

A Beginning To Advanced Web Scraping


Why this work? Sometimes, to automate data collection from websites, we must actually operate those web pages on that site to reach a page that has the data we need. These types of data collections cannot be done using simple Python `requests`. However, learning to automate web pages is a great skill for both testing web site operations AND for collecting data that could not be collected otherwise. This is an initial tutorial on how to setup advanced web scraping with selenium through Python.

All of the code shown in this `ReadMe.md` markdown file is in Thom's [DagsHub Web Scraping Repo](#), and this file is also stored as a PDF in that repo. I will be updating the code AND this document with additional and refactored tools and other examples as frequently as possible.




This initial work is for Linux, but I do have routines that work just as well on Windows, and I will share them soon. Mac users, I think the linux routines will be a good starting point for you, and I don't own a Mac, so you are on your own - sorry.

What Will NOT Work More And More Often

Consider that we are in a fantasy hockey league, and we want to automate capturing skater statistics from <https://www.nhl.com/stats/skaters>. Why? The data on our skater pool changes frequently, and we need updated data to make the best decisions about which skaters to place on our team. We would even hope to use some cool math machines from machine learning techniques or more to create the best fantasy team so that our team can win in this fantasy league. But we need to scrape the data from one of the pages on the NHL Skaters page first. Oh! There are many pages of these skaters. Hmmm. Let's start with at least one page first. Here's what the NHL Skaters site looks like in dark mode when we go there.



[Scores](#)
[News](#)
[Video](#)
[Standings](#)
[Stats](#)
[Schedule](#)
[Players](#)
[App](#)
[Tickets](#)
[Fantasy](#)
[Shop](#)
[Teams](#)
[Heritage Classic](#)
[Stream Games](#)

Statistics

[HOME](#)
[SKATERS](#)
[GOALIES](#)
[TEAMS](#)
[GLOSSARY](#)
[FANTASY](#)
[RECORDS.NHL.COM](#)

[BY SEASON](#)
[BY GAME](#)
[ALL-TIME](#)

[Legend](#)
[Print](#)

SEASONS

2021-22

-

2021-22

☒ SUM RESULTS

GAME TYPE

Regular Season

FRANCHISE

All Franchises

POSITION

All Skaters

REPORT


Summary

Get Stats

More Filters

Clear Filters

Summary


[Export](#)

#	Player	Season	Team	S/C	Pos	GP	G	A	P	+/-	PIM	P/GP	EVG	EVP	PPG	PPP	SHG	SHP	OTG	GWG	S	S%	TOI/GP	FOW%
1	Connor McDavid	2021-22	EDM	L	C	66	36	62	98	+16	39	1.48	28	59	8	38	0	1	2	6	254	14.2	22:16	54.1
2	Leon Draisaitl	2021-22	EDM	L	C	67	48	48	96	+18	40	1.43	27	59	20	36	1	1	1	10	227	21.2	22:48	53.9
3	Johnny Gaudreau	2021-22	CGY	L	L	65	30	60	90	+49	22	1.38	25	70	5	20	0	0	2	8	211	14.2	18:19	100.0
4	Jonathan Huberdeau	2021-22	FLA	L	L	65	21	69	90	+25	42	1.38	15	57	4	29	2	4	0	3	174	12.1	19:16	32.7

Those skilled in such arts of web scraping of data already know some cool tricks! Assuming you are in Python3, we can use the code below on some sites.

```
import requests
import pandas as pd

page_url = "https://www.nhl.com/stats/skaters"
page = requests.get(page_url)

# This didn't work!
# dfs = pd.read_html(page.text)
# print(dfs[0])

print("Connor McDavid" in page.text)
```

The Pandas read_html class is VERY smart. It will look through the text of page contents captured from the page = requests.get(page_ur1) line and then give us all the tables found in page.text as a list of Pandas DataFrames.

Well, as you likely guessed, due to foreshadowing of the language in the previous text, and the commented out lines, this did NOT work. Bummer! Note what we tried. Also, just to further investigate the extent of the issue, we checked to see if the first players name from the table appeared anywhere in page.text , and that last print("Connor McDavid" in page.text) did return False .

Why is this happening? If you do an inspection on this page (right click on the NHL Skaters page, and then click Inspect in the context sensitive menu), you will see that the html code for this page loads a ton of

scripts! Basically, the table that we see won't get loaded when we try to capture it using a `requests.get(page_url)` type command.

Is there hope? Yes, but it becomes a bit more complicated. Let's take that increased complication and break it into small and manageable steps. We hope that this document makes it much easier for you to get started than if you had to start learning this from the point where we did.

What You Need More And More Often

We need Selenium. Seleni-what? Haha! The language used to automate the testing and (as it turns out) the collection of data from highly dynamic websites. We basically use it to help us programmatically operate web browsers like we would, but with a program. Now, if you are a spoiled Python users like us (yes, we went there, because after so many other languages, we feel completely spoiled using Python), you want to be able to use Selenium through Python, and that is how we will use Selenium - thru Python.

Setting Linux Up For Selenium Thru Python With Chrome

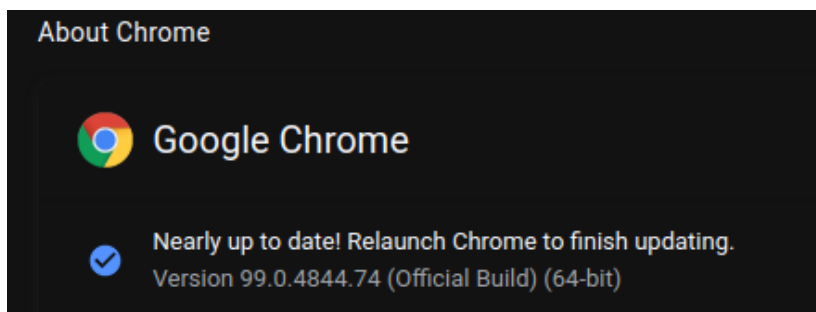
(Windows OS And Other Browsers Coming Soon)

First, we will need a driver that allows us to control an instance of the Chrome browser from our local host using selenium. Thankfully, the major browser companies create such drivers knowing that dynamic web pages need to be tested locally before they are released into the wild wild web. This driver for Chrome is simply called the `chromedriver`. As we grow this repo, we will add setup guides for other browsers.

The chromedriver you need for your system is dependent on the following.

1. Your OS and OS's bit size.
2. Your version of chrome.

To find your Chrome version, go to your chrome settings, and find your current version of Chrome as shown in the image below.



Once you know that, we can go to <https://chromedriver.chromium.org/downloads> to find the version of chromedriver that will match our chrome version as shown in the image below.



Downloads

Current Releases







- If you are using Chrome version 100, please download [ChromeDriver 100.0.4896.20](#)
- If you are using Chrome version 99, please download [ChromeDriver 99.0.4844.51](#)
- If you are using Chrome version 98, please download [ChromeDriver 98.0.4758.102](#)
- For older version of Chrome, please see below for the version of ChromeDriver that supports it.

If you are using Chrome from Dev or Canary channel, please following instructions on the [ChromeDriver Canary](#) page.

For more information on selecting the right version of ChromeDriver, please see the [Version Selection](#) page.

For Thom, currently, I'd click on the version 99 link unless I decided to upgrade to version 100 of Chrome. How to automate capturing a new chromedriver 🤔? We want to do this, but let's get these basics learned first.

Index of /99.0.4844.51/

	<u>Name</u>	Last modified	Size	ETag
	Parent Directory	-	-	-
	chromedriver linux64.zip	2022-03-02 08:47:18	6.68MB	59450a2dc9281e3f54bce151ee702283
	chromedriver mac64.zip	2022-03-02 08:47:20	8.00MB	5bc84c9fbf3242117788b35595aa9082
	chromedriver mac64 m1.zip	2022-03-02 08:47:22	7.30MB	c3dc459cefdd8e93920f5ed3da0245bd
	chromedriver win32.zip	2022-03-02 08:47:25	6.00MB	c42faa8f2d89fe5d3f3ba20658787144
	notes.txt	2022-03-02 08:47:30	0.00MB	ff87250a42b6025118bf261e3b9a338a

Location Of chromedriver On Linux

Once we have our chromedriver, OR an upgraded version of it, we put that chromedriver into the `/usr/bin/` directory, or replace the current version. In the end, your chromedriver goes here as shown. You can put it elsewhere, but you will need to create a virtual link to it.

```
/usr/bin/chromedriver
```

A Dedicated ChromeProfile For Our Web Scraper

Last, we want to create a copy of our current Chrome profile and put the profile into a dedicated directory for web scraping work. Thom even creates a separate profile for each web scraper. This may be overkill, and I may change this in the future, but right now, I just do it to keep things separated. So, find your current Chrome Profile, and copy it to a new directory that you can point to with our new routines later.

Operating Web Browser Ops With Selenium Through Python

Creating And Covering Convenience Functions

Rather than start from scratch, we want to show you some tools that we've developed that you can use over and over (trying to stay DRY - don't repeat yourself, or in our case - ourselves). As we always encourage, take our code as a starting point, and PLEASE refactor it to your specific needs and/or liking. If this helps you to make a great start, we'd appreciate you mentioning us in your code comments or any posts you write about the same with a link back to this repo. But if you don't, we won't take any action against you either - all of this is free to use.

Let's start in the `Basic_Tools.py` file. First, the imports.

```
import time
import subprocess
import selenium
import os
import pprint

from selenium.webdriver.chrome.options import Options

pp = pprint.PrettyPrinter(indent=2)
```

Next, the `send_Bot_to_sleep` function. What?!?! We want to put the Bot to sleep already? Well, this is just a convenience function. While developing a new web scraper that is specific to some new web scraping on a specific site, we often times need an automated way to simply `kill` the process that was running our web instance. In summary, all this does is collect output from the specific `netstat` command. If our `host_port` appears in a line, it collects the information needed to `kill` the id for our Bot (sounds so violent, but then the process isn't sentient, so we should be OK with this).

```
def send_Bot_to_sleep(host_port):
    print("Seeing if Bot is awake.")
    ops = subprocess.run("netstat -ano", shell=True, capture_output=True)
    time.sleep(4)
    print("Parsing terminal capture")
    bot_process = [r for r in ops.stdout.decode("utf-8").split('\r\n')
                    if host_port in r]

    if bot_process:
        print('\n', "\t Bot was awake. Trying to put it to sleep")
        bot_pid = bot_process[0].split()[-1]
        cs = f'taskkill /F /PID {bot_pid}'
        ops = subprocess.run(cs, shell=True, capture_output=True)
        time.sleep(4)

        return ops.stdout.decode("utf-8")

    else:
        return "Bot was already asleep"
```

Depending on where we are at in our web scraping development, we may or may not call the above function. As always, play with this. Take it line by line and study what each line has accomplished. You'll be glad that you did.

This next function, `bot_is_running`, similar in nature to the above, simply returns a boolean as to whether our Bot is running or not.

```
def bot_is_running(host_port):
    ops = subprocess.run("netstat -an", shell=True, capture_output=True)
    time.sleep(4)
    if [r for r in ops.stdout.decode("utf-8").split('\r\n') if host_port in r]:
        print(f'Bot is awake.')
        return True
    else:
        print(f'Waking Bot up.')
        return False
```

And finally, at least for this intro to automatic web ops, we need a function that literally opens a browser, so that we can start to operate it and collect data from it. That function is named `prepare_Bot`. Note that `prepare_Bot` calls `bot_is_running` so that we don't needlessly try to relaunch another browser, which the code in the `if` block will do if it's ran. This really helps with developing our specific web scraper routines. We can start at a certain place in our website operations and continue from there to test one new little thing at a time by using an existing running instance of a selenium controlled browser rather than operate through all the operations up to the point of our current development. This saves a TON of development time!

For the `cmd_str`, we build it up to call a linux terminal, keep it open, and run the command

```
google-chrome-stable --remote-debugging-port=8223 --user-data-dir ="home/thom/NHS/ChromeProfile"
```

, but we escape our " s where needed to build the string and send it properly to the system terminal we

opened. Take note of how we use our dedicated chrome profile in the `cmd_str` -
`--user-data-dir ="/home/thom/NHS/ChromeProfile"` .

The code below the `if` block will give us the `driver` to connect to the instance that we started, if it wasn't yet running, or connect to an existing instance. The line,
`options.add_experimental_option("debuggerAddress", host_port)` is key to what we are doing here. It makes reusing instances for our specific test profile possible.

```
def prepare_Bot(host_port):
    if not bot_is_running(host_port):
        cmd_str = "gnome-terminal -e 'bash -c \"google-chrome-stable \"
        cmd_str += "--remote-debugging-port=8223 --user-data-dir"
        cmd_str += "=\"/home/thom/NHS/ChromeProfile\"; exec bash\"'"
        os.system(cmd_str)
        time.sleep(3)

    options = Options()
    options.headless = False
    options.add_experimental_option("debuggerAddress", host_port)
    driver = selenium.webdriver.Chrome(options=options)
    time.sleep(3)

    return driver
```

Using Convenience Functions And More To Operate Our Website

Let's now explain the lines in the `Basic_Dev_1.py` script that is used to control our browser. First, `import` needed modules. Obviously, for your Python environment, `pip install <needed_packages>` also. Please note that the `Basic_Tools` module is the Python script `Basic_Tools.py` that we previously covered.

```
import Basic_Tools as bt
import numpy as np
import pandas as pd
import time
```

The next block of code is tasked with obtaining a `driver` that we can use to perform web operations on our website's pages.

```
put_Bot_down = False
host_port = "127.0.0.1:8223"

if put_Bot_down:
    bt.send_Bot_to_sleep(host_port)

driver = bt.prepare_Bot(host_port)
```

This next block is KEY to maintaining our sanity during the development of a new web scraper. If we are already where we want to go, don't reload to go there! And the `if` block could be changed to allow for many more page titles. This allows us to pull data and operate the page from where we were last doing operations. This is a BIG time saver and sanity maintainer as we refactor and add more and more functionality.

```
page_title = "NHL Stats | NHL.com"
if page_title not in driver.title:
    print(f"Going To {page_title}.\n")
    driver.get("https://www.nhl.com/stats/skaters")
    time.sleep(4)
else:
    print(f"Already on {page_title}\n")
```

We encourage you to comment out the code below and watch your terminal and test browser open. Pretty cool - right? Well, we are only just starting. Let the coolness continue. If the previous code has not yet seemed super cool, prepare your inner geek to be thrilled.

In the next block, we first use the `driver` to obtain the page source. Then we use the `driver` to `find_elements_by_class_name` that have the class name of `"rt-header-cell"` so that we can find the table header names / column names. How did we know to use `"rt-header-cell"` ?

Once your inspection window is open for the NHL Skaters site, you can right click on various elements in the web page again, and the highlighter in the html code window will highlight the code that controls that. We kept doing this until we saw the table header code highlighted for this first page of Skater Stats. Then we double clicked on the class name, which highlighted it, and copied it, and then pasted that into our find elements statement.

Next, we cycle through `col_elements` elements to collect the text to capture the text from each element of `col_elements` .

Next, we repeat a similar process for the `"rt-tbody"` class element(s). Now usually, we can take magical steps with Pandas' `read_html` method to capture this table data by giving it some form of the `page_source` data. On our NHL page, this did not work - DARN! So, we get creative!

```
page_source = driver.page_source

col_elements = driver.find_elements_by_class_name("rt-header-cell")
col_names = [e.text for e in col_elements]

table_sel = driver.find_elements_by_class_name("rt-tbody")[0]
table_list = np.array(table_sel.text.split('\n'))
table_array = np.reshape(table_list, (50, 25))

df = pd.DataFrame(data=table_array, columns=col_names).set_index('i')
print(df)
```


Creative step 1 is `table_list = np.array(table_sel.text.split('\n'))` , where we take what we got from `table_sel`, which was a long string of table data, split it into an array delimited by new line characters, and changed it into a numpy array. Then, we reshape that array to be a table (or a matrix). Finally, we put the array data into a Pandas DataFrame, and set the column names with the column names that we found previously, and then finally let the index values of the DataFrame be the ones from the web page's skater table data.

The next code block is just showing an alternative way to build the dataframe. We prefer the method above, but we wanted you to be aware of this other option too.

```
'''Another Method that is commented out in the actual file'''
table_data = {}
for i in range(25):
    table_data[col_names[i]] = table_array[:, i].tolist()

df = pd.DataFrame(data=table_data).set_index('i')
```

Summary And Next Steps

We couldn't get to our table data in a web page unless we operated that web page, and we wanted to automate these actions so that we could repeat this data gathering at will. So we used Selenium through Python to operate the web page and collect our data.

What do we still need to do? We need to cycle through the other skater pages and collect the table data from each page. Then we can concatenate these tables into one dataframe. Why? So that we can begin exploring the data and create the best fantasy hockey team possible. Well, hopefully we will use these new found skills in other ways too!