Quera Data Analysis Bootcamp:

WEB SCRAPING WITH PYTHON

A presentation by *Parsa Abbasi*

January, 2023

TABLE OF CONTENTS

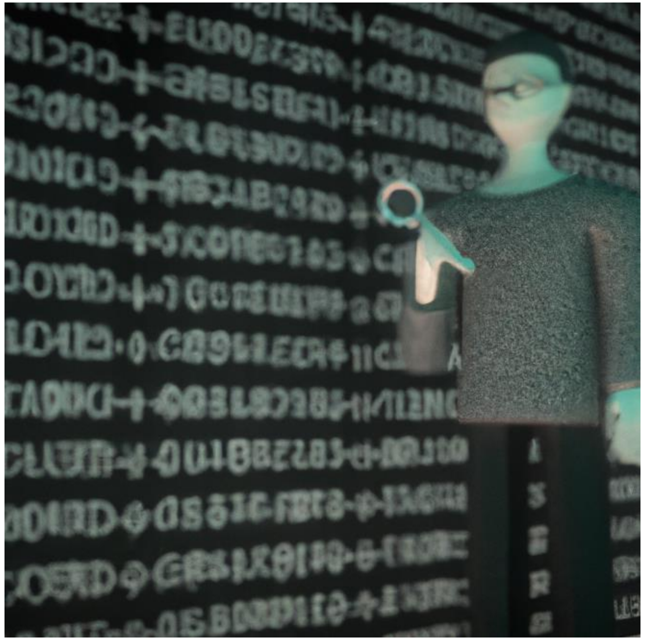
01

Introduction

What is Web Scraping?

*"**Web scraping**, **web harvesting**, or **web data extraction** is data scraping used for extracting data from websites.“* *(Thapelo, et al., 2016)*

“The construction of an agent to download, parse, and organize data from the web in an automated manner.” *(Broucke & Baesens, 2018)*


Made by DALL-E

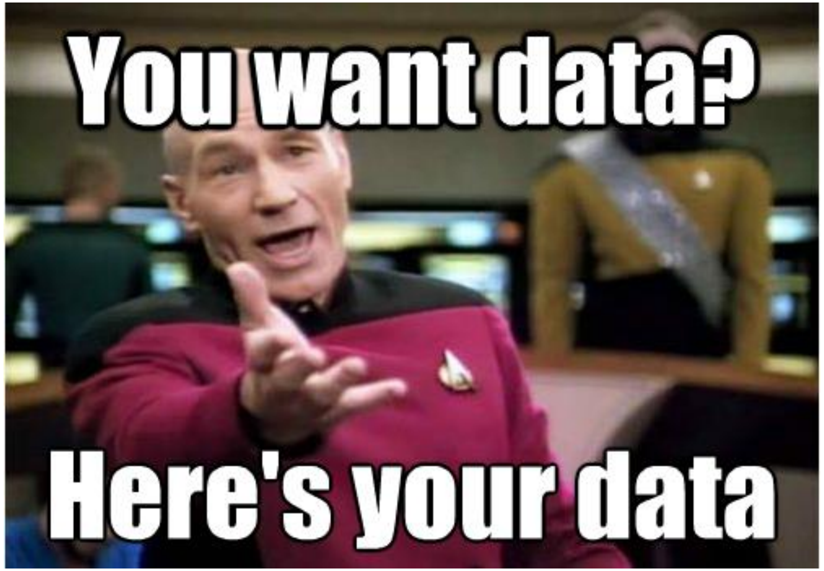
Why Web Scraping for Data Science?

- Gathering data for a research project
- Monitoring industry trends
- Analyzing consumer sentiment
- Creating a dataset for machine learning

Made by DALL-E

Why Web Scraping for Data Science?

- Social network analytics
- Retrieve information from Wikipedia
- Enrich the dataset
- Keeping track of competitors



What about APIs?

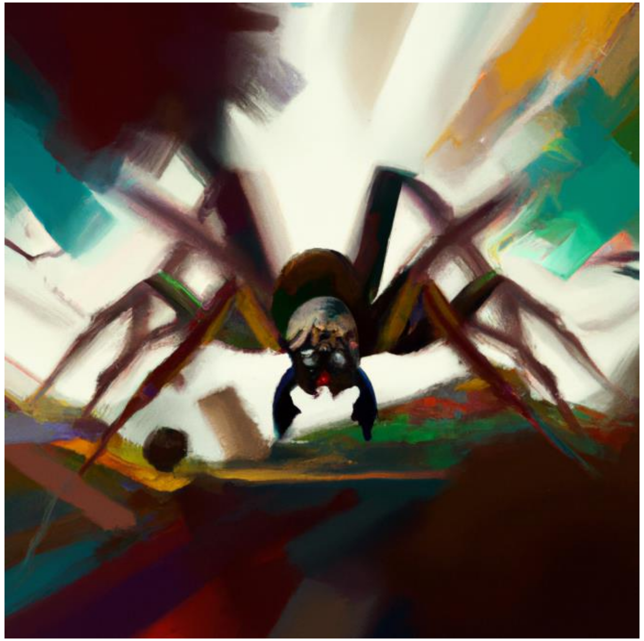
- The general rule of thumb:
  - Look for an API first and use that if you can.

- Some reasons why web scraping might be preferable over the use of an API:
  - The website does not provide an API.
  - The API provided is not free, or is too expensive.
  - API access rate is limited.
  - The API does not expose all the data you wish to obtain.
  - API documentation is poor.

What is Web Crawling?

Web crawling is the process of automatically visiting web pages and following links to other pages within the same website or to other websites.

"In general terms, the term ***crawler*** *indicates a program's ability to navigate web pages on its own, perhaps even without a well-defined end goal or purpose, endlessly exploring what a site or the web has to offer.."* *(Broucke & Baesens, 2018)*


Made by DALL-E

02

The Web

Connecting

“Once you start web scraping, you start to appreciate all the little things that browsers do for you.” *(Mitchell, 2018)*

To give you an idea of the infrastructure required to get information to your browser, let’s use the following example: *(Mitchell, 2018)*

- Alice owns a web server.
- Bob uses a desktop computer, which is trying to connect to Alice’s server.

When one machine wants to talk to another machine, something like the following exchange takes place:

Connecting: Example

01

- Bob’s computer sends along a stream of 1 and 0 bits, indicated by high and low voltages on a wire.

- These bits form some information, containing a **header** and **body**.

  - The *header* contains an immediate destination of his local router’s MAC address, with a final destination of Alice’s IP address.
  - The *body* contains his request for Alice’s server application.

Connecting: Example

02

- Bob’s local router receives all these 1s and 0s and interprets them as a packet, from Bob’s own MAC address, destined for Alice’s IP address.
- His router stamps its own IP address on the packet as the “from” IP address, and sends it off across the internet.

Connecting: Example

03

- Bob’s packet traverses several intermediary servers, which direct his packet toward the correct physical/wired path, on to Alice’s server.

Connecting: Example

04

- Alice’s server receives the packet at her IP address.

Connecting: Example

05

- Alice’s server reads the packet port destination in the header, and passes it off to the appropriate application—the web server application.

  (The packet port destination is almost always port **80** for web applications; this can be thought of as an apartment number for packet data, whereas the IP address is like the street address.)

Connecting: Example

06

- The web server application receives a stream of data from the server processor.

- This data says something like the following:

    - This is a GET request.
    - The following file is requested: *index.html*.

Connecting: Example

07

- The web server locates the correct HTML file
- Bundles it up into a new packet to send to Bob
- And sends it through to its local router, for transport back to Bob’s machine, through the same process.

HTTP

As we recall, a client (the web browser, in most cases) and web server will communicate by sending plain text messages. The client sends requests to the server and the server sends responses, or replies.

The core component in the exchange of messages consists of a HyperText Transfer Protocol (HTTP) request message to a web server, followed by an HTTP response (also oftentimes called an HTTP reply). *(Broucke & Baesens, 2018)*

HTTP: Example

```
GET / HTTP/1.1
Host: example.com
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 ↵
(KHTML, like Gecko) Chrome/60.0.3112.90 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: https://www.google.com/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,nl;q=0.6
<CR><LF>
```

HTTP: Example

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Encoding: gzip
Content-Type: text/html;charset=utf-8
Date: Mon, 28 Aug 2017 10:57:42 GMT
Server: Apache v1.3
Vary: Accept-Encoding
Transfer-Encoding: chunked
<CR><LF>
<html>
<body>Welcome to My Web Page</body>
</html>
```

The *requests* library

The Python `requests` library is a powerful and easy-to-use library for making HTTP requests in Python. With `requests`, you can send HTTP requests using various methods, such as GET, POST, PUT, DELETE, and more. It also supports various authentication methods, cookies, and other advanced features.

```
pip install requests
```

```python
import requests

response = requests.get('https://quera.org')

print(response.status_code) # prints the status code
print(response.headers)     # prints the response headers
print(response.text)        # prints the response body
```

HyperText Markup Language (HTML)

- It is the standard markup language for creating web pages
- HTML consists of a series of elements that are used to structure and format content on the web
- HTML is often used to structure and format the content that is included in an HTTP response.

Made by DALL-E

HyperText Markup Language (HTML)

An HTML document consists of a series of elements, each contained within an opening and closing tag.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>My Page</title>
    </head>
    <body>
        <h1>Welcome to my page</h1>
        <p>This is some text</p>
    </body>
</html>
```

HyperText Markup Language (HTML)

Elements are the building blocks of an HTML page.
Some common elements include:

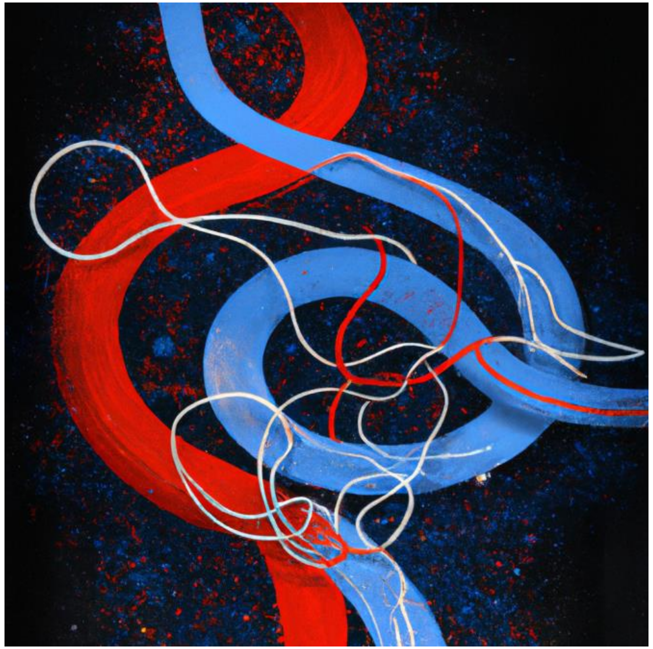
- **`<h1>` to `<h6>`:** headings
- **`<p>`:** paragraph
- **`<a>`:** hyperlink
- **`<div>`:** division or container
- **`<img>`:** image

Elements can also have attributes, which provide additional information about the element.

```
<img src="image.jpg" alt="A description of the image">
```

Cascading Style Sheets (CSS)

- It is a style sheet language used for describing the look and formatting of a document written in HTML
- CSS is used to control the layout, font, color, and other visual aspects of an HTML document


Made by DALL-E

Cascading Style Sheets (CSS)

CSS works by selecting elements in an HTML document and applying styles to them.

```css
h1 {
    color: red;
    font-size: 24px;
}

p {
    color: blue;
    font-size: 18px;
}
```

Cascading Style Sheets (CSS)

There are several ways to select elements in an HTML document using CSS. Some common selectors include:

- **`element:`** selects all elements with the given element name
- **`.class:`** selects all elements with the given class name
- **`#id:`** selects the element with the given id
- **`element element:`** selects all elements that are descendants of the given element
- **`element > element:`** selects all elements that are children of the given element

Cascading Style Sheets (CSS)

There are three ways to apply CSS to an HTML document:

```html
<!-- Inline -->
<h1 style="color: red; font-size: 24px;">Heading</h1>

<!-- Internal -->
<style>
    h1 {
        color: red;
        font-size: 24px;
    }
</style>

<!-- External -->
<link rel="stylesheet" href="style.css">
```

Inspect

The Inspect Element feature in web browsers allows you to view and modify the HTML and CSS of a web page in real-time. This can be useful for debugging, testing, and learning how a web page is constructed.

1. Open the web page you want to inspect.
2. Right-click on the page and select "*Inspect*" or "*Inspect Element*" from the context menu. Alternatively, you can use the keyboard shortcut:
    - Windows: *F12* or *Ctrl + Shift + I*
    - Mac: *Cmd + Opt + I*
3. The Inspector panel will open. It consists of two main sections: the *Elements* tab and the *Styles* tab.

I WILL FIND YOUR DATA
AND I WILL SCRAPE IT

03

BeautifulSoup

Introduction

The BeautifulSoup library was named after a Lewis Carroll poem of the same name in *Alice's Adventures in Wonderland*.

Like its Wonderland namesake, BeautifulSoup tries to **make sense of the nonsensical**; it helps format and organize the messy web by fixing bad HTML and presenting us with easily traversable Python objects representing XML structures. *(Mitchell, 2018)*

Beautiful Soup, so rich and green,
Waiting in a hot tureen!
Who for such dainties would not stoop?
Soup of the evening, beautiful Soup!

Installation

- BeautifulSoup4 requires **Python 3.6+**

- You can install BeautifulSoup4 and its dependencies from PyPI with:

```
pip install beautifulsoup4
```

- More information: https://www.crummy.com/software/BeautifulSoup/

Let’s talk with code!

File name: beautiful_soup.ipynb

Handling Exceptions

One of the most frustrating experiences in web scraping is to go to sleep with a scraper running, dreaming of all the data you'll have in your database the next day—only to find that the scraper hit an error on some unexpected data format and stopped execution shortly after you stopped looking at the screen. *(Mitchell, 2018)*

Made by DALL-E

Handling Exceptions: *requests*

HTTPError

HTTP error indicate that something went wrong on the server while processing the request.

```python
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()
except HTTPError as http_err:
    print('HTTP error occurred: {}'.format(http_err))
```

Handling Exceptions: *requests*

ConnectionError

Raised when a connection to the server could not be established.

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()
except HTTPError as http_err:
    print('HTTP error occurred: {}'.format(http_err))
except ConnectionError as conn_err:
    print('Connection error occurred: {}'.format(conn_err))
```

Handling Exceptions: *requests*

Timeout

The connection or server times out.

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()
except HTTPError as http_err:
    print('HTTP error occurred: {}'.format(http_err))
except ConnectionError as conn_err:
    print('Connection error occurred: {}'.format(conn_err))
except Timeout as timeout_err:
    print('Timeout error occurred: {}'.format(timeout_err))
```

Handling Exceptions: *requests*

RequestException

There was an ambiguous exception that occurred while handling your request.

```
try:
    response = requests.get(url, timeout=5)
    response.raise_for_status()
except HTTPError as http_err:
    print('HTTP error occurred: {}'.format(http_err))
except ConnectionError as conn_err:
    print('Connection error occurred: {}'.format(conn_err))
except Timeout as timeout_err:
    print('Timeout error occurred: {}'.format(timeout_err))
except RequestException as req_err:
    print('Request error occurred: {}'.format(req_err))
```

Handling Exceptions: *BeautifulSoup*

Every time you access a tag in a BeautifulSoup object, it's smart to add a check to make sure the tag actually exists.

If you attempt to access a tag that does not exist, BeautifulSoup will return a **`None`** object.

The problem is, attempting to access a tag on a **`None`** object itself will result in an **`AttributeError`** being thrown. *(Mitchell, 2018)*

```
try:
    badContent = bs.nonExistingTag.anotherTag
except AttributeError as e:
    print('Tag was not found!')
else:
    if badContent == None:
        print('Tag was not found!')
    else:
        print(badContent)
```

Regular Expressions (regex)

BeautifulSoup and regular expressions go hand in hand when it comes to scraping the web. In fact, most functions that take in a string argument (e.g., `find(id="aTagIdHere")`) will also take in a regular expression just as well. *(Mitchell, 2018)*

```
import re
url = 'http://www.pythonscraping.com/pages/page3.html'
html = requests(url)
bs = BeautifulSoup(html, 'html.parser')
images = bs.find_all('img’,
    {'src':re.compile('\.\.\/img\/gifts/img.*\.jpg')})
for image in images:
    print(image['src'])
```

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

Lambda Expressions

BeautifulSoup allows you to pass certain types of functions as parameters into the `find_all` function.
The only restriction is that these functions must take a tag object as an argument and return a boolean.
Every tag object that BeautifulSoup encounters is evaluated in this function, and tags that evaluate to `True` are returned, while the rest are discarded

```
bs.find_all(lambda tag: len(tag.attrs) == 2)
```

```
<div class="body" id="content"></div>
<span style="color:red" class="title"></span>
```

A PRACTICAL EXAMPLE

Scraping jobs from *Quera Magnet*

Folder name: quera_magnet

04

Scrapy

Introduction

Scrapy is an open-source web crawling framework written in Python. It is used to extract data from websites and store it in the desired format.

Feature	Scrapy	BeautifulSoup
Speed	Fast (able to crawl multiple pages concurrently)	Slow (single threaded)
Handling and following links	Built-in mechanism for handling and following links	No built-in mechanism for handling and following links
Extensibility	Highly extensible (middlewares and extensions)	Limited extensibility
Data cleaning and manipulation	Built-in support for data cleaning and manipulation	Limited support for data cleaning and manipulation
Community and documentation	Large and active community, with extensive documentation	Active community, with good documentation

Installation

- Scrapy requires **Python 3.7+**
- To install Scrapy using `conda`, run:

```
conda install -c conda-forge scrapy
```

- You can install Scrapy and its dependencies from PyPI with:

```
pip install Scrapy
```

- More information: https://docs.scrapy.org/en/latest/intro/install.html

Initialization

Once Scrapy is installed, you can create a new Scrapy project by running the following command:

```
scrapy startproject myproject
```

This will create a new directory called `myproject` with the following structure:

```
myproject/
    scrapy.cfg
    myproject/
        __init__.py
        items.py
        middlewares.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
```

The `items.py` file is where you will define the items that you want to scrape.

XPATH

XPATH is particularly useful for selecting elements with complex, nested structures, because it allows you to select elements based on their position in the document hierarchy.

Selecting the h2 element that is a child of the div element with the class col-md-6.

```
//div[@class="col-md-6"]/h2
```

Selecting all the a elements that have the attribute target with the value "_blank".

```
//a[@target="_blank"]
```

Selecting the third div element

```
//div[3]
```

```html
<html>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-6">
          <h2>Title</h2>
          <p>Description</p>
          <a href="http://www.example.com">Link</a>
        </div>
      </div>
    </div>
  </body>
</html>
```

robot.txt

A **robots.txt** file is used primarily to manage crawler traffic to a site, and usually to keep a file off search engines like Google, depending on the file type.

To bypass the `robot.txt` file in Scrapy, you can use the `ROBOTSTXT_OBEY` setting and set it to `False`.

```
# Obey robots.txt rules
ROBOTSTXT_OBEY = False
```

Made by DALL-E

Pipeline

After an item has been scraped by a spider, it is sent to the Item Pipeline which processes it through several components that are executed sequentially.

Each item pipeline component (sometimes referred as just “Item Pipeline”) is a Python class that implements a simple method. They receive an item and perform an action over it, also deciding if the item should continue through the pipeline or be dropped and no longer processed.

Don’t forget to enable the pipeline in the Scrapy settings by adding it to the ITEM_PIPELINES setting.

A PRACTICAL EXAMPLE

Scraping books from *Bidgol Publishing*

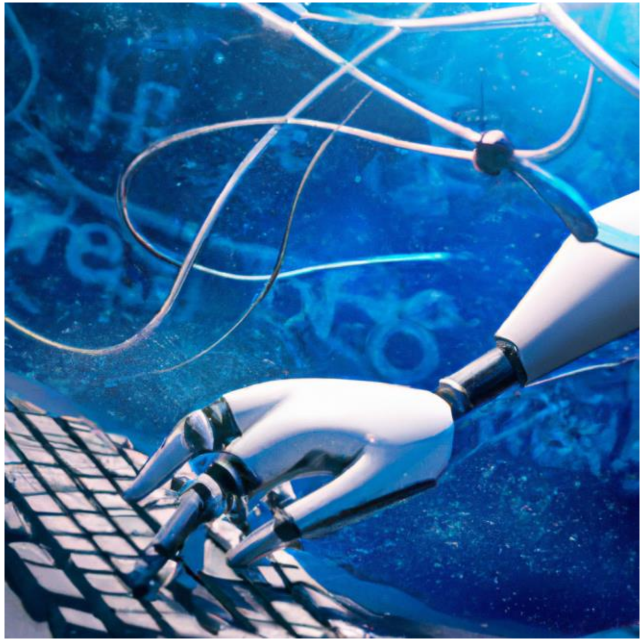
Folder name: bidgol

05

Forms and Logins

Introduction

One of the first questions that comes up when you start to move beyond the basics of web scraping is: “How do I access information behind a login screen?”
The web isincreasingly moving toward interaction, social media, and user-generated content. Forms and logins are an integral part of these types of sites and almost impossible to avoid. Fortunately, they are also relatively easy to deal with.


Made by DALL-E

A Basic Form

```html
<form method="post" action="processing.php">
  First name: <input type="text" name="firstname"><br>
  Last name:  <input type="text" name="lastname"><br>
  <input type="submit" value="Submit">
</form>
```

```python
import requests
params = {'firstname': 'Ryan', 'lastname': 'Mitchell'}
r = requests.post("http://pythonscraping.com/pages/processing.php", data=params)
print(r.text)
```

Other Types of Inputs

Standard HTML contains a wide variety of possible form input fields: radio buttons, checkboxes, and select boxes, to name a few. HTML5 adds sliders (range input fields), email, dates, and more. With custom JavaScript fields, the possibilities are endless, with color pickers, calendars, and whatever else the developers come up with next.

You need to worry about only two things: the **name** of the element and its **value**.
The element’s name can be easily determined by looking at the source code and finding the name attribute.
The value can sometimes be trickier, as it might be populated by JavaScript immediately before form submission.

Other Types of Inputs

Look at the URL of a site. If the URL is something like:

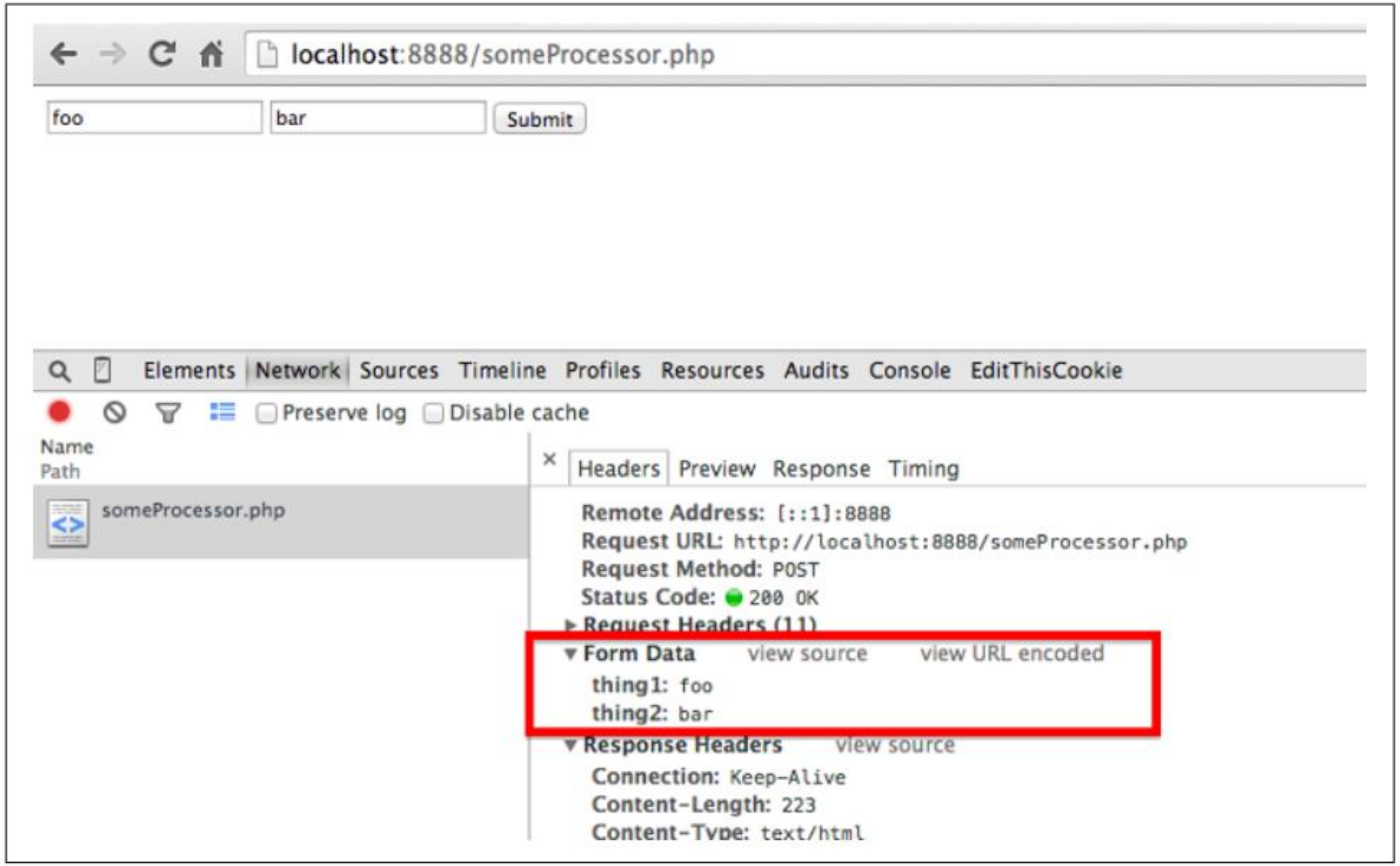
- http://domainname.com?thing1=foo&thing2=bar

You know that this corresponds to a form of this type:

```html
<form method="GET" action="someProcessor.php">
  <input type="someCrazyInputType" name="thing1" value="foo" />
  <input type="anotherCrazyInputType" name="thing2" value="bar" />
  <input type="submit" value="Submit" />
</form>
```

Other Types of Inputs

If you’re stuck with a complicated-looking POST form, and you want to see exactly which parameters your browser is sending to the server, the easiest way is to use your browser’s inspector or developer tool to view them.



```
Remote Address: [::1]:8888
Request URL: http://localhost:8888/someProcessor.php
Request Method: POST
Status Code: ● 200 OK
▶ Request Headers (11)
▼ Form Data    view source    view URL encoded
  thing1: foo
  thing2: bar
▼ Response Headers    view source
  Connection: Keep-Alive
  Content-Length: 223
  Content-Type: text/html
```

Login and Cookies

Most modern websites use cookies to keep track of who is logged in and who is not. After a site authenticates your login credentials, it stores them in your browser’s cookie, which usually contains a server-generated token, time-out, and tracking information.

Although cookies are a great solution for web developers, they can be problematic for web scrapers. You can submit a login form all day long, but if you don’t keep track of the cookie the form sends back to you afterward, the next page you visit will act as though you’ve never logged in at all.

Login and Cookies

```python
import requests

params = {'username': 'Ryan', 'password': 'password'}
r = requests.post('http://pythonscraping.com/pages/cookies/welcome.php',
params)
print('Cookie is set to:')
print(r.cookies.get_dict())
print('Going to profile page...')
r = requests.get('http://pythonscraping.com/pages/cookies/profile.php’,
                 cookies=r.cookies)
print(r.text)
```

Here you’re sending the login parameters to the welcome page, which acts as the processor for the login form. You retrieve the cookies from the results of the last request, print the result for verification, and then send them to the profile page by setting the cookies argument

Login and Cookies

What if you're dealing with a more complicated site that frequently modifies cookies without warning, or if you'd rather not even think about the cookies to begin with? The Requests **session** function works perfectly in this case:

```python
import requests
session = requests.Session()
params = {'username': 'username', 'password': 'password'}
s = session.post('http://pythonscraping.com/pages/cookies/welcome.php', params)
print('Cookie is set to:')
print(s.cookies.get_dict())
print('Going to profile page...')
s = session.get('http://pythonscraping.com/pages/cookies/profile.php')
print(s.text)
```

06

Selenium

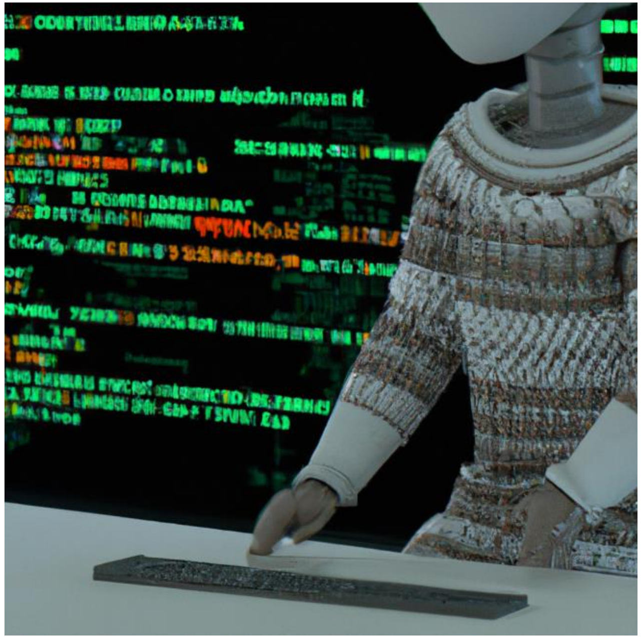
Introduction

Selenium is a tool that can be used for web scraping, as it allows you to control a web browser and interact with web pages programmatically.

Feature	Selenium	Scrapy
Requires web browser	Yes	No
Supports JavaScript	Yes	No
Speed and efficiency	Lower	Higher
Supports HTML parsing	Yes	Yes

Headless Browsers

A headless browser is a web browser that does not have a graphical user interface (GUI). Instead of rendering web pages and displaying them on a screen, a headless browser runs in the background and performs tasks such as loading web pages, clicking links, and filling out forms, without the need for a user to interact with it directly.



Made by DALL-E

Installation

- You can install BeautifulSoup4 and its dependencies from PyPI with:

```
pip install selenium
```

- Next, you will need to download the appropriate web driver for your web browser. Selenium supports a variety of web drivers for different browsers, including Chrome, Firefox, and Safari. You can download the web driver from the following link:
  - https://www.selenium.dev/downloads/

- Once you have downloaded the web driver, you will need to add it to your PATH environment variable so that Selenium can find it.

Initialization

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options

# instance of Options class allows us to configure Headless Chrome
options = Options()
# this parameter tells Chrome that it should be run without UI (Headless)
options.headless = True

# initializing webdriver for Chrome with our options
driver = webdriver.Chrome(options=options)

# Your Code

driver.quit()
```

Finding Elements

One of the most fundamental aspects of using Selenium is obtaining element references to work with. Selenium offers a number of built-in locator strategies to uniquely identify an element. Let’s consider this HTML snippet:

```html
<ol id="vegetables">
  <li class="potatoes">…
  <li class="onions">…
  <li class="tomatoes"><span>Tomato is a Vegetable</span>…
 </ol>
 <ul id="fruits">
   <li class="bananas">…
   <li class="apples">…
   <li class="tomatoes"><span>Tomato is a Fruit</span>…
 </ul>
```

Finding Elements: First Matching

When the **`find_element`** method is called on the driver instance, it returns a reference to the first element in the DOM that matches with the provided locator. This value can be stored and used for future element actions.

```
vegetable = driver.find_element(By.CLASS_NAME, "tomatoes")
```

```
fruits = driver.find_element(By.ID, "fruits")
fruit = fruits.find_element(By.CLASS_NAME,"tomatoes")
```

```
fruit = driver.find_element(By.CSS_SELECTOR,"#fruits .tomatoes")
```

Finding Elements: All Matching

There are several use cases for needing to get references to all elements that match a locator, rather than just the first one. The plural **`find_elements`** methods return a collection of element references. If there are no matches, an empty list is returned.

```
plants = driver.find_elements(By.TAG_NAME, "li")
```

```
# Get all the elements available with tag name 'p'
elements = driver.find_elements(By.TAG_NAME, 'p')

for e in elements:
    print(e.text)
```

Interactions

There are only 5 basic commands that can be executed on an element:

- **click** (applies to any element)
- **send keys** (only applies to text fields and content editable elements)
- **clear** (only applies to text fields and content editable elements)
- **submit** (only applies to form elements)
- **select**

Information

There are a number of details you can query about a specific element.

- **is_displayed():** This method is used to check if the connected Element is displayed on a webpage.

- **is_enabled():** This method is used to check if the connected Element is enabled or disabled on a webpage.

- **is_selected():** This method determines if the referenced Element is Selected or not. This method is widely used on Check boxes, radio buttons, input elements, and option elements.

Information

There are a number of details you can query about a specific element.

- **tag_name:** It is used to fetch the TagName of the referenced Element which has the focus in the current browsing context.

- **rect:** It is used to fetch the dimensions and coordinates of the referenced element. (*Returns height, width, x and y coordinates referenced element*)

- **value_of_css_property('prop'):** Retrieves the value of specified computed style property of an element in the current browsing context.

Information

There are a number of details you can query about a specific element.

- **text:** Retrieves the rendered text of the specified element.
- **get_attribute('attr'):** Fetches the run time value associated with a DOM attribute. It returns the data associated with the DOM attribute or property of the element.

```
# Identify the email text box
email_txt = driver.find_element(By.NAME, "email_input")

# Fetch the value property associated with the textbox
value_info = email_txt.get_attribute("value")
```

Wait

- In Selenium, the **wait** function is used to pause the execution of a script until a certain condition is met. This can be useful when interacting with a website, as it allows the script to wait for elements to load or for actions to complete before continuing.

- There are several types of wait functions available in Selenium, including **implicit** wait, **explicit** wait, and **fluid** wait.

Implicit Wait

An implicit wait tells Selenium to poll the DOM (Document Object Model) for a certain amount of time when trying to find an element or elements. If the element is not found within the specified time, a NoSuchElementException is raised. This type of wait is useful for cases where elements may take some time to appear on the page, but not necessarily a fixed amount of time.

```
# Set an implicit wait of 10 seconds
driver.implicitly_wait(10)

# Find an element
element = driver.find_element_by_id("example")
```

Explicit Wait

An explicit wait is a more flexible type of wait that allows you to specify a condition to wait for, as well as a maximum amount of time to wait. If the condition is not met within the specified time, a TimeoutException is raised. This type of wait is useful for cases where you need to wait for a specific condition to be met, such as an element becoming visible or clickable.

```python
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Set a wait of 10 seconds
wait = WebDriverWait(driver, 10)
# Wait for an element to become visible
element = wait.until(EC.visibility_of_element_located((By.ID, "example")))
```

Let’s code!

File name: duckduckgo.py

07

Techniques

Techniques

01 Timing

02 Adjust your header

03 Proxy Pool

04 TOR

05 CAPTCHA

06 HoneyPots

User Agents

A user agent is a string that a web browser or app sends to a web server along with every request to identify itself and its capabilities. A user agent can also include information about the operating system, device type, and even the browser version being used.

The easiest way to change the default Scrapy user-agent is to set a default user-agent in your `settings.py` file.

```
USER_AGENT = 'Mozilla/5.0 (iPad; CPU OS 12_2 like Mac OS X)
             AppleWebKit/605.1.15 (KHTML, like Gecko)
             Mobile/15E148'
```

User Agents: Rotating in Scrapy

Rotating through user-agents is also pretty straightforward, and we need a list of user-agents in our spider and use a random one with every request we make

```
import random
USER_AGENTS = [
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3',
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.1.2 Safari/605.1.15',
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0’]

class MySpider(scrapy.Spider):
    def start_requests(self):
        user_agent = random.choice(USER_AGENTS)
        yield scrapy.Request(url='http://example.com’,
                             headers={'User-Agent': user_agent})
```

Techniques

01 Timing

02 Adjust your header

03 Proxy Pool

04 TOR

05 CAPTCHA

06 HoneyPots

08

Ethics

Ethics

01 robots.txt

02 Infrastructure

03 Copyright

THANKS!

DO YOU HAVE ANY QUESTIONS?

parsa.abbasi1996@gmail.com

GitHub

https://github.com/parsa-abbasi/WebScraping

RESOURCES

Books:

- Broucke, S. v., & Baesens, B. (2018). Practical Web Scraping for Data Science: Best Practices and Examples with Python. Apress.
- Mitchell, R. (2018). Web Scraping with Python: Collecting More Data from the Modern Web. O'Reilly Media.

Papers:

- Thapelo, T. S., Namoshe, M., Matsebe, O., Motshegwa, T., & Bopape, M. J. M. (2021). SASSCAL webSAPI: A web scraping application programming interface to support access to SASSCAL's weather data. Data Science Journal, 20(1), [24]. https://doi.org/10.5334/dsj-2021-024

RESOURCES

Documentations:

- **requests:** https://requests.readthedocs.io/en/latest/
- **BeautifulSoup:** https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- **Scrapy:** https://docs.scrapy.org/en/latest/
- **Selenium:** https://www.selenium.dev/documentation/