

# How to Flexibly Scrape the Web<sup>\*</sup>

An Introduction to the Structure of Websites and Efficient Information Extraction

Lukas Puschnig

June 11, 2020

**Key Take-Aways and Skills:** *Understand the Basic Structure of most Websites; Control a Web Browser with your Code; Extract and Store Data for later Analysis;* **Technologies:** *Python incl. Selenium, HTML & CSS, XPath, Regular Expressions;* **Prerequisites:** *A very basic understanding of Python should suffice (a link to an introduction is provided below);*

## 1 Introduction

If nowadays you are looking for information, the internet is the place you will most likely find what you need. There are countless websites providing information on basically every topic you can imagine, and all that is just a mouse click away. But as much information as there is, much of it is unstructured. And even if there is some sort of structure, it may not be in the form you need it.


The most common way to represent data is by means of a relational database, with observations as rows, and variables as columns. A minimal example of such a dataset is shown in Table 1, where age and height of people are stored.

Table 1: Example of a Relational Database

	Variable 1: Age	Variable 2: Height	→
Observation 1	35	168	...
Observation 2	41	177	...
↓	⋮	⋮	⋮

Sometimes you are lucky and the data you need can already be downloaded in this form — in one of many potential file formats (popular ones are Excel-, CSV-, XML-Files, or the proprietary file formats of software packages, such as DTA-Files for Stata). Other times, you are a little less lucky, and you only find the table on the website, but not a download option. That’s usually not a huge problem, as you can just copy-paste it into your favorite spreadsheet software and save it to your liking. But even other times, the data is dispersed among several objects on a website, or even different pages, and the only way to bring the data into structure seems to do everything manually. But under

---

\* For the complete source codes and an online version of this guide, visit [puschnig.eu](https://puschnig.eu) .

certain circumstances, web scraping can stretch the boundary between what's convenient and inconvenient, at least to some extent.

The condition that needs to be met in order to use web scraping is simple: There needs to be some kind of structure, and this structure must be explained in logical terms. The raw data may not be in the relational database form which we want to obtain in the end, but if there are any rules and regularities it follows, we might exploit this, "scrape" the data and rearrange it to whatever form we need it to be.

Talking about structure, the remainder of this post is organized as follows. Section 2 gives a brief introduction to web design, focusing on the key aspects needed to apply web scraping. Section 3 sets up a basic scraping algorithm with which you can gather information from static websites. Section 4 expands the algorithm to dynamic websites. Section 5 concludes.

## 2 How Websites are Structured

Before this part begins, let me make a general recommendation. If you are interested in learning anything related to web design, or just need a compendium to look things up from time to time, I can very strongly recommend you to check out the website of [W3Schools](#). They have easy-to-follow tutorials on everything you might need, including [HTML](#), [CSS](#), and [XPath](#), and even [Python](#) (which can also be used to make a website!), all of the technologies relevant for this tutorial.

Most web browsers also offer *developer tools*, that allow you to not only look at the source code of the page, but also interactively sweep through the entire website, examine objects thoroughly, and even change them temporarily (although only on your machine, until you refresh the page — still a nice method to experiment and maybe pull a prank on a friend). Those tools are essential for web scraping and can make your life much easier, you should check them out, it's worthwhile. Just right-click somewhere on a website, and you should see the corresponding *inspect* command in the context menu (if not, check the settings of your browser, you might have to activate them beforehand).

### Hypertext Markup Language

Although there are several technologies with which you can set up a website, we will only focus on the crude building blocks they all have in common: The Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS). HTML is the language with which you set up the structure of the browser, ie. tell him what blocks and elements are where. CSS is used to change their appearance. But without further ado, let's jump straight into the first code block of this post. Although most websites consist of hundreds of lines of code (at least), it gives you a good idea nonetheless. In HTML, you have to think in

terms of blocks. Most blocks have to be opened and closed, which is done using `<...>` and `</...>` respectively, where you just replace the dots with the name of the block you want to generate. Whatever content you want to put into the block, you just place between the two tags.

```
<html>
<head>
<title>Title of Website</title>
</head>
<body>
<h1>Header 1</h1>
<h2>Header 2</h2>
<p>This is text.</p>
</body>
</html>
```

The above snippet should generate something like this:

# Header 1

## Header2

This is text.

You can think of this HTML document as a nested list, where the head and the body lie within the HTML block, the title lies within the head, and the two headers and the paragraph within the body:

- HTML
  - Head
    - \* Title
  - Body
    - \* Header 1
    - \* Header 2
    - \* Text

Each HTML document can be separated into two main parts, the head and the body. The head is where one sets the title of the website, loads style sheets and other scripts

on which the website draws on, and also provides some other, more technical information (for instance regarding search engine optimization, or SEO), whereas in the body, we find the blocks you can actually see on the website. Among the most common building blocks are headers (<h1></h1>, <h2></h2>, etc.), paragraphs (<p></p>), and blocks (<div></div>), which can take on various forms.

## Cascading Style Sheets

You can easily write your own little website with the code above — just copy it inside a text file (using the editor or a similar program, not Word!), save it, and change the file extension from .txt to .html. Your system should automatically recognize it as a website, and you can just double-click on it to open it with your browser. Quite neat, but also a little bit boring! Style Sheets allow to design the blocks to your liking. But unlike Word, where you select some text, and then click on some buttons until you're satisfied with the result (What-You-See-Is-What-You-Get), you have to write down in commands how specific blocks are supposed to look, and you can only see the result once you load the files into your browser. Let's see how this works by examining the below snippet:

```
<html>
<head>
<title>CSS Example</title>
<style>
h3 {
color: red;
}
.bluetextclass {
color: blue;
}
#greenid {
color: green;
}
</style>
</head>
<body>
<h3>Header 3</h3>
<div>
<p>This is text.</p>
<p style="color: yellow">Yellow Text (Inline CSS).</p>
<p class="bluetextclass">This is blue, by class.</p>
<p id="greenid">This green, by ID.</p>
</div>
</body>
</html>
```

In this new website, we have added color to four blocks, using four different techniques (and we have put them into a DIV, but ignore this for the time being):

### Header 3

This is text.

Yellow Text (Inline CSS).

This is blue, by class.

This green, by ID.

**Inline CSS.** The yellow text has been generated using inline CSS, that is, we have embedded the stylistic information directly into the tag by using `style="..."` and replacing the dots with our command (you can enter several commands if you separate them with a semicolon).

**CSS and Tags.** The header has been modified using CSS commands on an entire tag. This information is written in the head of the HTML document, either within `<style>` tags, or in a separate `.CSS`-file which is then loaded there instead. To do this, you simply write the name of the tag, `h3` in this case, and then line up the commands within curly brackets (here, we would separate them with semicolons as well).

**CSS and Classes.** Another way to assign styles to certain objects is to use classes, as for the blue paragraph above. You need to set them up with a preceding `.` in the style sheet (and you can give them almost any name you like), and then designate the blocks you want these rules to apply to by using `class="..."` within their tags. This is convenient because you can swiftly change the appearance of multiple elements by only changing the CSS!

**CSS and IDs.** In a fashion very similar to classes, you can also attribute styles using IDs, the green paragraph serves as an example for this. As for the syntactic differences, you use a preceding hashtag `#` in the CSS to declare a style for an ID, and you use `id="..."` in the tag of the object. However, there is a more grave difference between the two: IDs (and that's already in the name) are *unique*! You should not have two objects with the same ID in your website. A subtle fact that can save a web scraper quite some time and effort!

So far, so good. We have covered the basic structure of a website, now let's turn to the last puzzle piece, which helps us using this knowledge to get data!


## XPath

XPath is a language that was originally developed for the XML structure, but it also allows us to select certain elements of an HTML website by their characteristics. We will use this language later on to select the information we are interested in! To do so, you set up a piece of text, an *expression*, that defines which criteria an object must fulfill in order to be selected. Let's check out some basic expressions:

- `//h3` selects all objects that are headers of the third level. In a similar fashion, `//p` selects all paragraphs. The two slashes mean *"Select these objects, no matter where in the document they are"*.
- `//div/p` selects any paragraphs that are directly below, in the sense of one level below (this is also called child, or vice versa a parent) a DIV-object, as it is the case in the previous HTML snippet above. In contrast, if we had a paragraph within a form, within a div, it would not match the criteria, because the form would be in between the DIV and the paragraph. To ease this criterion, simply use two slashes between the tags: `//div//p`, this way, any paragraph that has a DIV somewhere above its hierarchy will be selected.
- `//div/*` selects any child object of any DIV in the website. The asterisk is a so-called *wildcard* and may stand for any given object.
- Square brackets allow to impose further restrictions on the elements of an expression.
  - `//div/p[1]` selects the first paragraph whose direct parent is a DIV.  
`//div/p[last()]` will select the last.
  - You can also impose restrictions on the class, the ID, or any other field within HTML tags in the square brackets, if you use the @-sign.  
`//div/p[@class='bluetextclass']` and `//div/p[@id='greenid']` select the blue and the green paragraph above, respectively.

These basic commands already get you quite far, and for most of your queries you won't need anything else. But you should nonetheless keep in mind that XPath is much more flexible, you can also have inequality restrictions, logical ands and ors, and complex statements like *"Select all A elements whose parent is a B element matching C restrictions"*. As long as there is any consistent structure in the document, there is a way to write that down in an expression!

A few practical tips before we move on:

- Since such statements can get pretty unclear pretty fast, it is a good idea to check whether they work or not. There are free tools available on the internet that allow you to see what you select! For instance check out this so-called [XPath Tester](#) .

- You don't always have to write your XPath manually. The developer tools of some web browsers allow you to directly copy the XPath of some elements (Firefox Version 77 supports this, for instance). They usually just create an expression from the outmost tag to the tag in question, which is very often not the most elegant solution, but it's a start!

We are now ready to move to Python in order to set up the first web scraping algorithm!

## 3 Scraping Static Websites

### Setting Up Python

If you have not already set up Python and are ready to rock'n'roll, I recommend installing the Anaconda toolkit. It provides you with everything you need and more. Check out [their website](#) for information on how to get started. Once installed, run Jupyter Notebook and create a new Python 3 Notebook (preferably in a separate folder for the project).

### Download Google Chrome and the Chromedriver

We will be using the Chromedriver application to control Google Chrome with Python. So install Google Chrome, if you haven't already, and download the Chrome Driver for your system at the [Chromium Website](#). Also, make sure the version of the Chromedriver matches your Google Chrome version! If after a while, your script stops working, chances are there was a Chrome update and you need to download the latest Chromedriver.

### Script 1: Static Web Scraping

For this tutorial, I have set up some simple HTML websites with some information to extract (I'm not a lawyer, and I cannot guarantee that scraping someone's website might not be legal all the times, but you are free to play around on these pages). You can find the first one here: [Static Web Scraping Page](#). It's just a nested list of elements, and we're gonna select and extract information we are interested in — in that case, we want the words that add up to the sentence, while ignoring the "ZZZ" which just represents irrelevant text. There may be multiple ways to select the desired elements, but the one below should do the trick:

- Select the first paragraph of the website.
- Select the element with ID "id1".
- Select all paragraphs that are children of any object with class "class1".

- Among the children of the DIV of class "class2", select the last paragraph.
- Finally, select the element with ID "id2".

Now that our roadmap is laid out, let's get started by importing the modules we will need. Modules provide us with functions that are not part of the basic Python distribution, but were written by other users. We will need the `webdriver` and `Options` part of the `selenium` module to control the browser, and the `re` module for string manipulations (more on that later on).

```
# Import Modules
# To control Chrome
from selenium import webdriver
# Further Options for Chromedriver
from selenium.webdriver.chrome.options import Options
# Regular Expressions
import re
```

If you get an error message, you might need to install some of these modules before you can use them, if you haven't done so already at an earlier stage. You can try the following code (just replace `Name` with the name of the module you wish to install), or check Google if it doesn't work for you:

```
import sys
!conda install --yes --prefix {sys.prefix} Name
```

That being done, let's set up the Chromedriver. The first two active lines in the code block below activate the *headless* option, ie. they tell the Chromedriver that we don't want to see the window. This is a good idea if you've already set up your script and everything is working — because you can economize on the resources of your system. But for the development stage, you might want to deactivate this by commenting these two lines by putting a `#` in front of them and deleting the `, options = chrome_options` from the next statement, where we load the Chromedriver. This function needs the path to the Chromedriver file you have downloaded earlier (make sure to un-zip it, if you've downloaded it as a ZIP-archive). Note that you can replace the driver on the left-hand side with any name you find useful — it's a variable where we store the driver's information in — and we will use that name whenever we need to control the browser. These are all the preliminary settings we had to make, the next step is already to load the website we want to scrape, to do this, we just call the driver and tell it to get the website, which we specify by its *URL* (ie., the link). If you have deactivated the *headless* option, you should see a Chrome window pop up at this point. Finally, we set up a text file to which we will "w"rite the extracted data.



```

# Set up Chromedriver

# This Section hides the Chrome Window
# If you want to see it, comment these two lines and delete the 'options'
# specification when opening the Driver
chrome_options = Options()
chrome_options.add_argument("--headless")

# Open Chromedriver
driver = webdriver.Chrome("/Path/chromedriver", options = chrome_options)

# Open Website
driver.get("http://puschnig.eu/WebScrapingPlayground/WS1_Static.html")

# Set up Result File
res = open("result_static.txt", "w")

```

This is where the actual scraping begins. As mentioned above, we will extract the first piece of information by selecting the first paragraph of the website, which can be accomplished by using the `find_element_by_xpath` function of our driver object, and supplying it with the XPath expression `"/p[1]"`. This selects the entire block: `<p>Paragraph 1.1.1: Web Scraping</p>`. Since the information we need is in-between the `<p></p>` tags, we extract everything between them by attaching the following function to the end of the command (we will see how to extract values of forms in the next section): `.get_attribute("innerHTML")`.

This leaves us with the string `"Paragraph 1.1.1: Web Scraping"`. To get rid of the preceding text, we will make use of *Regular Expressions* (RegEx), a concept that is similar to XPath, but instead of structured files, it is used to select parts of strings, ie. text. RegEx can be a true nightmare and even more messy than XPath, and it is not part of this guide, so we will just stick to a very simple application. If at any point you need further information on the topic, I can once more point to W3Schools and their [Python RegEx Tutorial](#). A useful tool to build or check your regular expression can be found on [Regex101](#).

Let's approach the last line of this code block from the inside to the outside. We call the `re.findall()` function and supply the RegEx and the string, in that order. `"\\: (.*)"` can be read as follows: Look for a colon (`"\\:"`, a backslash is needed to for symbols like the colon, and in Python, you have to make two; this is because of so-called escape sequences) that is followed by a whitespace (`" "`). Select the first group (`"()"`) that contains any number of random characters (`".*"`, a dot represents any character, and the asterisk represents any number of characters).

Finally, since the result is a list, we select the first element of this list by adding [0], and then we write it to the text file. Note the underscore \_ to which we assign the return value of the write function? It doesn't serve a special purpose. We just assign the return value somewhere in order to suppress output — you would see the number of characters written, if you were to remove the assignment.

```
# Gather Relevant Pieces of Text


# Get the first Paragraph of the Site via XPath, take Text within the Tags
raw1 = driver.find_element_by_xpath("//p[1]").get_attribute("innerHTML")
_ = res.write(re.findall("\\: (.*)", raw1)[0])
```

Similarly, we extract the information from the other objects. There are different commands for the different ways with which you can select objects from a website, the most important ones are the following:

`find_element(s)_by_id`: Identification by ID.

`find_element(s)_by_class`: Identification by (style sheet) class.

`find_element(s)_by_xpath`: Identification by XPath.

For the full list of commands, have a look the [Selenium Documentation](#) . Note that there are two versions of each command, one in the singular form, and one in the plural form. Their difference is precisely the number of elements they will return. If you use the singular form, the function will only return one element, even if there are several that would match the criteria. However, if you want to extract multiple elements, you can use the plural form, which instead returns a list of all elements. An example of this very important feature can be found down below, where we locate several paragraphs and then loop over the elements.

```
# Get the Element with ID "id1"
raw2 = driver.find_element_by_id("id1").get_attribute("innerHTML")
_ = res.write(re.findall("\\:(.*)", raw2)[0])

# Get all Paragraphs that are (direct) Children of any Element of Class
# "class1", via XPath
# Note the Plural!
raw3 = driver.find_elements_by_xpath("//*[@class='class1']/p")
for r3 in raw3: # Loop over Elements
_ = res.write(re.findall("\\:(.*)", r3.get_attribute("innerHTML"))[0])

# Get last Paragraphs of a DIV-Element with Class "class2", via XPath
raw4 = driver.find_element_by_xpath("//div[@class='class2']/p[last()]")
```

```
        .get_attribute("innerHTML")
_ = res.write(re.findall("\\:(.*)", raw4)[0])




# Get Element with ID "id2", this time via XPath, not knowing it's a Paragraph
raw5 = driver.find_element_by_xpath("//*[@id='id2']").get_attribute("innerHTML")
_ = res.write(re.findall("\\:(.*)", raw5)[0])
```

And that's that! Our first web scraping script is almost ready to run. Let's just close the result file and the Chromedriver and display the result, and the first script is all done.

```
# Close File, Quit Chrome
res.close()
driver.quit()

# Open File and Read Content
res = open("result_static.txt", "r")
print(res.read())
```

## Script 2: Static Multi-Page Web Scraping

The above script is already quite useful, but rarely is it the case that all the information you need can be found on one website. Luckily, it doesn't take much more effort to extend it to multiple websites. But let's see how that works by scraping information from three web pages that share a similar structure, [number 1 here](#) , [number 2 here](#) , and [number 3 here](#) .

For the preliminary code, we can re-use all of the previous script. We don't need any additional modules, we call the Chromedriver in the same way as before, and we set up a simple text file for the result. All that is new so far is that we define a list `sites`, which features the link to the three web pages we want to scrape.

Note that sometimes you don't even need to enter that list of links manually. You could take it to the next level and scrape the prerequisites you need for scraping itself! A gold mine for such things is Wikipedia, where lots of information is stored in a structured way, including links to other websites.

```
# Import Modules
# To control Chrome
from selenium import webdriver
# Further Options for Chromedriver
from selenium.webdriver.chrome.options import Options
# Regular Expressions
import re
```

```

# Set up Chromedriver

# This Section hides the Chrome Window
# If you want to see it, comment these two lines and delete the 'options'
# specification when opening the Driver
chrome_options = Options()
chrome_options.add_argument("--headless")

# Open Chromedriver
driver = webdriver.Chrome("/Path/chromedriver", options = chrome_options)

# Set up Result File
res = open("result_multipage.txt", "w")

# Set up Websites to Scrape
sites = ["http://puschnig.eu/WebScrapingPlayground/WS2.1_MultiPage.html",
"http://puschnig.eu/WebScrapingPlayground/WS2.2_MultiPage.html",
"http://puschnig.eu/WebScrapingPlayground/WS2.3_MultiPage.html"]

```

Extending the algorithm to cover several pages is as simple as wrapping a loop around the whole thing: For each site separately, open it, get the information, and store it in the file!

```

# Loop over Sites
for site in sites:
# Open Website
driver.get(site)

# Get the Element with ID "id1", and Write to File
raw = driver.find_element_by_id("id1").get_attribute("innerHTML")
_ = res.write(re.findall("\:(.*)", raw)[0])

```

And done! Closing the file and the driver, and reading the content of the former yields the same result as the first version of the script.

```

# Close File, Quit Chrome
res.close()
driver.quit()


# Open File and Read Content
res = open("result_multipage.txt", "r")
print(res.read())

```

## 4 Scraping Dynamic Websites

So far, we have learned how to scrape information from different elements of a page, and from different pages, which already gets us a long way. But we still face a major limitation: All the elements we have scraped so far have been static. By static, I mean that they don't change their content, be that over time or dependent on user input. In this chapter, we will see how to handle dynamic content — by scraping a form that produces output based on some user input. We will also move on to a more structured way of storing data using comma-separated values (CSV).

### Script 3: Dynamic Web Scraping

Let's have a look at the target site before we step into action. You can find it [here](#) . As you can see, it only features an input field that can take on values from 1 to 10, and a button, on which you must click to update the output. You will easily recognize the pattern by clicking through a couple of values, or by analyzing the JavaScript in the site. But that doesn't always work. It is not uncommon that the output is taken from a database, that only a server-side script can access (at least legally). Then you can only either manually go through all the possibilities — an arduous task, dependent on the number of combinations (after all, there could be several input fields) and the number of sites — or you can use web scraping to do the dirty work for you, which is most definitely the more elegant way to do it.

Let's begin by importing the modules and loading Chromedriver, as usual. For the dynamic scraping script, we need to additional ones: Keys to send key strokes to the website through Python, and csv to write CSV files.

```
# Import Modules
# To control Chrome
from selenium import webdriver
# Further Options for Chromedriver
from selenium.webdriver.chrome.options import Options
# Send Keys to Website, to Control Forms
from selenium.webdriver.common.keys import Keys
# Regular Expressions
import re
# Write CSV-Files
import csv

# Set up Chromedriver

# This Section hides the Chrome Window
# If you want to see it, comment these two lines and delete the 'options'
# specification when opening the Driver
```

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

# Open Chromedriver
driver = webdriver.Chrome("/Users/lukas/Desktop/chromedriver",
                          options = chrome_options)

# Open Website
driver.get("http://puschnig.eu/WebScrapingPlayground/WS3_Dynamic.html")
```

In order to bring some structure into the way we store data, we will use a so-called dictionary (dict). It is a data structure that has several named fields, and is enclosed by curly brackets. In our dict, we will have two fields, input and output, both numbers, and we are going to have one tuple for each observation we make (just like the example of a relational database at the beginning of this post!).

We want to repeat an action for several different values of a variable, and to do so we will loop over these values. We could just fill in the range of the loop manually, but a more elegant way to set it would be to read the minimum and the maximum values of an input of an HTML form in the code (think of the case where you want to scrape several forms that might differ in this regard). Just like class or id, the min and the max of the input are declared within the HTML tag, and can be obtained by finding the element and reading the attribute in question.

```
# Set up Result Format
result = {"input" : 0, "output" : 0}

# Get Minimum and Maximum Value of the Slider
min = int(driver.find_element_by_id("inputID").get_attribute("min"))
max = int(driver.find_element_by_id("inputID").get_attribute("max"))
```

Now let's get to the core of the new script. On the outmost level, we open a new CSV-file, to which we are going to write the results. To do this, we call the DictWriter function of the csv module and specify some settings for the CSV, namely the delimiter and the fieldnames.

The next step is to loop over the input values. The range(start, stop, increment) function provides a convenient way to do so — but don't forget to add + 1 at the upper bound, because the function omits this value by default. Next, we are going to locate the input field, which is no big deal since it features an ID field "inputID".

The real novelty lies within the next lines. For the first time, it's not only us taking data from the website (output), but also us delivering information to it (input)! We first

empty the input field, by selecting it and exerting the `clear()` function on it. then we will enter the current input value, as it is determined by the loop. We do so by sending the corresponding keys to the website, using the `send_keys` function. Lastly, we submit the changes by locating the button and clicking on it — with the `click()` function. Yes, it is really as easy as that! The remainder of the code within the loop is similar to the previous scripts. We locate the output fields, get the values, the only thing that's different now is that we store them in the dict, and then write them to the CSV-file with `writerow()`.

```
# Create and Open CSV-File
with open("result_dynamic.csv", mode='w') as csv_file:

# Set Field Names, Write Header
writer = csv.DictWriter(csv_file, delimiter = ";",
                        fieldnames = ["input", "output"])
writer.writeheader()

# Loop over Input Values
for i in range(min, max + 1, 1):

# Get Input Element
ip = driver.find_element_by_id("inputID")

# Write new Input
ip.clear()
ip.send_keys(str(i))

# Click Button to Submit new Input
btn = driver.find_element_by_id("buttonID")
btn.click()

# Read & Store Result
result["input"] = driver.find_element_by_id("inputID")
                        .get_attribute("value")
result["output"] = driver.find_element_by_id("outputID")
                        .get_attribute("innerHTML")

writer.writerow(result)
```

I suggest you turn off the headless option for this part, to see the magic at work. It's like an invisible user that surfs on the web, filling out forms and writing the dataset! Finally, let's have a look at the output of the script, which should yield a nice CSV with input and output — ready to be analyzed.

```
# Quit Chrome
driver.quit()

# Open File and Read Content
res = open("result_dynamic.csv", "r")
print(res.read())
```

## 5 Conclusion

Congratulations, you've made it to the end! I hope that you found this short guide helpful, and that it serves as a kickoff for you to pursue bigger projects. As long as there is any consistent pattern, there is a way to exploit it. Don't hesitate to get in touch if you have any questions or if you wish to provide feedback. Happy coding!



# Appendices

## Appendix A Static Scraping

```
1 #####
2 #
3 #   How to Flexibly Scrape the Web   #
4 #   Script 1: Static Web Scraping   #
5 #   Lukas Puschnig                 #
6 #   June 2020                      #
7 #                                   #
8 #####
9
10 # Import Modules
11 from selenium import webdriver # To control Chrome
12 from selenium.webdriver.chrome.options import Options # Further
    Options for Chromedriver
13 import re # Regular Expressions
14
15 # Set up Chromdriver
16
17 # This Section hides the Chrome Window
18 # If you want to see it, comment these two lines and delete the '
    options' specification when opening the Driver
19 chrome_options = Options()
20 chrome_options.add_argument("--headless")
21
22 # Open Chromedriver
23 driver = webdriver.Chrome("/Path/chromedriver", options =
    chrome_options)
24
25 # Open Website
26 driver.get("http://puschnig.eu/WebScrapingPlayground/WS1_Static.
    html")
27
28 # Set up Result File
29 res = open("result_static.txt", "w")
30
31 # Gather Relevant Pieces of Text
32
33 # Get the first Paragraph of the Site via XPath, take Text within
    the Tags
34 raw1 = driver.find_element_by_xpath("//p[1]").get_attribute("
    innerHTML")
```

```

35 _ = res.write(re.findall("\\: (.*)", raw1)[0])
36
37 # Get the Element with ID "id1"
38 raw2 = driver.find_element_by_id("id1").get_attribute("innerHTML")
39 _ = res.write(re.findall("\\: (.*)", raw2)[0])
40
41 # Get all Paragraphs that are (direct) Children of any Element of
    Class "class1", via XPath
42 # Note the Plural!
43 raw3 = driver.find_elements_by_xpath("//*[@class='class1']/p")
44 for r3 in raw3: # Loop over Elements
45     _ = res.write(re.findall("\\: (.*)", r3.get_attribute("innerHTML")
46     ))[0])
47 # Get last Paragraphs of a DIV-Element with Class "class2", via
    XPath
48 raw4 = driver.find_element_by_xpath("//div[@class='class2']/p[last
    ()]").get_attribute("innerHTML")
49 _ = res.write(re.findall("\\: (.*)", raw4)[0])
50
51 # Get Element with ID "id2", this time via XPath, not knowing it's
    a Paragraph
52 raw5 = driver.find_element_by_xpath("//*[@id='id2']").get_attribute
    ("innerHTML")
53 _ = res.write(re.findall("\\: (.*)", raw5)[0])
54
55 # Close File, Quit Chrome
56 res.close()
57 driver.quit()
58
59 # Open File and Read Content
60 res = open("result_static.txt", "r")
61 print(res.read())

```

## Appendix B Multi-Page Scraping

```

1 #####
2 # #
3 # How to Flexibly Scrape the Web #
4 # Script 2: Multi-Page Scraping #
5 # Lukas Puschnig #
6 # June 2020 #
7 # #

```

```

8 #####
9
10 # Import Modules
11 from selenium import webdriver # To control Chrome
12 from selenium.webdriver.chrome.options import Options # Further
    Options for Chromedriver
13 import re # Regular Expressions
14
15 # Set up Chromedriver
16
17 # This Section hides the Chrome Window
18 # If you want to see it, comment these two lines and delete the '
    options' specification when opening the Driver
19 chrome_options = Options()
20 chrome_options.add_argument("--headless")
21
22 # Open Chromedriver
23 driver = webdriver.Chrome("/Path/chromedriver", options =
    chrome_options)
24
25 # Set up Result File
26 res = open("result_multipage.txt", "w")
27
28 # Set up Websites to Scrape
29 sites = ["http://puschnig.eu/WebScrapingPlayground/WS2.1_MultiPage.
    html",
30         "http://puschnig.eu/WebScrapingPlayground/WS2.2_MultiPage.
    html",
31         "http://puschnig.eu/WebScrapingPlayground/WS2.3_MultiPage.
    html"]
32
33 # Loop over Sites
34 for site in sites:
35     # Open Website
36     driver.get(site)
37
38     # Get the Element with ID "id1", and Write to File
39     raw = driver.find_element_by_id("id1").get_attribute("innerHTML")
40     _ = res.write(re.findall("\\:(.*)", raw)[0])
41
42 # Close File, Quit Chrome
43 res.close()
44 driver.quit()
45

```

```
46 # Open File and Read Content
47 res = open("result_multipage.txt", "r")
48 print(res.read())
```

## Appendix C Dynamic Scraping

```
1 #####
2 #
3 #   How to Flexibly Scrape the Web   #
4 #   Script 3: Dynamic Web Scraping   #
5 #   Lukas Puschnig                   #
6 #   June 2020                         #
7 #                                     #
8 #####
9
10 # Import Modules
11 from selenium import webdriver # To control Chrome
12 from selenium.webdriver.chrome.options import Options # Further
    Options for Chromedriver
13 from selenium.webdriver.common.keys import Keys # Send Keys to
    Website, to Control Forms
14 import re # Regular Expressions
15 import csv # Write CSV-Files
16
17 # Set up Chromdriver
18
19 # This Section hides the Chrome Window
20 # If you want to see it, comment these two lines and delete the '
    options' specification when opening the Driver
21 chrome_options = Options()
22 chrome_options.add_argument("--headless")
23
24 # Open Chromedriver
25 driver = webdriver.Chrome("/Path/chromedriver", options =
    chrome_options)
26
27 # Open Website
28 driver.get("http://puschnig.eu/WebScrapingPlayground/WS3_Dynamic.
    html")
29
30 # Set up Result Format
31 result = {"input" : 0, "output" : 0}
32
```

```

33 # Get Minimum and Maximum Value of the Slider
34 min = int(driver.find_element_by_id("inputID").get_attribute("min")
    )
35 max = int(driver.find_element_by_id("inputID").get_attribute("max")
    )
36
37 # Create and Open CSV-File
38 with open("result_dynamic.csv", mode='w') as csv_file:
39
40     # Set Field Names, Write Header
41     writer = csv.DictWriter(csv_file, delimiter = ";", fieldnames =
        ["input", "output"])
42     writer.writeheader()
43
44     # Loop over Input Values
45     for i in range(min, max + 1, 1):
46
47         # Get Input Element
48         ip = driver.find_element_by_id("inputID")
49
50         # Write new Input
51         ip.clear()
52         ip.send_keys(str(i))
53
54         # Click Button to Submit new Input
55         btn = driver.find_element_by_id("buttonID")
56         btn.click()
57
58         # Read & Store Result
59         result["input"] = driver.find_element_by_id("inputID").
            get_attribute("value")
60         result["output"] = driver.find_element_by_id("outputID").
            get_attribute("innerHTML")
61
62         writer.writerow(result)
63
64 # Quit Chrome
65 driver.quit()
66
67 # Open File and Read Content
68 res = open("result_dynamic.csv", "r")
69 print(res.read())

```