Web scraping the Travis County public records database

A painful memoir

These are my experiences from my recent web scraping endeavors. I'm pretty sure I have fallen in every trap and have made every mistake there is, so, if you ever find yourself in similar paths, you may find something useful in here! If you just read it now, without having experienced what true webserver rejection means, it will probably make little sense.

There are certainly no recipes for web scraping: every website is unique, so there is no one-size-fits-all. Sometimes it feels that there is not even a one-size-fits-one, because the behavior of the site may change with time and the server's load, possibly even reacting to your own web scraping. Here are some guidelines:

DON'T JUDGE A SITE BY ITS LOOKS
This innocent ugly-looking site (http://tccsearch.org/RealEstate/SearchEntry.aspx) seemed like an easy target. But I did not notice that it was built by Thomson-Reuters and they are actually selling a commercial product for public records storage, so I suppose they tried to make it hard to scrape. Either this or their web developers write code as messy as I do.

FIRST THINGS FIRST
- Study the website html using the Chrome (or other) browser DevTools. Right-click → Inspect all the links you need. Keep in mind that if there are embedded frames, what you see in the right panel is not what you will get if you save the html. The html on the right is an amalgam of the code of all levels, while a simple "save" will keep only the top frame. You will need to navigate to the individual frames where the data is.

- If the images/documents/pages you want to download have their own unique page, go buy a lottery ticket before your luck runs out. Figure out what differentiates each URL and how it is associated with the final object you need, generate the URLs and start downloading.
Note that even if you are not able to see this differentiating value in the URL, you may be able to see it passed behind the scenes. Check the network tab in the DevTools, where you can record your browser's incoming and outgoing traffic.

-Unfortunately, most modern websites use javascript, ajax and other technologies I don't understand, so you have to earn your scrapings! This is where selenium comes to the rescue. Selenium is a way to simulate the way a human would navigate a page. Since it has to go through all the clicking and typing that a human would do, it is slow, but it may be the only way.

AWS & EC2s
- Unless you are planning to have your laptop running continuously for a week or a month, you will want to run this on AWS. Parallelize your scraping to different EC2 instances, which means different IP addresses. Customize one instance exactly the way you want it (with all the python tools installed etc) and then create an AMI (*Amazon Machine Image)*. From this image you can launch as many EC2 instances as you like, and they will be ready for use out of the box. If you see that your IP is blocked, simply restart the instance and it will be assigned a different IP address.

EC2 & SELENIUM

- There is a minor catch if you use selenium on AWS: you cannot use a normal browser; instead you need to use what is called a "headless" browser which does not have a GUI. In my case the transition was not hard: I wrote the core code using the chromedriver so I could visually check what was going on and when it was working, just changed to `webdriver.PhantomJS()` which is one of the popular headless browsers.

DON'T BE RUDE

- Web scraping a large site which accommodates thousands of requests per second guarantees that you will not overload their servers. On the flip side, they have probably automated their web scraping defenses.
- A small site may have less sophisticated defenses, but it will probably cave in under your requests if you overdo it. In my case the server is slow even for a simple browser based request. I gradually increased the scraping instances to five and it was still behaving OK. I raised it to twenty and they started throttling their server, fortunately until the end of the workweek. So I learned to be gentler with my requests.
- You can see how the site owner would like the visiting robots to behave by reading the mainURL/robots.txt file. In my case it says:

```
User-agent: *
Disallow: /
```

which means that no robot should visit any pages. That makes me feel like a bad robot :[

DON'T END UP IN JAIL

- I used to think that web scraping is not such a big deal; after all you are downloading public data. But others don't share my thoughts:
https://techcrunch.com/2016/08/15/linkedin-sues-scrapers/ (and several similar)
or even worse :(
https://en.wikipedia.org/wiki/Aaron_Swartz
So it's not a bad idea to check if there are any legal agreements you are unwillingly entering when you access a website. In that case you can go the Tor way, which is certainly safe but very slow. If you have to use a username and password there are certainly some binding agreements.

RECORD YOUR DOWNLOADING PROGRESS…

One fundamental problem is to know where you were left the last time your scraper crashed. My naive initial approach was to use the terminal output (duh…) or log files, because I thought that building an AWS database would be complicated. After a few times when the scraper crashed in an unexpected way and there was no record left, I did it the proper way:

… AND CREATE A DATABASE ALREADY!

Build a simple database with a simple table that simply records each object your scraper sees and mark it 'done' if it completed downloading it. You can test it locally first and then create a RDS (Relational Database Service) instance on AWS. Having a database.ini file keeps your python code the same between the two databases. And make sure that all the EC2 instances that need to see your RDS are in the same security group.

CATCH'EM ALL! ('em = ERRORS AND EXCEPTIONS)
It is at least 100% certain that you will get errors when running your scraper. Run it for a while and see what the most common ones are. If you can pinpoint what causes them and fix it, great. But don't spend too much time trying to predict *everything* that can go wrong and prepare an appropriate response, because it is simply impossible. There will be unique errors on the website that may appear once in tens of thousands of pages. There will be problems on the server's side that are out of your reach and hard to decipher. After a lot of frustration and time, I added a try - catch all unknown exceptions that just restarted everything. I thought it would waste a lot of time, but it did the exact opposite.
Having said that, it's not a bad idea to try a couple of times to reload a page that creates a problem. But if it fails, go nuclear and start over.

BEAUTIFUL SOUP
When this is all over you will likely have some html code to sift through. This is simply a big long string, but if you try to manipulate it with standard string operations you will soon be overwhelmed. Beautiful soup can usually extract in a few lines most of the things you could be interested in.

TROUBLESHOOTING
- I don't know how common it is, but one rudimentary way to figure out if you are a robot or a real browser is through the custom headers that are passed to the server at connection time. I did not have to deal with that, but instructions are here:
https://stackoverflow.com/questions/35666067/selenium-phantomjs-custom-headers-in-python

-A problem you may face if you do not use selenium is passing cookies or other session parameters or request properties. The site sets a cookie on your browser and oftentimes you need to pass it back, I guess as proof that this is the same legitimate connection. See here:
https://stackoverflow.com/questions/7164679/how-to-send-cookies-in-a-post-request-with-the-python-requests-library

- Finally, from what I read, it seems that there is no way to completely mask selenium. So, if a site is completely bent on blocking robots, you are out of luck.