DTF1 Advanced Data Analysis

Web Scraping

Martina Fraschini

University of Luxembourg

Web Scraping

Objective:

- Web Scraping is the process of gathering data from websites on the internet.
- Since almost everything rendered by an internet browser as a web page uses HTML, the first step in web scraping is being able to extract information from HTML.
- In this lab, we introduce the requests library for scraping web pages, and BeautifulSoup, Python's canonical tool for efficiently and cleanly navigating and parsing HTML.
- We also see two methods for crawling through multiple web pages without violating copyright laws or straining the load on a server.

Data Serialization

- *Serialization* is the process of packaging data in a form that makes it easy to transmit the data and quickly reconstruct it on another computer or in a different programming language.
- Many serialization metalanguages exist, such as Python's `pickle`, YAML, XML, and JSON.
- *Deserialization* is the process of reconstructing an object from the string.

JSON

- JSON, which stands for *JavaScript Object Notation*, is the dominant format for serialization in web applications. Despite having "JavaScript" in its name, JSON is a language-independent format and is frequently used for transmitting data between different programming languages.
- JSON stores information about objects as a specially formatted string that is easy for both humans and machines to read and write. It is built on two types of data structures: a collection of key/value pairs similar to Python's built-in `dict`, and an ordered list of values similar to Python's built-in `list`.

```
{                                   # A family's info written in JSON format.
    "lastname": "Smith",            # The outer dictionary has two keys:
    "children": [                   # "lastname" and "children".
        {                           # The "children" key maps to a list of
            "name": "Timmy",        # two dictionaries, one for each of the
            "age": 8                # two children.
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

Python Module for JSON

- The Python standard library module for JSON is called `json`.
- A string written in JSON format that represents a piece of data is called a *JSON message*.
- The `json.dumps()` function generates the JSON message for a single Python object.
- To load a JSON string or file, use the json decoder `json.loads()` or `json.load()`, respectively.

```
>>> import json
# Store info about a car in a nested dictionary.
>>> my_car = {
...     "car": {
...         "make": "Ford",
...         "color": [255, 30, 30]  },
...     "owner": "me" }

# Get the JSON message corresponding to my_car.
>>> car_str = json.dumps(my_car)
>>> car_str
'{"car": {"make": "Ford", "color": [255, 30, 30]}, "owner": "me"}'

# Load the JSON message into a Python object, reconstructing my_car.
>>> car_object = json.loads(car_str)
>>> for key in car_object:          # The loaded object is a dictionary.
...     print(key + ':', car_object[key])
...
car: {'make': 'Ford', 'color': [255, 30, 30]}
owner: me
```

APIs

- An *Application Program Interface* (API) is a particular kind of server that listens for requests from authorized users and responds with data.
- Every API has *endpoints* where clients send their requests.
- Though standards exist for creating and communicating with APIs, most APIs have a unique syntax for authentication and requests that is documented by the organization providing the service.
- The `requests` module is the standard way to send a download request to an API in Python.

```
>>> import requests
>>> requests.get(endpoint).json()        # Download and extract the data.
```

- **Careful!** Each website and API has a policy that specifies appropriate behavior for automated data retrieval and usage. If data is requested without complying with these requirements, there can be severe legal consequences. Most websites detail their policies in a file called *robots.txt* on their main page.

HTTP and Requests 1/2

- HTTP stands for *Hypertext Transfer Protocol*, which is an application layer networking protocol.
- The HTTP protocol is centered around a request and response paradigm, in which a client makes a request to a server and the server replies with a response.
- There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. GET requests request information from the server, POST requests modify the state of the server, and PUT requests add new pieces of data to the server.
- The standard way to get the source code of a website using Python is via the `requests` library.
- Calling `requests.get()` sends an HTTP GET request to a specified website. The website returns a response code, which indicates whether or not the request was received, understood, and accepted. If the response code is good, typically 200, then the response will also include the website source code as an HTML file.

HTTP and Requests 2/2

```
>>> import requests

# Make a request and check the result. A status code of 200 is good.
>>> response = requests.get("http://www.byu.edu")
>>> 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 lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/↩
    content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/  ↩
    og: http://ogp.me/ns# rdfs: http://www.w3.org/2000/01/rdf-schema# schema:↩
    http://schema.org/ sioc: http://rdfs.org/sioc/ns# sioct: http://rdfs.org↩
    /sioc/types# skos: http://www.w3.org/2004/02/skos/core# xsd: http://www.↩
    w3.org/2001/XMLSchema# " class=" ">

  <head>
    <meta charset="utf-8" />
# ...
```

- Some websites aren't built to handle large amounts of traffic or many repeated requests. Most are built to identify web scrapers or crawlers that initiate many consecutive GET requests without pauses, and retaliate or block them.
- Make sure to store the data that you receive in a file and include error checks to prevent retrieving the same data unnecessarily.

HTML

- *Hyper Text Markup Language*, or HTML, is the standard markup language–a language designed for the processing, definition, and presentation of text–for creating webpages.
- It structures a document using pairs of *tags* that surround and define content.
- Opening tags have a tag name surrounded by angle brackets (`<tag-name>`), while the closing tag looks the same with a forward slash before the tag name (`</tag-name>`).
- Most tags can be combined with *attributes* to include more data about the content, help identify individual tags, and make navigating the document much simpler.

```
<html>                                     <!-- Opening tags -->
    <body>
        <p>
            Click <a id='info' href='http://www.example.com'>here</a>
            for more information.
        </p>                               <!-- Closing tags -->
    </body>
</html>
```

- A list of all current HTML tags can be found here.

BeautifulSoup 1/3

- BeautifulSoup (`bs4`) is a package that makes it simple to navigate and extract data from HTML documents.
- It is not part of the standard library; install it with `pip install beautifulsoup4`.
- The `bs4.BeautifulSoup` class accepts two parameters to its constructor: a string of HTML code and an HTML parser to use under the hood. The standard choice for the parser is `"html.parser"`.
- A `BeautifulSoup` object represents an HTML document as a tree. In the tree, each tag is a *node* with nested tags and strings as its *children*.
- The `prettify()` method returns a string that can be printed to represent the BeautifulSoup object in a readable format that reflects the tree structure.
- Each tag in a `BeautifulSoup` object's HTML code is stored as a `bs4.element`. Tags are accessible directly through the `BeautifulSoup` object.

BeautifulSoup 2/3

```
>>> from bs4 import BeautifulSoup

>>> small_example_html = """
<html><body><p>
    Click <a id='info' href='http://www.example.com'>here</a>
    for more information.
</p></body></html>
"""
>>> small_soup = BeautifulSoup(small_example_html, 'html.parser')
>>> print(small_soup.prettify())
<html>
 <body>
  <p>
   Click
   <a href="http://www.example.com" id="info">
    here
   </a>
   for more information.
  </p>
 </body>
</html>

# Get the <p> tag (and everything inside of it).
>>> small_soup.p
<p>
    Click <a href="http://www.example.com" id="info">here</a>
    for more information.
</p>

# Get the <a> sub-tag of the <p> tag.
>>> a_tag = small_soup.p.a
>>> print(a_tag, type(a_tag), sep='\n')
<a href="http://www.example.com" id="info">here</a>
<class 'bs4.element.Tag'>
```

BeautifulSoup 3/3

```
# Get just the name, attributes, and text of the <a> tag.
>>> print(a_tag.name, a_tag.attrs, a_tag.string, sep="\n")
a
{'id': 'info', 'href': 'http://www.example.com'}
here
```

Attribute	Description
`name`	The name of the tag.
`attrs`	A dictionary of the attributes.
`string`	The single string contained in the tag.
`strings`	Generator for strings of children tags.
`stripped_strings`	Generator for strings of children tags, stripping whitespace.
`text`	Concatenation of strings from all children tags.

Navigating the Tree Structure 1/3

- Every HTML tag (except for the topmost tag, which is usually `<html>`) has a *parent* tag.
- Each tag also has zero or more *sibling* and *children* tags or text.
- Following a true tree structure, every `bs4.element.Tag` in a soup has multiple attributes for accessing or iterating through parent, sibling, or child tags.

Attribute	Description
`parent`	The parent tag
`parents`	Generator for the parent tags up to the top level
`next_sibling`	The tag immediately after to the current tag
`next_siblings`	Generator for sibling tags after the current tag
`previous_sibling`	The tag immediately before the current tag
`previous_siblings`	Generator for sibling tags before the current tag
`contents`	A list of the immediate children tags
`children`	Generator for immediate children tags
`descendants`	Generator for all children tags (recursively)

Navigating the Tree Structure 2/3

```
# Start at the first <a> tag in the soup.
>>> a_tag = pig_soup.a
>>> a_tag
<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>

# Get the names of all of <a>'s parent tags, traveling up to the top.
# The name '[document]' means it is the top of the HTML code.
>>> [par.name for par in a_tag.parents]         # <a>'s parent is <p>, whose
['p', 'body', 'html', '[document]']             # parent is <body>, and so on.

# Get the next siblings of <a>.
>>> a_tag.next_sibling
'\n'                                            # The first sibling is just text.
>>> a_tag.next_sibling.next_sibling             # The second sibling is a tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>


# Alternatively, get all siblings past <a> at once.
>>> list(a_tag.next_siblings)
['\n',
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 ', and\n',
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>,
 '\n',
 <p>The three pigs had an odd fascination with experimental construction.</p>,
 '\n',
 <p>...</p>,
 '\n']
```

Navigating the Tree Structure 3/3

- Newline characters are considered to be children of a parent tag.
- Iterating through children or siblings often requires checking which entries are tags and which are just text.

```
# Get to the <p> tag that has class="story".
>>> p_tag = pig_soup.body.p.next_sibling.next_sibling
>>> p_tag.attrs["class"]                # Make sure it's the right tag.
['story']

# Iterate through the child tags of <p> and print hrefs whenever they exist.
>>> for child in p_tag.children:
...     if hasattr(child, "attrs") and "href" in child.attrs:
...         print(child.attrs["href"])
http://example.com/larry
http://example.com/mo
http://example.com/curly
```

Searching for Tags

- The `find()` and `find_all()` methods of the BeautifulSoup class identify tags that have distinctive characteristics, making it much easier to jump straight to a desired location in the HTML code.
- The `find()` method only returns the first tag that matches a given criteria, while `find_all()` returns a list of all matching tags.
- Tags can be matched by name, attributes, and/or text.

```
# Find the first <b> tag in the soup.
>>> pig_soup.find(name='b')
<b>The Three Little Pigs</b>

# Find all tags with a class attribute of 'pig'.
# Since 'class' is a Python keyword, use 'class_' as the argument.
>>> pig_soup.find_all(class_="pig")
[<a class="pig" href="http://example.com/larry" id="link1">Larry,</a>,
 <a class="pig" href="http://example.com/mo" id="link2">Mo</a>,
 <a class="pig" href="http://example.com/curly" id="link3">Curly.</a>]

# Find the first tag that matches several attributes.
>>> pig_soup.find(attrs={"class": "pig", "href": "http://example.com/mo"})
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>

# Find the first tag whose text is 'Mo'.
>>> pig_soup.find(string='Mo')
'Mo'                                    # The result is the actual string,
>>> pig_soup.find(string='Mo').parent   # so go up one level to get the tag.
<a class="pig" href="http://example.com/mo" id="link2">Mo</a>
```

Scraping Etiquette

- 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* that specifies which parts of the website are off-limits and how often requests can be made according to the *robots exclusion standard*.
- 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*.**

Pausing

- 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()`. 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 1/2

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

```
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
```

Crawling Through Multiple Pages 2/2

- An alternative approach that is often useful is to first identify the links to relevant pages, then scrape each of these page in succession.

```
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
```

Homework

Problem 1

The file *nyc_traffic.json* contains information about 1000 traffic accidents in New York City during the summer of 2017. Each entry lists one or more reasons for the accident, such as "Unsafe Speed" or "Fell Asleep."
Write a function that loads the data from the JSON file. Make a readable, sorted bar chart showing the total number of times that each of the 7 most common reasons for accidents are listed in the data set. To check your work, the 6th most common reason is "Backing Unsafely," listed 59 times.
(Hint: the `collections.Counter` data structure and `plt.tight_layout()` may be useful here.)

Problem 2

Use the `requests` library to get the HTML source for the website `http://www.example.com`. Save the source as a file called *example.html*. If the file already exists, make sure not to scrape the website, or overwrite the file.

Problem 3

Using the output from Problem 2, examine the HTML source code for `http://www.example.com`. What tags are used? What is the value of the type attribute associated with the style tag?
Write a function that returns the set of names of tags used in the website, and the value of the type attribute of the style tag (as a string).
(Hint: there are ten unique tag names.)

Problem 4

The `BeautifulSoup` class has a `find_all()` method that, when called with `True` as the only argument, returns a list of all tags in the HTML source code.
Write a function that accepts a string of HTML code as an argument. Use BeautifulSoup to return a list of the names of the tags in the code. Use your function and the source code from `http://www.example.com` (or the output from Problem 2) to check your answers from Problem 3.

Problem 5

Using the output from Problem 2, write a function that reads the file and loads the code into BeautifulSoup. Find the only <a> tag with a hyperlink, and return its text.

Problem 6

Problem 6. The file *large_banks_index.html* is an index of data about large banks, as recorded by the Federal Reserve. Write a function that reads the file and loads the source into BeautifulSoup. Return a list of the tags containing the links to bank data from September 30, 2003 to December 31, 2014, where the dates are in reverse chronological order.

Problem 7

The file *large_banks_data.html* is one of the pages from the index in Problem 6. Write a function that reads the file and loads the source into BeautifulSoup. Create a single figure with two subplots:

1. A sorted bar chart of the seven banks with the most domestic branches.
2. A sorted bar chart of the seven banks with the most foreign branches.

In the case of a tie, sort the banks alphabetically by name.

Problem 8 (optional)

The Basketball Reference website `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).