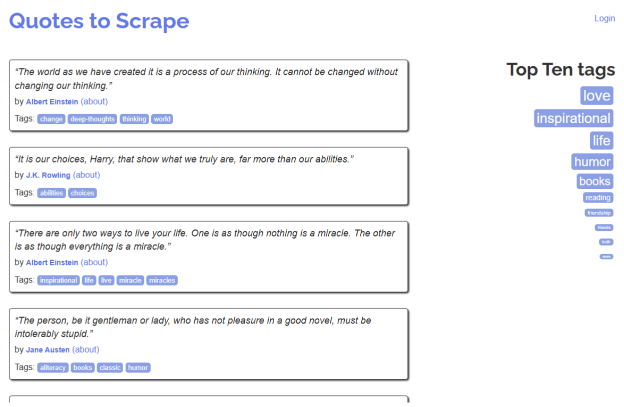
WEB SCRAPING CHEAT SHEET
(BeautifulSoup, Selenium, Scrapy and Splash)

BeautifulSoup

TEST DATA

Throughout the cheat sheet, if examples are needed, we will use the following site unless otherwise stated:

https://quotes.toscrape.com/



Quotes to Scrape

Login

"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."
by Albert Einstein (about)
Tags: change deep-thoughts thinking world

"It is our choices, Harry, that show what we truly are, far more than our abilities."
by J.K. Rowling (about)
Tags: abilities choices

"There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."
by Albert Einstein (about)
Tags: inspirational life live miracle miracles

"The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."
by Jane Austen (about)
Tags: aliteracy books classic humor

Top Ten tags
love
inspirational
life
humor
books
reading
friendship
friends
truth

HTML PRELIMINARIES

A typical HTML element looks like the following:

$$\underbrace{<\text{h1}}_{\text{opening tag}}\ \underbrace{\text{class} = \text{"title"}>}_{\text{attributes}}\ \underbrace{\text{My Title}}_{\text{content}}\ \underbrace{</\text{h1}>}_{\text{closing tag}}$$

HTML files are built using elements with various tags. Some common examples are:

- head
- body
- header
- article
- p - very common! Used for paragraphs
- h1,h2,...,h6 - varying heading levels (h1 is the largest)
- div - division/container
- nav - used for navigation links
- ul - unordered list
- li - a list item
- a - anchor (hyperlink), usually contains a "href" attribute
- button
- table
- tr - a table row
- td - table data (a cell in a table, a child of a tr element)

For example, the following table

Bob	Anna
20	25

is given by the HTML:

```
<table border="1">
    <tr>
        <td>Bob</td>
        <td>Anna</td>
    </tr>
    <tr>
        <td>20</td>
        <td>25</td>
    </tr>
</table>
```

Some ways to find HTML elements

When scraping sites, we make use of the HTML elements to find the information we want. Some ways to find elements are:

1. ID (this is an attribute, unique to the element)
2. Class Name (an attribute, not unique)
3. Tag Name (i.e. the tag of the element)
4. XPath (see later, doesn't work with BeautifulSoup).

IMPORTS AND INSTALLS

Begin the script with the following imports

```
from bs4 import BeautifulSoup
import requests
```

We will also need an HTML parser. For example, install **lxml** using pip.

THREE BASIC STEPS WHEN SCRAPING A SITE

1. Fetch page (makes an HTTP request to the URL and retrieves the content of the page):

   ```
   result = requests.get(my_url:str)
   ```

2. Get page content (the return above returns the HTML content and a bunch of other information, this step extracts the HTML from it)

   ```
   content = result.text
   ```

3. Create a "Soup" (creates a BeautifulSoup object from the HTML)

   ```
   soup = BeautifulSoup(content,
                        'lxml')
   ```

FINDING ELEMENTS WITH BEAUTIFULSOUP

Once we have a “Soup" object (called ‘soup’ say) we can **find the first occurrence** of an HTML element by id, class or tag as follows:

- by ID - finding an element by its ID attribute

  ```
  soup.find(id="example_id")
  ```

- by Class (notice the underscore in the argument)

  ```
  soup.find(class_="example_class")
  ```

- by Tag - pass the tag to the first argument. Unlike the other two, this is a positional argument, not a keyword argument

  ```
  soup.find("example_tag")
  ```

To instead **find all occurrences** of elements with a particular property, use the find_all method instead of find. For example:

```
soup.find_all(class_="example_class")
```

The find method returns BeautifulSoup objects (and find_all returns a list of BeautifulSoup objects), so we may use the find and find_all methods with them.
For example, to find the first occurrence of a table, and then find all rows in that table we can do

```
table = soup.find(‘table’)
rows = table.find_all(‘tr’)
```

To extract the text/content from a HTML element we use the get_text method. For example, to extract the text from the main (h1) heading we do

```
header_text = soup.find(‘h1’).get_text()
```

SCRAPING MULTIPLE LINKS WITHIN A PAGE

Often, we want to scrape a page for all links, then extract some common information from those links. This can be achieved as follows:

1. Make a soup from the HTTP request.
2. Find all anchors in the HTML using the find_all method

   ```
   soup.find_all(‘a’,href=True)
   ```

3. Loop over all soup elements in the return, extracting the href from them. This is done as follows

   ```
   for ele in soup.find_all(‘a’,href=True):
       ele[‘href’] #some logic
   ```

4. Add some logic, usually starting with making a request and a creating a soup using the new href.

Two warnings/things to note:

- The links may be relative links, so may have to concatenate with the original URL (i.e. the root of the site).
- Some websites may not allow multiple requests in a short period of time. If so, ensure to **import time** and use **time.sleep(s : int)** within loops to wait ‘s’ seconds between passes.

PAGINATION

We can scrape sites where the information is spread over multiple pages (navigated between using a button system with a next page button).

With BeautifulSoup, this is somewhat ad-hoc.

- If the "next page" HTML button element has a consistent format.
  - Right-click on webpage and inspect, view the HTML of the next page button.
  - Commonly, this can be an anchor (tag a) inside a list item (li) inside a ul (unordered list).
- Sometimes the site follows a pattern with its pages. For example "url?page=1", "url?page=2" etc. If so, this can be made use of.

Selenium

XPATH BASICS

We will make use of XPath to scrape data from sites using Selenium, some basic syntax is as follows:

- Select all elements with a certain tag via //tag_name.
  For example, //h1, //table.
- Select the n^th element with a certain tag using //"tag_name"[n] (indexing starts from 1).
- Select elements with a certain tag and attribute using //"tag_name"[@Attribute_name="value"]. E.g.

  ```
  //h1[@class='main_header'].
  ```

- There are functions such as contains() or starts-with() that can be used alongside attributes.
  For example, to find all h2's with class attribute value starting with 'secondary' we use the following:

  ```
  //h2[starts-with(@class, 'secondary')]
  ```

- Special Characters:
  - / - select direct children of the current element(s).
    For example, to select the <p> children of the <div> elements in a HTML document we can do //div/p
  - /.. - selects the parent node of an element.
    For example, to obtain the parent of the first paragraph element in a document, we can do //p[1]/..
  - * - select all direct child nodes of an element.
    For example, to select all siblings of a node (including itself) we can do node/../*, as this first obtains the parent of 'node', and then obtains its direct children.

IMPORTS

Selenium is a browser automation tool, that allows us to scrape Javascript-driven sites.

Every script making use of selenium must have the following imports

```
from selenium import webdriver
from selenium.webdriver.chrome.service import
    Service
```

SETTING UP THE WEB DRIVER

To scrape sites, a web driver must be set up.

As of Selenium 4.6, there is no need to download a Chrome Driver (or similar) and supply the path as an argument to the initialisation, instead, the Service import takes care of this behind the scenes.

```
driver = webdriver.Chrome(service=Service())
```

To get data from a website with URL "my_url" we do:

```
driver.get(my_url)
```

By default, this driver will open the webpage, scrape it, and close it again. This can be modified by passing some options to the initialisation of the driver. To keep the webpage open, set the options as follows:

```
options = webdriver.ChromeOptions()
options.add_experimental_option("detach",True)
```

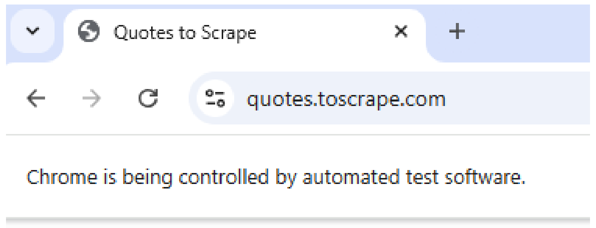
Alternatively, the script can be performed without displaying the webpage GUI at all. To enable this, use

```
options.headless = True
```

Options should be passed to the "options" kwarg of the webdriver when initilisating. e.g.

```
driver = webdriver.Chrome(service=Service(),
    options=options)
```

When visiting a site using the get method of the driver, a new Chrome window will open and you will see the following message



Chrome is being controlled by automated test software.

Once you are finished scraping, ensure to close any opened windows using

```
driver.quit()
```

SCRAPING METHODS/FINDING ELEMENTS

We can find elements using the "By" method, to use this we need to import it

```
from selenium.webdriver.common.by import By
```

We can find elements using

- ID (By.ID)
- Name (By.NAME)
- XPath (By.XPATH)
- Tag name (By.TAG_NAME)
- Class name (By.CLASS_NAME)
- CSS Selector (By.CSS_SELECTOR)

To find elements, use the find_element() method of the web driver. To find all occurrences of an element, use find_elements() instead.

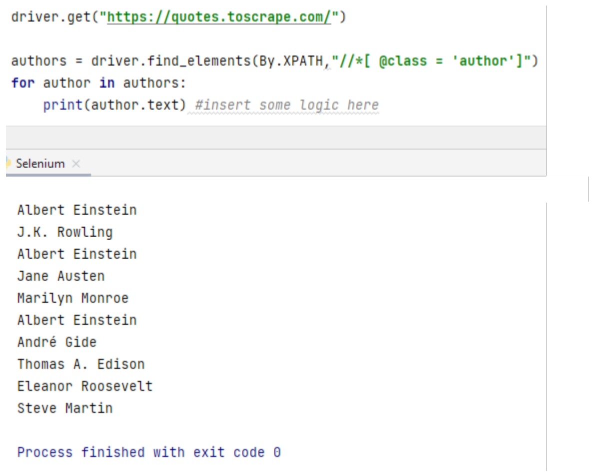
For example, to find all elements with class 'author' in our test site (i.e. quotes.toscrape.com) we can do:

```
driver.find_elements(By.CLASS_NAME, "author")
```

or

```
driver.find_elements(By.XPATH, "//*[@class='author']")
```

An example of this being used is as follows:



```
driver.get("https://quotes.toscrape.com/")

authors = driver.find_elements(By.XPATH,"//*[ @class = 'author']")
for author in authors:
    print(author.text) #insert some logic here
```

```
Albert Einstein
J.K. Rowling
Albert Einstein
Jane Austen
Marilyn Monroe
Albert Einstein
André Gide
Thomas A. Edison
Eleanor Roosevelt
Steve Martin

Process finished with exit code 0
```

Note that the return value of driver.find_element(...) is a WebElement object. These objects also have find_element(s) methods and so can be used in place of "driver" to limit the scope of the search.

CLICKING A BUTTON

To click buttons, you need a way to identify them using HTML. This could be done using tags, classes, IDs etc.

Once found, for example,

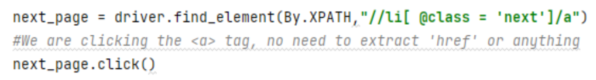
```
button = driver.get_element(By.ID, "next_pg_button")
```

it can be pressed using the click() method i.e.

```
    button.click()
```

In the automated browser window, you will see a new webpage load (more on how to tell the code to wait for the new webpage to load later).

In our test data, the next page button is a child of a list item (li) with class "next", so we press the button as follows (notice that it is the anchor tag we are clicking):



```
next_page = driver.find_element(By.XPATH,"//li[ @class = 'next']/a")
#We are clicking the <a> tag, no need to extract 'href' or anything
next_page.click()
```

EXTRACTING DATA FROM TABLES

Recall that table elements are made up of tags

- <tr> - a row of data.
- <td> - a cell of data within a row, these are children of <tr> elements.

For example, a table with one row and three columns containing data a,b,c will look as follows:

```
    <table>
        <tr>
            <td>a</td>
            <td>b</td>
            <td>c</td>
        </tr>
    </table>
```

To extract all data from a table, we can do something like

```
my_table = driver.find_element(By.TAG_NAME, "table")
rows = my_table.find_elements(By.TAG_NAME, "tr")
for row in rows:
    row_data = row.find_elements(By.TAG_NAME, 'td')
    #do something with the data, e.g. to print the
        text contents do
    print([cell.text for cell in row_data])
```

SELECT AN OPTION FROM A DROPDOWN MENU

To select from a dropdown (an element of type select or option), we need to locate it and make it a Select object. First, we need to import Select.

```
    from selenium.webdriver.support.ui import Select
```

Suppose we have the following HTML

```
    <select id="dropdown">
        <option value="1">Option 1</option>
        <option value="2">Option 2</option>
        <option value="3">Option 3</option>
    </select>
```

We first locate the dropdown and make it a Select object

```
    dropdown = driver.find_element(By.ID, "dropdown")
    select = Select(dropdown)
```

We can either select by visible text (using the select_by_visible_text method) or by position (using the select_by_index method, where indexing starts from 0).

For example, to select Option 2 we can do either of

```
    select.select_by_visible_text("Option 2")
    #or
    select.select_by_index(1)
```

IMPLICIT AND EXPLICIT WAIT TIMES

Explicit Waits

For example, "I want to wait 2 seconds after loading new pages".

This is achieved using the time module. For example, to wait 2 seconds in a script you would do the following:

```
    import time
    time.sleep(2)
```

These waits are very easy to implement, however, they can lead to errors if the wait time is not long enough, and the next webpage has not fully loaded.

Implicit Waits

These waits can be set to wait for a button to be pressable, an element to be visible etc. They are implemented using the WebDriverWait and expected_conditions classes, as well as "By". These are imported as follows

```
    from selenium.webdriver.support.ui import
        WebDriverWait
    from selenium.webdriver.support import expected_
        conditions as EC
```

expected_conditions has many available functions, some are:

- element_to_be_clickable
  The argument is a so-called locator (i.e. a way to find the element), it is a tuple, usually with a "By component" and a value.
  For example (By.TAG_NAME, "my_tag").
- presence_of_element_located
  The argument is a locator (same idea as above).

An example of an implicit wait is as follows:

```
WebDriverWait(driver,max_secs:int).until(EC.presence_
    of_element_located((By.CLASS_NAME, "my_class")))
```

This will wait for a maximum of max_secs. It is waiting for an element with class name = "my_class" to be located on the page.

(INFINITE) SCROLLING

To scroll through the webpage and load the contents we make use of scrolling and returning the page height. We create a loop and keep scrolling whilst the page height keeps getting longer (as this implies new content is loading). This can be done as follows:

```
    # Get initial page height
    last_height = driver.execute_script("return
        document.body.scrollHeight")

    while True:
        # Scroll down          driver.execute_script("
            window.scrollTo(0, document.body.
            scrollHeight);")

        # Wait for new content to load
        time.sleep(3)

        # Calculate new scroll height
        new_height = driver.execute_script("return
            document.body.scrollHeight")

        # Check if page height has changed
        if new_height == last_height:
            break  # If no new content, exit the loop

        last_height = new_height
```

ENTERING DATA

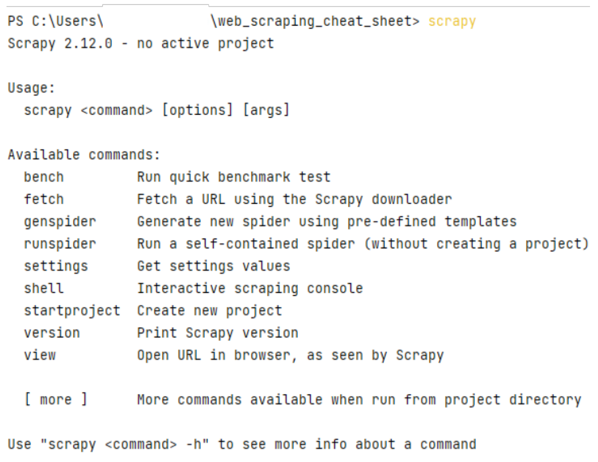
To input data we use the send_keys method. This is done as follows (assuming inbox_box is the input element).

```
input_box = driver.find_element(By.ID, "input_box")

input_box.send_keys("your_username")
```

Scrapy

TERMINAL INPUTS AND SETUP

Type "scrapy" in the terminal to see all available commands (start with "pip install scrapy" if needed):

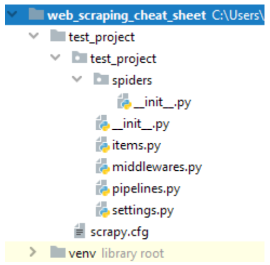


```
PS C:\Users\            \web_scraping_cheat_sheet> scrapy
Scrapy 2.12.0 - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  bench         Run quick benchmark test
  fetch         Fetch a URL using the Scrapy downloader
  genspider     Generate new spider using pre-defined templates
  runspider     Run a self-contained spider (without creating a project)
  settings      Get settings values
  shell         Interactive scraping console
  startproject  Create new project
  version       Print Scrapy version
  view          Open URL in browser, as seen by Scrapy

  [ more ]      More commands available when run from project directory

Use "scrapy <command> -h" to see more info about a command
```

To start a new project, type

```
scrapy startproject test_project
```

(where test_project is replaced with your project name) into the terminal, this creates a file structure as follows:

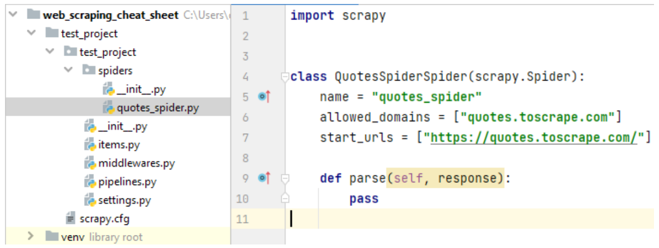


```
web_scraping_cheat_sheet C:\Users\
  test_project
    test_project
      spiders
        __init__.py
      __init__.py
      items.py
      middlewares.py
      pipelines.py
      settings.py
    scrapy.cfg
  venv library root
```

In the terminal, cd to the newly created project directory and create a new spider, you will give the spider a name and a website URL. This is done as follows:

```
cd test_project
...
scrapy genspider quotes_spider https://quotes.
    toscrape.com/
```

This creates the quotes_spider.py file in the test_project/spiders directory:



```
web_scraping_cheat_sheet C:\Users\
  test_project
    test_project
      spiders
        __init__.py
        quotes_spider.py
      __init__.py
      items.py
      middlewares.py
      pipelines.py
      settings.py
    scrapy.cfg
  venv library root
```

```python
import scrapy


class QuotesSpiderSpider(scrapy.Spider):
    name = "quotes_spider"
    allowed_domains = ["quotes.toscrape.com"]
    start_urls = ["https://quotes.toscrape.com/"]

    def parse(self, response):
        pass

```

MODIFYING AND RUNNING THE SPIDER

To customise the spider (so it scrapes the site), we modify the parse function.

Note: When returning multiple pieces of data e.g. a row of a table one at a time, use ‘yield‘ instead of ‘return‘.

Note: Scrapy expects the returned/yielded objects to be requests, dictionaries or None. you cannot yield a string, for example.

The ‘response‘ argument is a scrapy.http.Response object. We will use the .xpath() method to find elements.

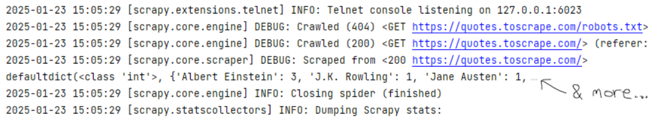
For example, to extract the number of times each author appears on page 1 of the quotes site, we would modify the parse function to:

```python
def parse(self, response):
    author_count = defaultdict(int)
    for quote in response.xpath('//div[@class="quote"]
        '):
        author_name = quote.xpath('span/small[@class="
            author"]/text()').get()
        author_count[author_name] += 1
    yield author_count
```

To run the spider, go to the terminal, and whilst inside the test_project directory, type

```
scrapy crawl quotes_spider
```

The output will be displayed to the terminal. It contains a bunch of information/logs, and amongst it you will see



```
2025-01-23 15:05:29 [scrapy.extensions.telnet] INFO: Telnet console listening on 127.0.0.1:6023
2025-01-23 15:05:29 [scrapy.core.engine] DEBUG: Crawled (404) <GET https://quotes.toscrape.com/robots.txt>
2025-01-23 15:05:29 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://quotes.toscrape.com/> (referer:
2025-01-23 15:05:29 [scrapy.core.scraper] DEBUG: Scraped from <200 https://quotes.toscrape.com/>
defaultdict(<class 'int'>, {'Albert Einstein': 3, 'J.K. Rowling': 1, 'Jane Austen': 1, …
2025-01-23 15:05:29 [scrapy.core.engine] INFO: Closing spider (finished)
2025-01-23 15:05:29 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
```

ABSOLUTE V.S. RELATIVE REFERENCING

To work with relative links (e.g. "page/2/", with the whole URL being "https://quotes.toscrape.com/page/2/") there are three methods, with the latter two being preferred.

- Use f-strings.
  If the root is stored in a variable, we can form the whole URL via

  ```
  f"{root}/{relative}".
  ```

  This is very prone to errors e.g. missing a slash, or having two slashes side by side.

- Use urljoin. For example:

  ```
  response.urljoin(relative).
  ```

  This handles stitching together the root and relative URLs much better.

- Work directly with the relative one via follow. For example:

  ```
  response.follow(url=relative)
  ```

SCRAPING DATA FROM LINKS (.FOLLOW())

We may follow and scrape the contents of links on a page using follow, moreover, we can use custom parse functions for these pages.

Suppose we have a relative link called ‘link‘. We can scrape it using

```
response.follow(url='link',callback=self.custom_
    parse)
```

where custom_parse is a custom parse function:

```python
def custom_parse(self,response):
    #scrape instructions.
```

SAVING TO JSON/CSV

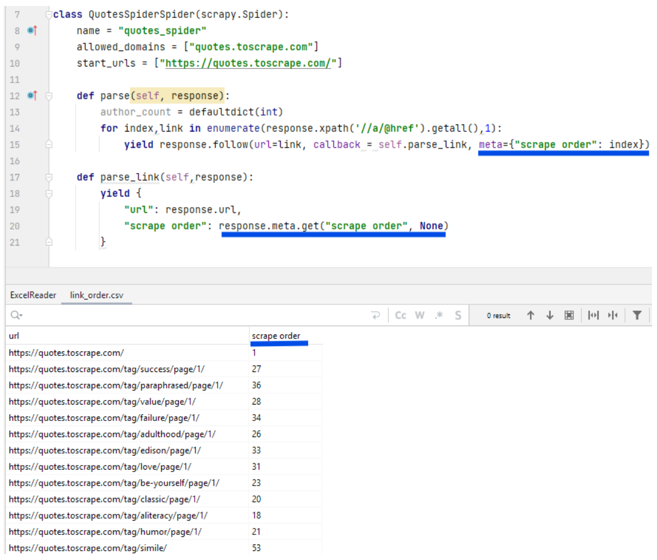
Currently, displaying the output of our scrapes to the terminal is not super helpful. For example, we cannot do any further data manipulation on it.

To do this, we simply append a command to the scrapy crawl command in the terminal. To Save to JSON or csv respectively, we do:

```
#to save to JSON
scrapy crawl quotes_spider -o my_json.json
#or to save to CSV
scrapy crawl quotes_spider -o my_csv.csv
```

If we wish to add a custom column for each scraped page (something like a tag/comment), we can pass data within the 'meta' argument of the 'follow' method, and then retrieve it in the custom parse function.

The following is an example of scraping each link from the homepage of the quotes to scrape page, and tagging each scrape with the order in which they were scraped.



```
class QuotesSpiderSpider(scrapy.Spider):
    name = "quotes_spider"
    allowed_domains = ["quotes.toscrape.com"]
    start_urls = ["https://quotes.toscrape.com/"]

    def parse(self, response):
        author_count = defaultdict(int)
        for index,link in enumerate(response.xpath('//a/@href').getall(),1):
            yield response.follow(url=link, callback = self.parse_link, meta={"scrape order": index})

    def parse_link(self,response):
        yield {
            "url": response.url,
            "scrape order": response.meta.get("scrape order", None)
        }
```

url	scrape order
https://quotes.toscrape.com/	1
https://quotes.toscrape.com/tag/success/page/1/	27
https://quotes.toscrape.com/tag/paraphrased/page/1/	36
https://quotes.toscrape.com/tag/value/page/1/	28
https://quotes.toscrape.com/tag/failure/page/1/	34
https://quotes.toscrape.com/tag/adulthood/page/1/	26
https://quotes.toscrape.com/tag/edison/page/1/	33
https://quotes.toscrape.com/tag/love/page/1/	31
https://quotes.toscrape.com/tag/be-yourself/page/1/	23
https://quotes.toscrape.com/tag/classic/page/1/	20
https://quotes.toscrape.com/tag/aliteracy/page/1/	18
https://quotes.toscrape.com/tag/humor/page/1/	21
https://quotes.toscrape.com/tag/simile/	53

USER AGENT

Currently, websites we scrape know that we are using Scrapy - this is bad practice in general. We want to change this so that it looks like we are sending requests from Chrome (i.e. browsing normally).
To see the current user agent, type the following into the terminal:

```
> scrapy shell https://quotes.toscrape.com
#wait...
>>> request.headers
```

and within the output dictionary, there is a key called 'User-Agent' containing the current user agent (by default, it will be something like 'Scrapy/2.12.0 (+https://scrapy.org)').

We need to find a new user agent, to do this open a Chrome window and follow these steps:

- Right-click and Inspect
- Go to the Network tab
- Refresh the page
- Click any item in the 'Name' column
- Find Request Headers -> User Agent
- e.g. Mozilla/5.0 (Linux; Android 6.0 ...

To change the user agent used by the spider:

- Go to settings.py
- Uncomment "DEFAULT_REQUEST_HEADERS" (located at approximately line 40).
- Add key "User-Agent", and paste the user agent copied from Chrome as the value.

EXPORT TO DATABASE

To scrape the export the data to a database (say a SQLite3 db), we need to create a connection to SQLite3, create a table, insert scraped data into the table, and then close the connection.

This is done inside the 'pipelines.py' file. To do this, we will create a class 'SQLitePipeline' in the 'pipelines.py' file, and it will have three methods, as follows:

- open_spider - This is called when the spider starts. We will use this function to setup the SQLite connection and create the table
- process_item - This method will be used to insert the scraped data into the table. This method in the pipeline is called every time an item is yielded from the main parse function (or any other callback function) in your Scrapy spider. As not every yield will contain data to be inserted in the table, we should wrap the logic in try/except blocks.
- close_spider - This is called when the spider ends. We will use this to close the connection to SQLite.

An example class in the pipelines.py file looks something like

```
import sqlite3
import logging

class SQLitePipeline:
    def open_spider(self, spider):
        # Establish a connection to a SQLite db
        self.connection = sqlite3.connect('scrapy_data
            .db')
        self.cursor = self.connection.cursor()

        # Create the table if it doesn't exist
        # Use SQL syntax here
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS scraped_data (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                column_one TEXT,
                column_two TEXT,
                column_three TEXT
            )
        ''')
        self.connection.commit()

    def process_item(self, item, spider):
        try:
            self.cursor.execute('''
                INSERT INTO scraped_data (column_one,
                    column_two, column_three)
                VALUES (?, ?, ?)
            ''', (item.get('field1'), item.get('field2
                '), item.get('field3')))
            self.connection.commit()

        # VERY IMPORTANT
        # In place of "field1", "field2", "field3",
            you should put the keys from the yielded
            dictionary in the parse function of the
            spider.
        # THESE MUST MATCH EXACTLY TO AVOID ERRORS.

        except sqlite3.DatabaseError as db_error:
            #handle the error, e.g.
            logging.error(f"Database error: {db_error}
                ")

        except Exception as e:
            # Catch any other exceptions, e.g.
            logging.error(f"Unexpected error: {e}")

        return item

    def close_spider(self, spider):
        # Close the database connection when the
            spider finishes
        self.connection.close()
```

Finally, inside the 'settings.py' file, change the ITEM_PIPELINES key to 'SQLitePipeline' (or whatever the class is called). e.g.

```
ITEM_PIPELINES = {
   "test_project.pipelines.SQLitePipeline": 300,
}
```

The numerical value is the priority. If multiple pipelines are being run, a lower number equals higher priority (so it will run first). This can be used to, for example, clean data (using a high-priority pipeline) before writing to a database.

SCRAPING APIS/HANDLING JSON DATA

To see API calls from a webpage, navigate to Inspect -> Network -> Fetch/XMR. Clicking on an item in the left-hand column (may need to refresh the webpage) shows details about it.

To handle JSON data (i.e. if the webpage consists of JSON data, such as the return of an API request), include logic similar to the following in the parse function:

```
import json

def parse(self, response):
    json_response = json.loads(response.text)
    #use 'get' method to extract values. e.g.
    my_value = json_response.get("my_key")
    #...
```

LOGIN TO A WEBSITE

To login to a website, we need to send a POST request containing the username and password to the login page.

For an example, we create a new spider whose start page is the login page quotestoscrape, i.e. https://quotes.toscrape.com/login , but of course, an existing spider can also handle this, where the login data is passed to the login page via response.follow.

Then the idea is to extract the CSRF token using XPath or otherwise (this looks something like dGlPJRuyAbhxFjiIH...). Then send a request using FormRequest.from_response.

This request should contain a callback to handle what should be done once login has occurred. The callback should start by verifying if the login was successful, for example, by checking for the existence of a logout button.

An example spider is as follows:

```
import scrapy
from scrapy.http import FormRequest

class QuotesLoginSpider(scrapy.Spider):
    name = 'quotes_login_spider'
    start_urls = ['https://quotes.toscrape.com/login']

    def parse(self, response):
        # Extract the CSRF token using XPath or
            otherwise
        csrf_token = response.xpath('//input[@name="
            csrf_token"]/@value').get()

        # Prepare the login data.
        login_data = {
            "csrf_token": csrf_token,
            "username": "my_username",
            "password": "my_password",
        }
        yield FormRequest.from_response(
            response,  # (containing the form element)
            formdata=login_data,
            callback=self.after_login
        )

    def after_login(self, response):
        # Verify login success by checking for the
            logout button
        if "Logout" in response.text:
            self.log("Login successful!")
            # Continue scraping here
        else:
            self.log("Login failed.")
```

Two things to note:

- Scrapy will use the first form it finds in the HTML response by default. If there are multiple forms consider adding a "formxpath" key to FormRequest.from_response to precisely locate the form.
- The keys in login_data should match the ‘name‘ attributes of the ‘input‘ elements in the HTML form.

Splash

Scrapy does not allow us to scrape JavaScript driven sites. For this, we need the help of Splash.

SETTING UP SPLASH WITH DOCKER

Download and Install Docker Desktop from docker.com. You will also need WSL.

Install splash using the terminal (e.g. just use the one inbuilt with Docker)

```
> docker pull scrapinghub/splash
```

To run Splash in the browser, input the following command into the terminal.

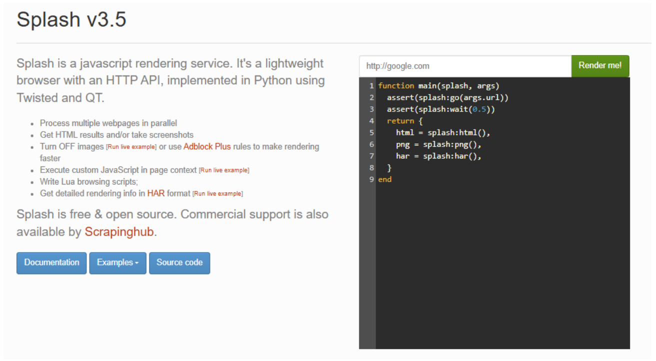
```
> docker run -p 8050:8050 scrapinghub/splash
```

This can be opened in the browser with the URL

```
http://localhost:8050/
```

and in the future splash can be found in the docker dashboard.

When opened in the browser, you see the following:



Splash v3.5

Splash is a javascript rendering service. It's a lightweight browser with an HTTP API, implemented in Python using Twisted and QT.

- Process multiple webpages in parallel
- Get HTML results and/or take screenshots
- Turn OFF images [Run live example] or use Adblock Plus rules to make rendering faster
- Execute custom JavaScript in page context [Run live example]
- Write Lua browsing scripts;
- Get detailed rendering info in HAR format [Run live example]

Splash is free & open source. Commercial support is also available by Scrapinghub.

```
function main(splash, args)
  assert(splash:go(args.url))
  assert(splash:wait(0.5))
  return {
    html = splash:html(),
    png = splash:png(),
    har = splash:har(),
  }
end
```

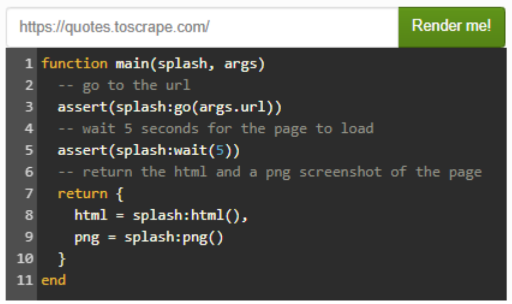
USING SPLASH IN THE BROWSER

Splash uses the Lua programming language.

To use, add the URL to the box above the script editor.

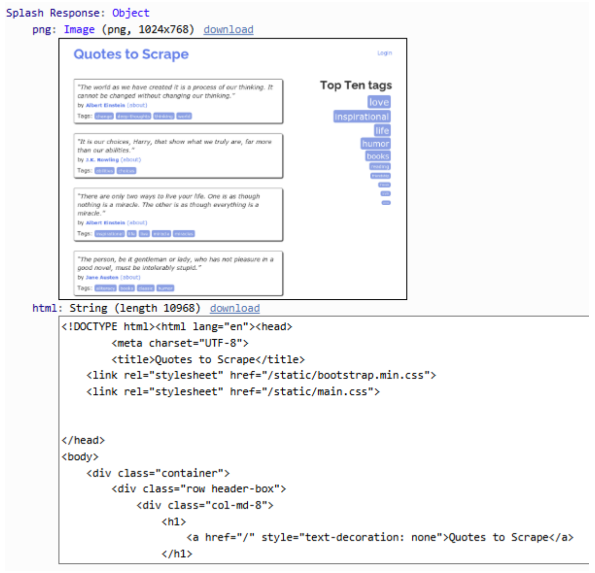
Here is an example of returning a screenshot of the quotestoscrape homepage as well as the HTML of the page.

Input:



```
function main(splash, args)
  -- go to the url
  assert(splash:go(args.url))
  -- wait 5 seconds for the page to load
  assert(splash:wait(5))
  -- return the html and a png screenshot of the page
  return {
    html = splash:html(),
    png = splash:png()
  }
end
```

Output:



```
Splash Response: Object
  png: Image (png, 1024x768) download
```

"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."
by Albert Einstein (about)

"It is our choices, Harry, that show what we truly are, far more than our abilities."
by J.K. Rowling (about)

"There are only two ways to live your life. One is as though nothing is a miracle. The other is as though everything is a miracle."
by Albert Einstein (about)

"The person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably stupid."
by Jane Austen (about)

```
  html: String (length 10968) download
<!DOCTYPE html><html lang="en"><head>
        <meta charset="UTF-8">
        <title>Quotes to Scrape</title>
    <link rel="stylesheet" href="/static/bootstrap.min.css">
    <link rel="stylesheet" href="/static/main.css">


</head>
<body>
    <div class="container">
        <div class="row header-box">
            <div class="col-md-8">
                <h1>
                    <a href="/" style="text-decoration: none">Quotes to Scrape</a>
                </h1>
```

FINDING ELEMENTS WITH SPLASH

HTML elements are found using CSS selector. The basics are:

- Use the element name directly to find them. e.g. "p" or "div"
- To find elements by their ID, use #id. For example, to find an element with id="main_title" use '#main_title'.
- Similarly, use .class to find elements by their class.

To find and store an element as a variable in Splash, we do

```
my_selection = assert(splash:select(<CSS selector
    >))
#For example
main_box = assert(splash:select("#main_box"))
```

Interaction with certain aspects of elements is only possible when focused on that element. To focus on an element do the following:

```
my_selection:focus()
```

To input text into in input box element, do the following:

```
my_input_box:send_text("my_text")
```

To click a button, do the following

```
my_button:mouse_click()
```

USING SCRAPY AND SPLASH TOGETHER

To use splash with scrapy we first need to install scrapy-splash

```
> pip install scrapy-splash
```

Then we need to modify the settings.py file. This is usually a standard copy-and-paste modification, as follows:

```
#In settings.py

# Splash settings
SPLASH_URL = 'http://localhost:8050'

# Enable Splash's middleware
DOWNLOADER_MIDDLEWARES = {
    'scrapy_splash.SplashCookiesMiddleware': 723,
    'scrapy_splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.
        HttpCompressionMiddleware': 810,
}

# Use Splash's duplicate filter
DUPEFILTER_CLASS = 'scrapy_splash.
    SplashAwareDupeFilter'

# Use Splash's cache storage
HTTPCACHE_STORAGE = 'scrapy_splash.
    SplashAwareFSCacheStorage'
```

In the spider file (e.g. quotes_spider.py), add a script variable within the class containing a Splash/Lua script.

For example, our script from before:

```
#In quotes_spider.py

script = """
  function main(splash, args)
    -- go to the url
    assert(splash:go(args.url))
    -- wait 5 seconds for the page to load
```

```
      assert(splash:wait(5))
      -- return the html and a png screenshot of the
          page
      return {
        html = splash:html(),
        png = splash:png()
      }
    end
  """
```

We also need to import SplashRequest, and create a ‘start_requests‘ function which will be called before our parse function (so Splash can do any rendering of JavaScript objects before our scrapy parse function scrapes the required data).

```
from scrapy-splash import SplashRequest

def start_requests(self):
  for url in self.start_urls:
      yield SplashRequest(
          url=url,
          callback=self.parse,
          endpoint="execute",
          args={'lua_source': self.script},
      )

#parse function defined as usual
```

CHANGE AGENT IN SPLASH

To change the user agent, add the line

```
  splash:set_user_agent("my_user_agent")
```

to the Lua script i.e. our script becomes

```
script = """
  function main(splash, args)
    splash:set_user_agent("Mozilla/5.0 (Windows NT
         10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
          like Gecko) Chrome/91.0.4472.124 Safari
         /537.36")
    -- go to the url
    assert(splash:go(args.url))
    -- etc...
"""
```