Day 3: More Selenium, Review Solutions, and wget

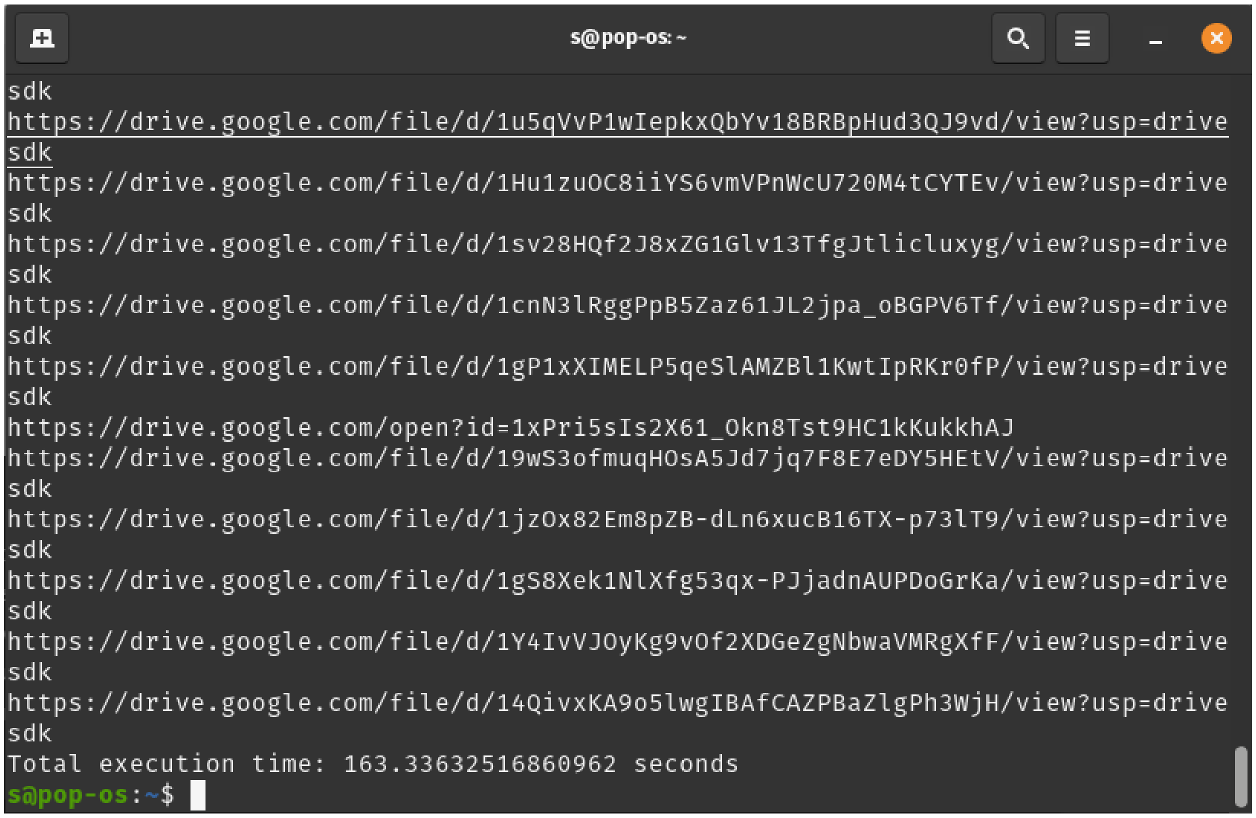
Requirements:
python 3.7+
pip

pip install selenium
geckodriver (instructions)

References:
Selenium Documentation

Activity 1: Review FOMB Scrape Solutions

Solution 1*: The complete & correct, but time costly way to scrape the site.*



```
sdk
https://drive.google.com/file/d/1u5qVvP1wIepkxQbYv18BRBpHud3QJ9vd/view?usp=drive
sdk
https://drive.google.com/file/d/1Hu1zuOC8iiYS6vmVPnWcU720M4tCYTEv/view?usp=drive
sdk
https://drive.google.com/file/d/1sv28HQf2J8xZG1Glv13TfgJtlicluxyg/view?usp=drive
sdk
https://drive.google.com/file/d/1cnN3lRggPpB5Zaz61JL2jpa_oBGPV6Tf/view?usp=drive
sdk
https://drive.google.com/file/d/1gP1xXIMELP5qeSlAMZBl1KwtIpRKr0fP/view?usp=drive
sdk
https://drive.google.com/open?id=1xPri5sIs2X61_Okn8Tst9HC1kKukkhAJ
https://drive.google.com/file/d/19wS3ofmuqHOsA5Jd7jq7F8E7eDY5HEtV/view?usp=drive
sdk
https://drive.google.com/file/d/1jzOx82Em8pZB-dLn6xucB16TX-p73lT9/view?usp=drive
sdk
https://drive.google.com/file/d/1gS8Xek1NlXfg53qx-PJjadnAUPDoGrKa/view?usp=drive
sdk
https://drive.google.com/file/d/1Y4IvVJOyKg9vOf2XDGeZgNbwaVMRgXfF/view?usp=drive
sdk
https://drive.google.com/file/d/14QivxKA9o5lwgIBAfCAZPBaZlgPh3WjH/view?usp=drive
sdk
Total execution time: 163.33632516860962 seconds
s@pop-os:~$ 
```

Solution 2*: Interacting with the DOM using Javascript*

Remember how I originally manipulated the FOMB site to change the options on the site to show all the links in the page at once on day 1? We can do this programmatically using `execute_script()`

First, we pull the last dropdown menu item by CSS Selector like so:

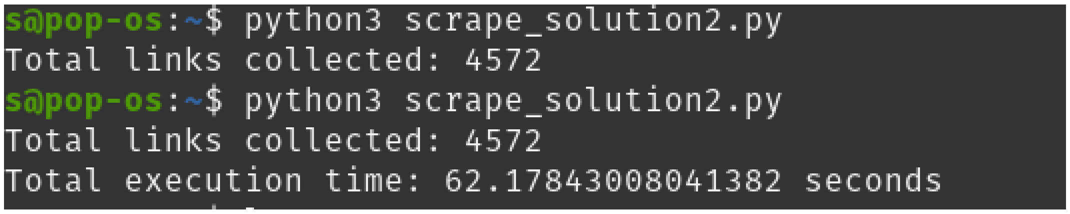
```
last_dropdown_item=browser.find_element(By.CSS_SELECTOR,
"#docsDataTable_length > label:nth-child(1) > select:nth-child(1) >
option:nth-child(4)")
```

Then, we can change the text on the display of that option and the actual value of items it will display like so:

```
browser.execute_script("arguments[0].innerText = '4572'",
last_dropdown_item)
browser.execute_script("arguments[0].value = '4572'", last_dropdown_item)
```

Obviously, this is still using the value I manually copied over from the webpage for the total number of entries. You can, however, put a larger number, like 5,000 in there to give you a buffer. It will display as many documents as there are, regardless. Using this method, I leave some flexibility in my script to handle more documents being loaded, but I don't run short. It also means I don't have to load the next page of results, wait, and keep clicking next.

With this approach, here is the run time:



```
s@pop-os:~$ python3 scrape_solution2.py
Total links collected: 4572
s@pop-os:~$ python3 scrape_solution2.py
Total links collected: 4572
Total execution time: 62.17843008041382 seconds
```

Solution 3: Introducing headless mode, list comprehension, and turning off loading of costly images

Wouldn't it be nice if we could use Selenium, which interacts with a web-browser to load elements on a page, but not actually have to load the graphical user interface for the web-browser? Well, you can. First, we will have to import Selenium's options.

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

Instantiate our options, and add an option

```
options = Options()
options.add_argument("--headless")
```

Then, we can run the browser with options.

```
browser = webdriver.Firefox(options=options)
```

Loading images on a webpage that we won't actually be scraping is also a costly operation. Let's make sure we don't load those either. We can add that as an option as well.

```
prefs = {"profile.managed_default_content_settings.images": 2}
options.set_preference("permissions.default.image", 2)
```

If you run this particular option and don't run it headless, you will see Selenium not load the images in the GUI. It also doesn't load them in the headless driver, which is good.

These options alone greatly reduce the run time. The last inefficiency to address is the costly for loop present in solution 1 and 2 in grab_links that looks like this:

```
def grab_links(browser):
      links = []

      rows = browser.find_elements(By.CSS_SELECTOR, "#docsDataTable tbody 
tr")
      for row in rows:
          link_element = row.find_element(By.TAG_NAME, 'a')
          href_value = link_element.get_attribute('href')
          links.append(href_value)

      return links
```

We can summarize the actions of the for loop in a list comprehension, so our method changes to:

```
def grab_links(browser):
      links = []
      link_elements = browser.find_elements(By.CSS_SELECTOR,
"#docsDataTable tbody tr a")
      links = [link.get_attribute('href') for link in link_elements]
      return links
```

Now, list comprehension isn't always faster than a for loop, but it generally does get to skip calling .append() and other list constructs, which do have costs.

Activity 2: Pop-Up Prompts

Selenium has a built in way to deal with pop up alerts once they load. First, include Alert in your imports:

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

Navigate to the alert testing page:

```
browser = webdriver.Firefox()
browser.get('https://the-internet.herokuapp.com/javascript_alerts')
```

Find the CSS Selector for the JS Alert button and click it.

```
js_alert = browser.find_element(By.CSS_SELECTOR, ".example >
ul:nth-child(3) > li:nth-child(1) > button:nth-child(1)")
```

```
js_alert.click()
```

Now, you can switch your browser context to the highest alert like so:

```
alert =  browser.switch_to.alert
```

and accept the alert like so:

```
alert.accept()
```

Try on your own for the third button "Click for JS Prompt" - make sure to wrrite some code to send keys to the prompt before you submit so that it appears on the page. Consult the documentation.

Activity 3: Frames

Definition: "In the context of a web browser, a frame is a part of a web page or browser window which displays content independent of its container, with the ability to load content independently. The HTML or media elements in a frame may come from a web site distinct from the site providing the enclosing content"

Most common example: iframes.

Example: An article I wrote contains an embedded iframe of a collection of DocumentCloud projects pertaining to the review.

How do we work with iframes?

```
switch_to
```

```
browser.switch_to.frame('frame_id')
```

In our example site, we can switch to the TinyMCE WYSIWYG Editor iframe by the iframe's ID.

<**iframe id="mce_0_ifr"** frameborder="0" allowtransparency="true" title="Rich Text Area" class="tox-edit-area__iframe"></iframe>

which is **mce_0_ifr**

```
frame_id = 'mce_0_ifr'
browser.switch_to.frame(frame_id)
```

Now, we want to actually type something into the rich text field in that iframe.

```
text_area = browser.find_element("tag name", "body")
text_area.click()
```

and of course we can leave a message:

```
text_area.send_keys("Hello World")
```

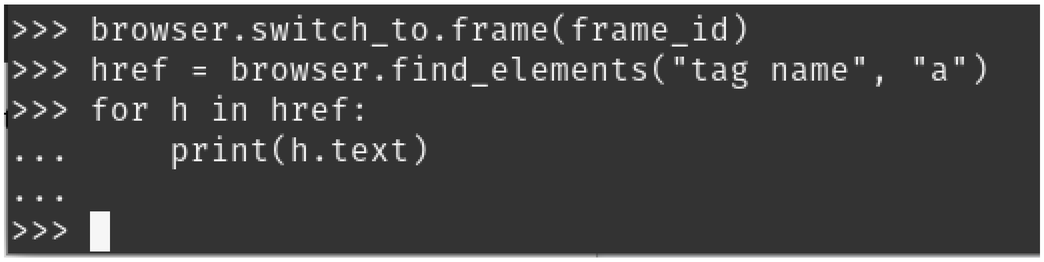
We can return to the main context using

```
browser.switch_to.default_content()
```

and confirm we have returned by checking out some elements.

```
hrefs = browser.find_elements("tag name", "a")
for href in hrefs:
      print(href.text)
```

Notice that if you run this exact code while in the iframe, you won't have any a tags returned.



```
>>> browser.switch_to.frame(frame_id)
>>> href = browser.find_elements("tag name", "a")
>>> for h in href:
...     print(h.text)
...
>>> 
```

You can do the same with window handling in Selenium using switch_to.

Activity 4: Infinite Scroll
Code
This is a special case of dynamically generated content. We need to use a while loop to deal with newly generated content and generally keep note of what index was last seen.

Activity 5: Memory & Scheduling

Thus far, all of our scrapers have been one time use, meaning they run once and they do not keep track of what it has seen. Wouldn't it be better if it did?

I will briefly cover the approach of the DocumentCloud Scraper Add-On.

For scheduling, you can always use the task scheduler in Windows or use cron to schedule things on Linux and Mac OS X.

Even better though, is to run it somewhere where it doesn't matter if your laptop is on or not. You could rent your own virtual private server(VPS) somewhere, be responsible for administering and securing it, and setting up the environment to run the code, or you could use GitHub.

GitHub is a major repository store and if you haven't used it before, it can have its intricacies to navigate. It has a lot of open source projects and an immense community of knowledgable people.

GitHub Actions are repositories with workflow files that tell them when and how to run the code in the repository and do specific tasks. They enable the use of runners(basically temporary virtual machines), which are temporary servers that do a specific task when told to do so, and then the server lets up resources for others who need them. There are limits to GitHub runners in terms of execution time and the number of concurrent jobs you can do with a free account, but it's hard to apprach those limits.

You can put some code into a repository on GitHub, create a file in **./github/workflows/** called **main.yml** and set up an environment (including server type, what Python versions you want to set up, and you can even schedule it to run however frequently you want).