH. On a Mac, open the file `/etc/path` and add the new location. On Linux, add `export PATH="path/to/driver/file:$PATH"` to the file `/.bashrc` . For Windows, follow a tutorial such as this one: `https://www.architectryan.com/2018/03/17/add-to-the-path-on-windows-10/`.

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 8.1: 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. If you want to find more than just the first matching object, 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 3. The website IMDB contains a variety of information on movies. Specifically, information on the top 10 box office movies of the week can be found at `https://www.imdb.com/chart/boxoffice`. Using BeaufiulSoup, Selenium, or both, return a list, with each title on a new row, of the top 10 movies of the week and order the list according to the total grossing of the movies, from most money to the least. Break ties using the weekend gross, from most money to the least.

Using CSS Selectors

In addition to the methods listed in Table 8.1, you can also use CSS or XPath selectors to interact more precisely with the page source. Refer to Table 3 from the WebScraping lab for a review of CSS syntax. The following code searches Google for "Python Selenium Docs" and then clicks on the second result.

```
#As before, go to Google and type in the search bar,
# but this time we use CSS selectors

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

>>> browser = webdriver.Chrome()
>>> try:
...     browser.get("https://google.com")
...     try:
...         search_bar = browser.find_element_by_css_selector(
                        "input[name='q']")
...         search_bar.clear()
...         search_bar.send_keys("Python Selenium Docs")
...         search_bar.send_keys(Keys.RETURN)
...         try:
...             # Wait a second, then get the second search result
...             time.sleep(1)
...             # "+ div" returns the element's  next sibling with a "div" tag
...             second_result = browser.find_element_by_css_selector(
                                "div[class='g'] + div")
...             try:
...                 # Get the link, which is a child of second_result
...                 link = second_result.find_element_by_css_selector(
                            "div[class='r']")
...                 link.click()
...                 time.sleep(1)

...             #Remember to handle exceptions
...             except NoSuchElementException:
...                 print("Could not find link")
...         except NoSuchElementException:
...             print("Could not find second result")
```

```
...     except NoSuchElementException:
...         print("Could not find the search bar")
... finally:
...     browser.close()
```

In the above example, we could have used `find_element_by_class_name()`, but when you need more precision than that, CSS selectors can be very useful. Remember that to view specific HTML associated with an object in Chrome or Firefox, you can right click on the object and click "Inspect." For Safari, you need to first enable "Show Develop menu" in "Preferences" under "Advanced." Keep in mind that you can also search through the source code (ctrl+f or cmd+f) to make sure you're using a unique identifier.

> **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, some websites are somewhat defensive about scrapers, but Selenium can sometimes make it possible to gather info without offending the administrators.

Problem 4. *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, record the number of people who have solved each of the 700+ problems in the archive at https://projecteuler.net/archives. Plot the number of people who have solved each problem against the problem IDs, using a log scale for the y-axis. Display the scatter plot, then state the IDs of which problems have been solved most and least number of times.

Problem 5. The website `http://example.webscraping.com` contains a list of countries of the world. Using Selenium, go to the search page, enter the letters `"ca"`, and hit `enter`. Remember to use the crawl delay time you found in Problem 1 so you don't send your requests too fast. Gather the `href` links associated with the `<a>` tags of all 10 displayed results. Print each link on a different line.