Python selenium download pdf chrome

Web scraping has been used to extract data from websites almost from the time the World Wide Web was born. In the early days, scraping was mostly done on static pages - those with known elements, tags and data. Lately, however, advanced technology in web development has made the task a little more difficult. In this article we explore how we could go about scraping data in case new technologies and other factors prevent standard scraping. Traditional scraping data As most websites produce pages designed for human readability rather than automated reading, web scraping basically consisted of software digesting web page markup data (think right click, View Source) and then discovering static patterns in that data that would allow the program to read various pieces of information and store it in a file or database. If the report data can be found, the data will often be accessed by transmitting either variable forms or parameters from the URL. For example: Python has become one of the most popular web scraping languages partly due to the various web libraries that have been created for it. One popular library, Beautiful Soup, is designed to other pull data from HTML and XML files, allowing you to search, navigate, and modify tags (i.e. the parsing tree). Browser-based scraping recently, I had a scraping project that seemed pretty simple and I was completely ready to use traditional scraping to handle it. But as I got further into it, I found obstacles that couldn't be overcome with traditional methods. Three main problems prevented me from my standard scraping methods: Certificate.There was a certificate needed to install to access the part of the website where the data was. When accessing the original page, a hint appeared asking me to choose a proper certificate of those installed on my computer and click OK. Iframes.The site used iframes, which messed up my usual scraping. Yes, I could try to find all the iframe URLs and then build a sitemap, but it seemed like it might get cumbersome. JavaScript.Data was available after filling out a form with options (e.g. customer ID, date range, etc.). Normally, I would bypass the form and simply transfer the variable forms (via URL or as hidden form variables) to the result page and see the results. But in this case, the form contained JavaScript, which prevented me from accessing the form variables in the usual way. So, I decided to abandon my traditional methods and look at a possible browser tool based on scraping. This will work differently than usual -- instead of directly onto the page, loading the parsing tree, and pulling out data, I would instead act like a person and use a browser to get to the page I need and then scratch the data - thus bypassing the need to deal with the barriers mentioned. Selenamius Selenjumis as an open source testing platform for web applications, allowing professionals to perform automated tests, playback, and implement remote control functions (allowing many browser instances to test load and multiple types of browsers). In my case, it seemed like it might be helpful. My way to the language is for the web scraping Python, since it has a well-integrated library that can usually handle all the necessary functionality. And of course, aSelenium libraryexists for Python. This would allow me an instant browser - Chrome, Firefox, IE, etc. - then pretend I used the browser myself to access the data I was looking for. And if I didn't want the browser to really appear, I could create a browser in a ready mode, making it invisible to any user. Set up the project To start the experiments I needed to set up my project and get everything I needed. I used the Windows 10 machine and made sure I had a relatively updated version of Python (it was v. 3.7.3). I created an empty Python script and then downloaded the libraries that I thought might be required using PIP (Python package installer) if I haven't had a downloaded library yet. These are the main libraries that I started with: I also added some call options to the script (using the argparse library) so I could play with different data sets, triggering a script from a command line with different options. These include customer ID, month to year, and month to year. Problem 1 - Certificate First Choice I needed to do which browser I was going to tell Selen to use. As I usually use Chrome, and it's built on an open source Chromium project (also used by Edge, Opera, and Amazon Silk browsers), I realized that I wanted to try that first. I was able to run Chrome in the script by adding the library components I needed and then issuing a few simple commands: Since I didn't run the browser headless, the browser actually appeared and I could see what it was doing. He immediately asked me to choose a certificate (which I had installed earlier). The first problem to be solved was the certificate. How to choose the right one and take it in order to get on the website? In my first script test, I received this query: It wasn't good. I didn't want to manually press the OK button every time I run my script. As it turned out, I was able to find a workaround for

this - without programming. While I was hoping that Chrome would be able to transfer the name of the certificate to the startup, this feature does not exist. However, Chrome has the ability to auto-elect a certificate if a certain record exists in your Windows registry. You can install it, choose the first certificate he sees, or be more specific. Since I only had one certificate downloaded, I used a generic format. So with this set when I said selenium to start running the launch and the certificate quickly came up, Chrome will AutoSelect certificate and continue. Problem 2 - Iframes OK, so now I was on the site and a form appeared that prompted me to enter the customer ID and report range date. After studying the form in the developer's tools (F12), I noticed that the form was presented in iframe. So before I could start filling out the form, I needed to switch to a proper iframe where the form existed. To do this, I referred to the Selenium switch function as so: Ok, so now in the right frame, I was able to identify the components, fill the customer identification field, and choose the drop date: Problem 3 - JavaScript The only thing left on the form was to press the Find button, so it would start searching. This was a little more complicated because the Find button seemed to be controlled by JavaScript and was not a regular Send button. After looking at it in the developer's tools, I found an image of the button and was able to get XPath of it, right click. Then, armed with this information, I found an item on the page and then clicked on it. And the vual, the form was presented and the data appeared! Now I could just scrape together all the data on the result page and save it as needed. Or can I? Getting data First, I had to handle a case where the search found nothing. It was pretty simple. It will display a message in the form of a search without leaving it, something like no records found. I was just looking for this line and stopped right there if I found it. But if the results did come, the data was presented in divs with a plus sign () to open the transaction and show all its details. The open transaction showed a minus sign (-) that will close the div when pressed. Clicking the plus sign will call the URL to open your div and close any open. So you had to find any plus signs on the page, collect the URL next to each one, and then cycle through each to get all the data for each transaction. The above-stated field code that I received was the type of transaction and status, and then added to the account to determine how many transactions complied with the rules. However, I could get other fields in the transaction details such as date and time, subtype, etc. for this project account was returned back to the call application. However, it and other scraped data could be stored in a flat file or database as well. Additional possible obstacles and solutions to obstacles and solutions to a host of other obstacles can be presented when scraping modern websites with your own browser instance, but most of them can be solved. Here are a few: Trying to find something before it appears While browsing yourself, how often you find that you're waiting for a page to sometimes for many seconds? Well, the same thing can happen while navigating navigation You're looking for a class or another item - and it's not there! Fortunately, Selenium has the ability to wait until it sees a certain item, and can time out if an item doesn't show up like this: Getting through CaptchaSome sites use Captcha or similar to prevent unwanted robots (which they might consider you). This can put a damper on the web scraping and slow it down. For simple clues (like what is 2 and 3?), they can usually be read and found out easily. However, for more advanced barriers, there are libraries that can help try to crack it. Some examples2Captcha, Death Captcha, andBypass Captcha. Website structural changesWebsites are designed to change - and they often do. That's why when writing a scraping script, it's best to keep that in mind. You want to think about which methods you'll use to find data and which not to use. Let's look at partial comparison methods rather than trying to match an entire phrase. For example, a website can change a message with no records found on No Records located - but if your match is on No Records, you should be fine. Also, consider whether to match XPATH, ID, name, link text, tag or class name, or CSS selector - and which is least likely to change. It was a brief demonstration that almost any website can be scraped, no matter what technology is used or what complexity is involved. Basically, if you can browse the site yourself, you can usually be scratched. Now, as a caveat, this does not mean that every websiteshouldbe is scraped. Some have legal restrictions in place, and there have been numerous court cases of the legality of scraping some sites. On the other hand, some sites welcome and encourage data from their website, and in some cases provide APIs to make the task easier. Either way, it's best to check with the terms before you start any project. But if you go ahead, rest assured that you can get the job done. Recommended Resources for Integrated Web Scraper: Toptal Engineering Blog is a hub for in-depth development tutorials and new technological ads created by professional software engineers on the Toptal network. You can read the original piece written by Neil Barnett here. Follow Toptal Engineering on Twitter and LinkedIn. Linkedin. python selenium download chromedriver. how to set download location in chrome using selenium python. selenium download file python chrome. selenium chromedriver set download folder python. selenium download pdf chrome python. python selenium chrome headless download file. python selenium chrome wait for download to finish. python selenium chrome download multiple files