Chapter 9

Scraping Sites that Don't Want to be Scraped/ Scraping Sites that Use Search Forms

Skills you will learn: Basic setup of the Selenium library, which allows you to control a web browser from a Python script. Filling out a form on a website and scraping the returned records.

Introduction: Sometimes, websites throw up hurdles, deliberately or incidentally, that can make scraping more difficult. For example, they may place cookies on your computer containing values that must be resubmitted with the next request to the server, failing which the request will be ignored or redirected to the landing page. These cookies may expire after a set period of time and have to be regenerated by the server. Ordinary scraping scripts using libraries such as urllib2 may fail when you attempt to scrape these kinds of sites because the script doesn't send the expected values. Similarly, pages may contain hidden import fields (basically form fields that are not visible to the user) that automatically submit long strings of alphanumeric characters to validate the request. Again, the script fails to submit the required string, and the request fails or is redirected.

On other occasions, you may need to fill out a form before the server will send data, or you may wish to use the form to selectively choose which data to scrape.

Selenium allows us to deal with these challenges because it controls an actual browser, such as Firefox, and when the script "clicks" a submit button, as an example, any cookies, hidden form fields, or JavaScript routines are handled by the browser just as they would be if a human user had clicked the button. Similarly, visible form fields can be filled out, values chosen from dropdown menus, and forms submitted to retrieve data. Selenium implements an open protocol called WebDriver that is designed to allow remote control of any major web browser. You can read more about it at https://www.w3.org/TR/webdriver/

Setting up Selenium

Getting Selenium up and running is a little more complex than the normal procedure of just importing a library and calling its functions and methods. This is because Selenium needs to make the connection to the web browser. This is usually done through the installation of an additional executable, provided by the browser vendor, that runs alongside the browser. These instructions will focus on Firefox, but you can also use Selenium with Google Chrome.

The first step is to ensure that you have the latest version of Firefox installed on your Windows PC or Mac. You can use Selenium with either platform. If you are on a Mac, make note of the location of the Firefox binary file. Note that you should avoid any version of Firefox before version 47 for compatibility reasons as the method of connecting to Selenium was different, albeit simpler. If you wish to use the previous method, you'll need to download and install a legacy version of Firefox.

Also, be sure to install Selenium itself by issuing the command pip install Selenium in a command or Terminal window. If you are unsure how to use pip, see the tutorial **Getting Python up and Running**.

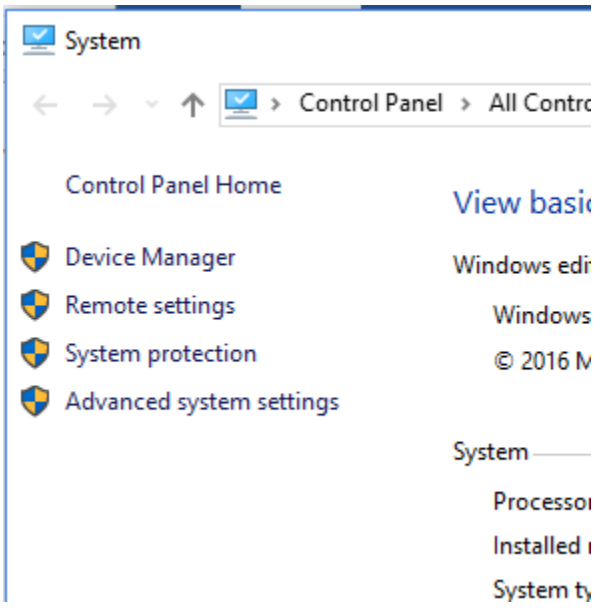
https://coolestguidesontheplanet.com/add-shell-path-osx/

Next, you will need to obtain a small binary file called geckodriver, the file that will run alongside Firefox so Selenium can do its magic. It translates the commands coming from Selenium into the Marionette automation protocol now being used by Mozilla in Firefox. You should use the latest stable version because the driver is constantly being updated to improve performance. You can read more about the technical details here: https://github.com/mozilla/geckodriver

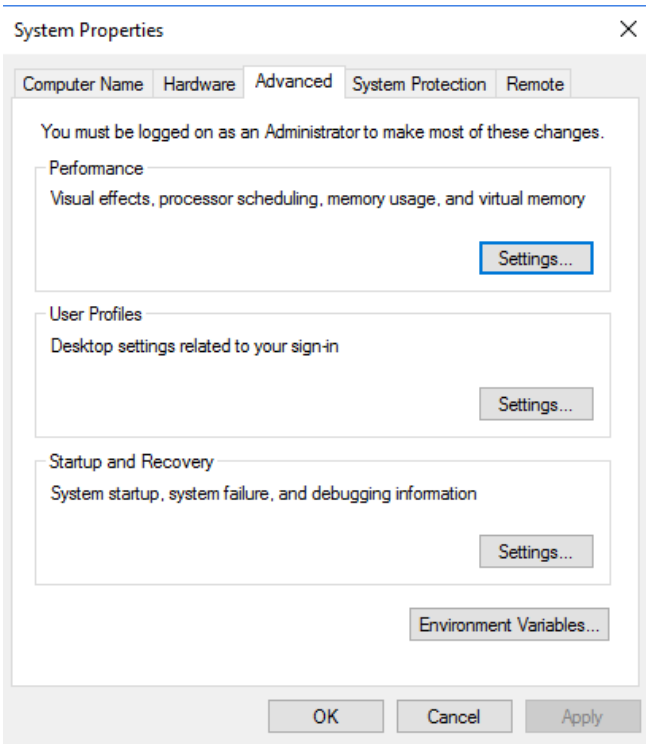
Download the geckodriver version for your operating system from https://github.com/mozilla/geckodriver/releases/. It's a compressed file, so you will need to extract it before it can be used. Make note of where you save the file. You will need that information for the next step.

You now need to modify the path system variable on your computer so Selenium will be able to find geckodriver. This is similar to the procedure used to ensure that your Mac Terminal or Windows Command window can locate the Python executable or a MySQL server.

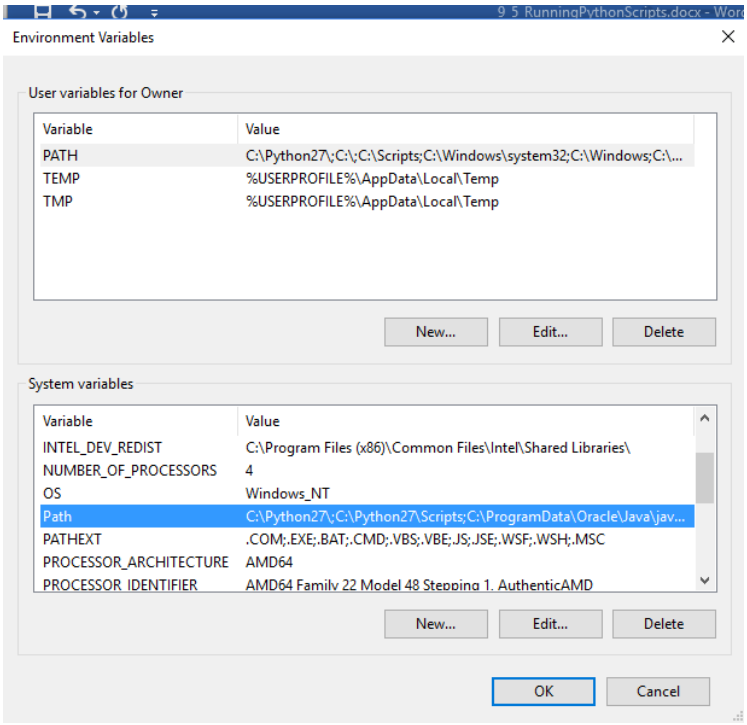
In Windows, open the Control Panel and choose System. A dialogue will open. Click on Advanced System Settings, seen here.



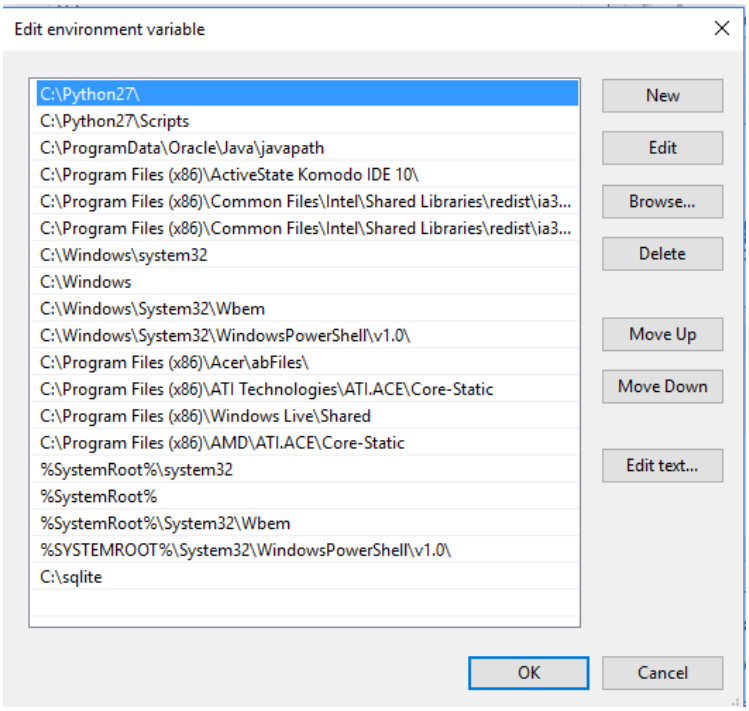
The System Properties dialogue will open to the Advanced tab. Click on Environment Variables.



In the next dialogue, highlight Path in the system variables area, then click Edit.



This will open the Edit Environment Variable dialogue.



Click New, then add the complete path to the geckodriver executable to the list. Click OK, close all dialogues, and Selenium should be able to locate the necessary driver to control Firefox.

On a Mac, you will need to add geckodriver to the path by issuing a command in Terminal. Type the following: `export PATH=$PATH:/full/file/path/to/driver`

This will be effective so long as the Terminal window session is open. You can also add geckodriver to the path permanently by creating or modifying the .bash_profile file. Open .bash_profile in a text editor, then add the following line to the file:

```
export PATH='/full path to the driver:$PATH'
```

There are many online tip sheets on creating or editing the .bash_profile file, including this one: https://coolestguidesontheplanet.com/add-shell-path-osx/

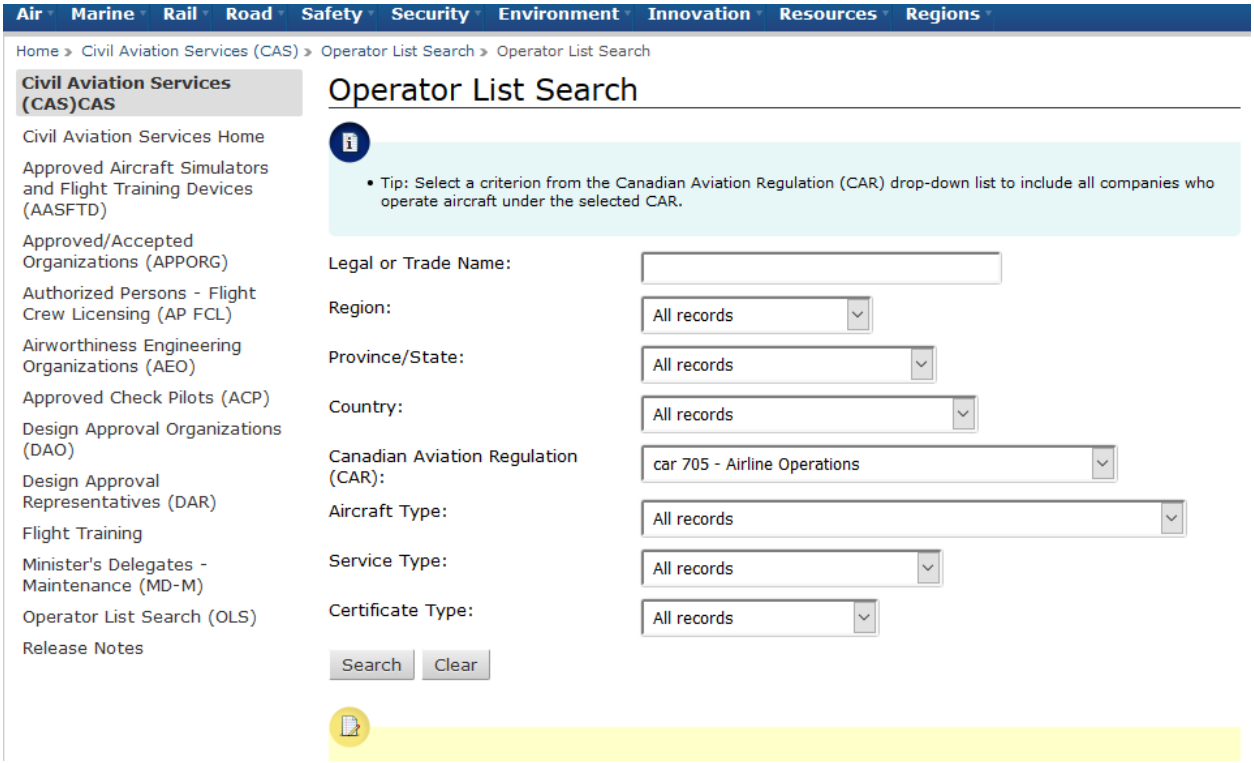
With either method of adding the location of geckodriver to your path, you can check to see if the driver actually is on the path by typing echo `$PATH` at a terminal prompt.

Note that some earlier versions of Selenium searched for an executable called "wires" rather than "geckodriver". If that is the case, rename geckodriver to wires.

Be aware that this configuration process can sometimes fail, and if it does, go back and check that you have followed each step precisely. Correct any errors. If you still can't find the answer, you should be able to find help by Googling search terms that describe the problem you are having. Stack Overflow is a particularly valuable resource for troubleshooting coding problems.

Writing your script

These steps completed, you should be able to start writing your scraping script using Selenium. For the purposes of this tutorial, we will do a simple scrape of Transport Canada's searchable database of airlines and other aviation operators in Canada. You can find it at http://wwwapps.tc.gc.ca/Saf-Sec-Sur/2/CAS-SAC/olsrlel.aspx?lang=eng This is what the search page looks like:



As you can see, you are able to search by using a keyword; any operator that contains the keyword will be returned in the results. You can also narrow the search by a variety of other criteria, available in dropdown lists. Here, CAR 704 – Airline Operations has been chosen from the Canadian Aviation Regulation dropdown.

The values entered in the various boxes will be sent to the remote web server when the Search button is clicked.

Get vs post

Web searches are sometimes submitted to the server as a get request, in which case the search criteria will be embedded in the URL. We saw that type of site in the tutorial **Scraping Using Regular Expressions**. At other times, however, the criteria will be submitted via a post request, which means they will be sent in the request headers. When that is the case, you can't simply modify the URL on the fly; you need to emulate filling out the form and sending the query criteria in the headers. There are different tools that allow you to do that. For example, you can use the Python Requests library, but as soon as a site requires session or query ids to accompany a request (these are typically saved in browser cookies), or has hidden input fields, a request may fail because the server doesn't receive information required to provide a response. Extensive use of JavaScript can also derail efforts to use such libraries.

Selenium solves these problems by leaving the heavy lifting of dealing with such requirements to the browser that it is controlling. All the script does is control the browser, automating what a human would otherwise do.

That said, let's get to work.

As in any script, the first thing we need to do is import the packages we plan to use.

```
import time, unicodecsv

from bs4 import BeautifulSoup

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

from selenium.webdriver.support.ui import Select

from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities
```

The first two lines import modules we have seen in Chapter 9 of *The Data Journalist*, and in previous tutorials.

In lines 3 to 6, we import various classes that are part of Selenium, including the core webdriver, Keys, which is used to submit text as if a user were typing on the keyboard, Select, which is used to select import fields in forms, and desired_capabilities, which we'll use to configure Selenium.

The next two lines are crucial to making the script work.

```
caps = DesiredCapabilities.FIREFOX

caps["marionette"] = True
```

In line 7, we define a new name, caps, to which we assign the capability to control Firefox, and in line 8 we instruct Selenium to use the Marionette bindings. Marionette is a driver that can be used to control Firefox

All of this is a bit technical, and for now, you'll do fine if you simply copy the six lines used to configure Selenium. You can explore the fine details of how Selenium works, later.

```
driver=webdriver.Firefox(capabilities=caps)

driver.implicitly_wait(30)
```

These two lines create our driver object, which we will use to do the heavy lifting of actually controlling Firefox.

From here, we can tell Selenium which page Firefox should open. In our case, we will provide the URL as a text string, but it could just as easily be contained in a variable, such as in a loop. To do this, we use the driver's get method. The URL is the only argument.

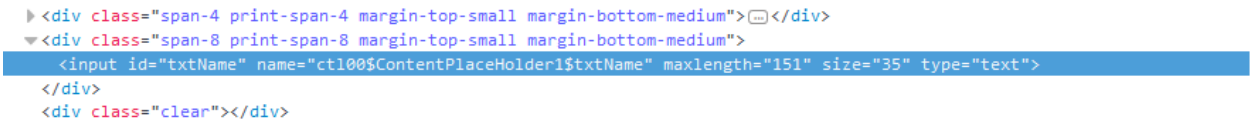
```
driver.get('http://wwwapps.tc.gc.ca/saf-sec-sur/2/CAS-
SAC/olsrlel.aspx?lang=eng')
```

This is the equivalent of using the urlopen method in urllib2, the difference between that this command causes an instance of Firefox to open. If everything works as it should , you will see Firefox open. Note that Selenium is opening the Firefox executable directly, so any plugins that you have installed in the Firefox you open manually will not be present.

Filling out a form

To fill out the form on the Transport Canada search page, we'll first need to take a look at the HTML, and to do that we'll use our developer tools.

Using the element inspector in Firefox developer tools, we can zero in on the top field, the one that allows the input of free text.



The key things to note here are the kind of tag, that is, an input tag, its id, in this case txtName, and its type, a text box.

That's enough information to write the code to fill out the text box.

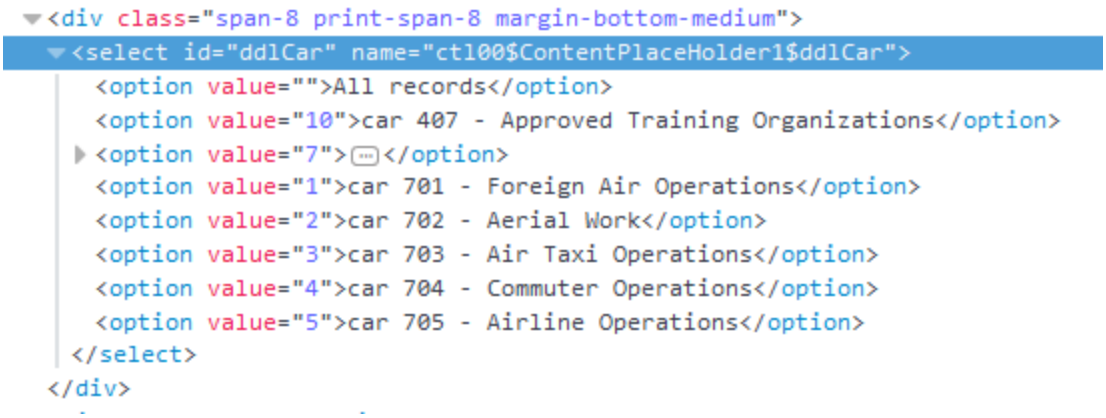
```
driver.find_element_by_id("txtName").clear()

driver.find_element_by_id("txtName").send_keys("Air")
```

The first line uses the find_element_by_id function of the driver to search through the webpage (technically, the DOM, or document object model), looking for a HTML element with the id of 'txtName'. Since an ID name should only appear once in an HTML document, that should uniquely identify our text input control. Now, we'll chain on the .clear() method, which will clear any existing content out of the text box. There's not likely to be any, but this will do no harm in that case.

The next line again uses the find_element_by_id function to find the text input box, but this time we chain on the send_keys method, which causes the word Air to be "typed" into the input box. Note that while we have written the script to enter a specific literal string, the argument to the send_keys method could just as well be a variable containing a string. In this way, you could easily program your script to include a loop so as to iterate through a series of values. This is a good technique if you want to scrape all of the data available from a searchable database that requires you to enter some minimum number of letters before it will submit a search. For example, if a site required the first three letters of any name, you could write a loop to cycle through all possible three letter combinations. For example, Aaa, followed by Aab, followed by Aac, Aad, Aae through to Aaz, then followed by Aba, and so on until you got to Zzz. Alternatively, if you had a specific list of names to search, you could put all the names in a list, then iterate over the list, feeding each list item to the send_keys method.

The remainder of the form consists of dropdown lists. Again, using the element inspector, we can examine the Canadian Aviation Regulations dropdown item.



This time, the control is a Select tag, which is used to create this kind of dropdown. It has an id of ddlCar, and eight option tags, each of which provides one of the possible text values, as well as the value attribute. We can use the ID of the select tag and the text value of the option we wish to search, in another line of code.

```
Select(driver.find_element_by_id("ddlCar")).select_by_visible_te
xt("car 705 - Airline Operations")
```

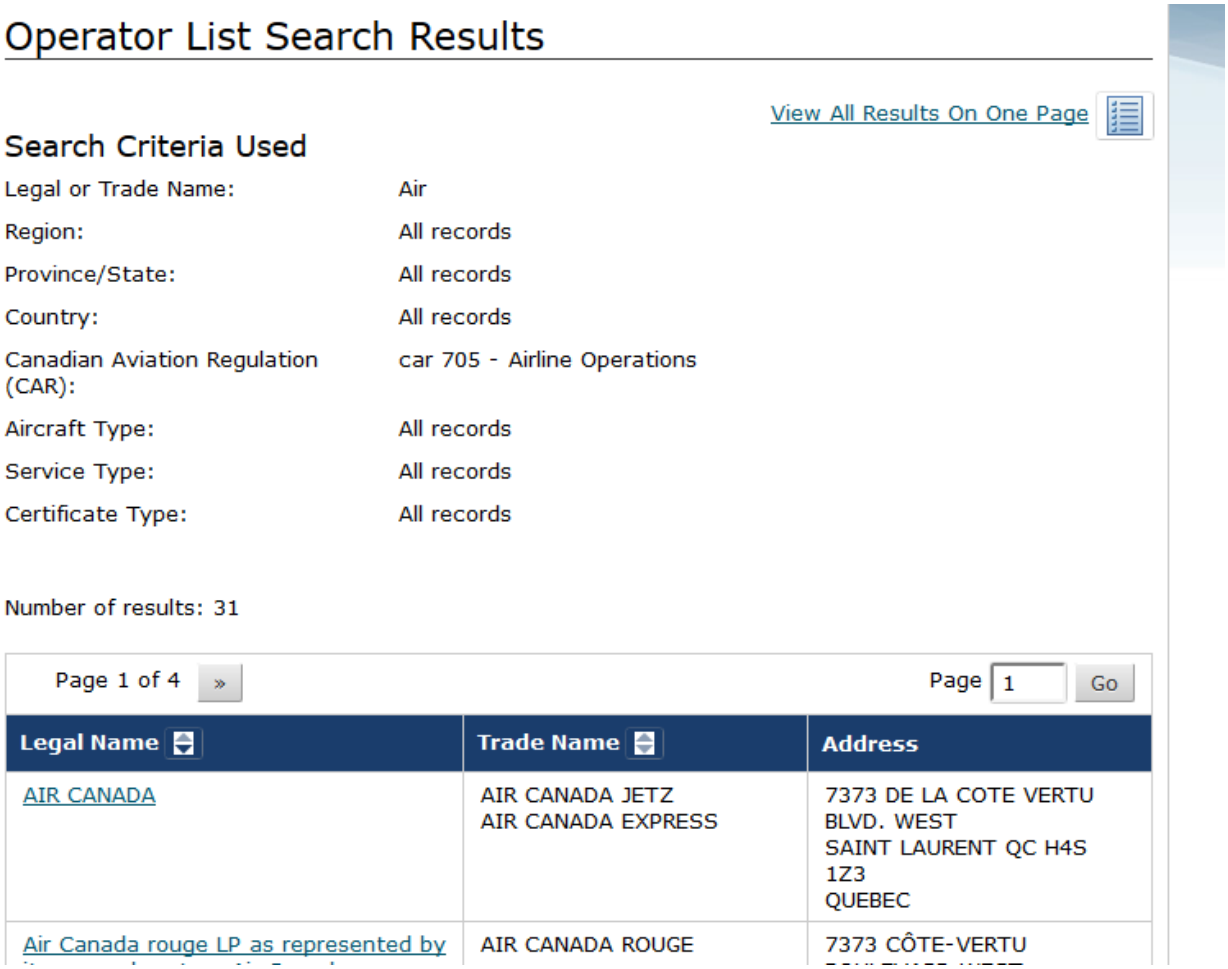
Here, we are using Selenium's Select function. As its lone argument, we use a string of functions. We call our driver once more, again using the find_element_by_id function, then chaining on the select_by_visible_text function, which will select one of the options based on the user-selectable text, "Car 705 - Airline Operations"

We have now filled out two of the form's fields. We could fill out all of the remaining dropdowns in the same way, but we'll just go with the two.

To submit the form, we need to click on the Search button, which has an ID of btnSearch. We then use the .click() method to "click" it.

```
driver.find_element_by_id("btnSearch").click()
```

If all goes well, that should produce a results page.



Operator List Search Results

View All Results On One Page

Search Criteria Used

Legal or Trade Name:	Air
Region:	All records
Province/State:	All records
Country:	All records
Canadian Aviation Regulation (CAR):	car 705 - Airline Operations
Aircraft Type:	All records
Service Type:	All records
Certificate Type:	All records

Number of results: 31

Page 1 of 4 » Page 1 Go

Legal Name	Trade Name	Address
AIR CANADA	AIR CANADA JETZ AIR CANADA EXPRESS	7373 DE LA COTE VERTU BLVD. WEST SAINT LAURENT QC H4S 1Z3 QUEBEC
Air Canada rouge LP as represented by	AIR CANADA ROUGE	7373 CÔTE-VERTU

There's one more thing to do, though. The results are divided into four pages.

A close look at the HTML code behind the link View All Results on One Page reveals that it is another input control. So we'll write a bit of code to click on it.

```
driver.find_element_by_id("btnAll").click()
```

That should cause the server to send back a new page, with all the results on one page. A look using the element inspector of the HTML code behind the list of results shows it is a standard HTML table.

```
<table id="ctl00_ContentPlaceHolder1_gridview1" class="ASPgridview width-100" style="border-collapse:collapse;"
cellspacing="0">
  <tbody>
    <tr>
      <td colspan="3">
        <div id="ctl00_ContentPlaceHolder1_gridview1_ctl01_pnlPaging" onkeypress="javascript:return
        WebForm_FireDefaultButton(event, 'ctl00_ContentPlaceHolder1_gridview1_ctl01_btnGoToPage')">
          <div class="span-5 margin-bottom-small print-span-6">…</div>
          <div class="span-6 align-right float-right margin-bottom-small print-span-6 cn-invisible-print">…</div>
        </div>
      </td>
    </tr>
```

From here, we can use the methods described in Chapter 9 of *The Data Journalist* to scrape the actual data from the table.

The first step will be to get the text of the fetched webpage. To do that, we'll add, right after the last line, the following:

```
time.sleep(10)

mainPage = driver.page_source
```

The first line is the now familiar sleep command. We're using it here to ensure that the new page has time to load. It also programs in a gap between requests to the server. Once the waiting period expires, the driver.page_source attribute returns the html content of the page currently loaded in the browser, as a text string.

We can now use the Beautiful Soup library, just as we did in Chapter 9, to grab the data from the HTML table and write it to a CSV file. We won't repeat the detailed instructions here, so please see Chapter 9 if you would like to review the procedures.

This is what the final code looks like:

```
import time, unicodecsv

from bs4 import BeautifulSoup

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

from selenium.webdriver.support.ui import Select

from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities

caps = DesiredCapabilities.FIREFOX

caps["marionette"] = True

outfile = open("c:/WriteFiles/AirOperators.csv",'wb')

writer = unicodecsv.writer(outfile,dialect='excel')

writer.writerow(['Name','Trade Name(s)','Address'])

driver=webdriver.Firefox(capabilities=caps)

driver.implicitly_wait(30)

driver.get('http://wwwapps.tc.gc.ca/saf-sec-sur/2/CAS-
SAC/olsrlel.aspx?lang=eng')

driver.find_element_by_id("txtName").clear()

driver.find_element_by_id("txtName").send_keys("Air")

Select(driver.find_element_by_id("ddlCar")).select_by_visible_te
xt("car 705 - Airline Operations")

driver.find_element_by_id("btnSearch").click()
```

```
driver.find_element_by_id("btnAll").click()
time.sleep(10)
mainPage = driver.page_source
soup = BeautifulSoup(mainPage,'html.parser')
theTable = soup.find('table')
theRows = theTable.findAll('tr')
for row in theRows:
    outList = []
    theCells = row.findAll('td')
    for cell in theCells:
        outList.append(cell.text.strip())
    writer.writerow(outList)
```

You might also wish to scrape the detail page for each air operator, which is found by clicking on the link for the name of the operator in the table. The routine to do this can be written in to the final for loop of the existing script:

```
    theCells = row.findAll('td')
    for cell in theCells:
        outList.append(cell.text.strip())
    if len(theCells) > 0:
        detailURL = 'http://wwwapps.tc.gc.ca/saf-sec-sur/2/CAS-
SAC/'+theCells[0].find('a').get('href')
        driver.get(detailURL)
        time.sleep(10)
        secondPage = driver.page_source
```

From this point, you could use Beautiful Soup to extract the contents of the detail page, which are contained in one or more HTML tables.

Further horizons

We have only touched on a few of the things that Selenium's webdriver can do. You can find more detail on its various functions at http://www.seleniumhq.org/docs/03_webdriver.jsp

Selenium has methods that allow you to traverse the DOM of a webpage and extract individual elements, much the way Beautiful Soup does. You can write scripts that take advantage of these elements and not use Beautiful Soup at all. It all depends on our preference.

There is one more tool that you may find handy, and that is the Selenium IDE plugin for Firefox. This plugin allows you to record the actions you take as you manually navigate links, fill out forms, etc. It then generates code that you can paste into your own script, saving you development time. Go to Tools>Add-ons in Firefox, search for Selenium IDE, and install it.

Finally, Mozilla has also released a Python library that allows you to control Python using the Marionette driver, without using Selenium. Unlike when using Selenium, you can use this official library to control features of the Browser itself, something Selenium doesn't do. This may, in time, become a good alternative to Selenium for scraping and we will amend this tutorial as necessary to reflect advances in the technology.