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.*

NOTE

For grading purposes, before returning a value in each function of this lab, write the value being returned to a `pickle` file. Name the file `ans1` if it contains the returned value of Problem 1, `ans2` for Problem 2, and so on, so that you have five pickle files saved. In order to create a pickle file, use the following code:

```
>>> import pickle
>>>
>>> # Write the answer of Problem 1, "value", to a pickle file
>>> with open("ans1", "wb") as fp:
>>>     pickle.load(value, fp)
```

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. [a]

[a]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.[2] Calling `requests.get()` sends an HTTP GET request to a specified website; the result is an object with a response code that indicates whether or not the request was successful and if so, access to the website source code.

```
>>> 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" />
# ...
```

Recall that 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.
```

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

[2]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/`.

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 `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.

```python
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(2):
        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-2" 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 1. Modify `scrape_books()` so that it gathers the price for each fiction book and returns the mean price, in £, of a fiction book.

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.

```python
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. Return a list of lists where each list contains the assets of each bank.

Problem 3. The Basketball Reference website at `https://www.basketball-reference.com` contains data on NBA athletes, including which player led different categories for each season. For the past ten seasons, identify which player had the most season points and find how many points they scored during that season. Return a list of triples consisting of the season, the player, and the points scored, ("season year", "player name", points scored).

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 and driver download 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 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. 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 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 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.

Problem 5. 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 defaulting to `linkedin`. Use Selenium to enter the query into the search bar of `https://arxiv.org` and press Enter. The resulting page has up to 50 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 150 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.