Chapter 9

Scraping Sites That Use JavaScript and AJAX

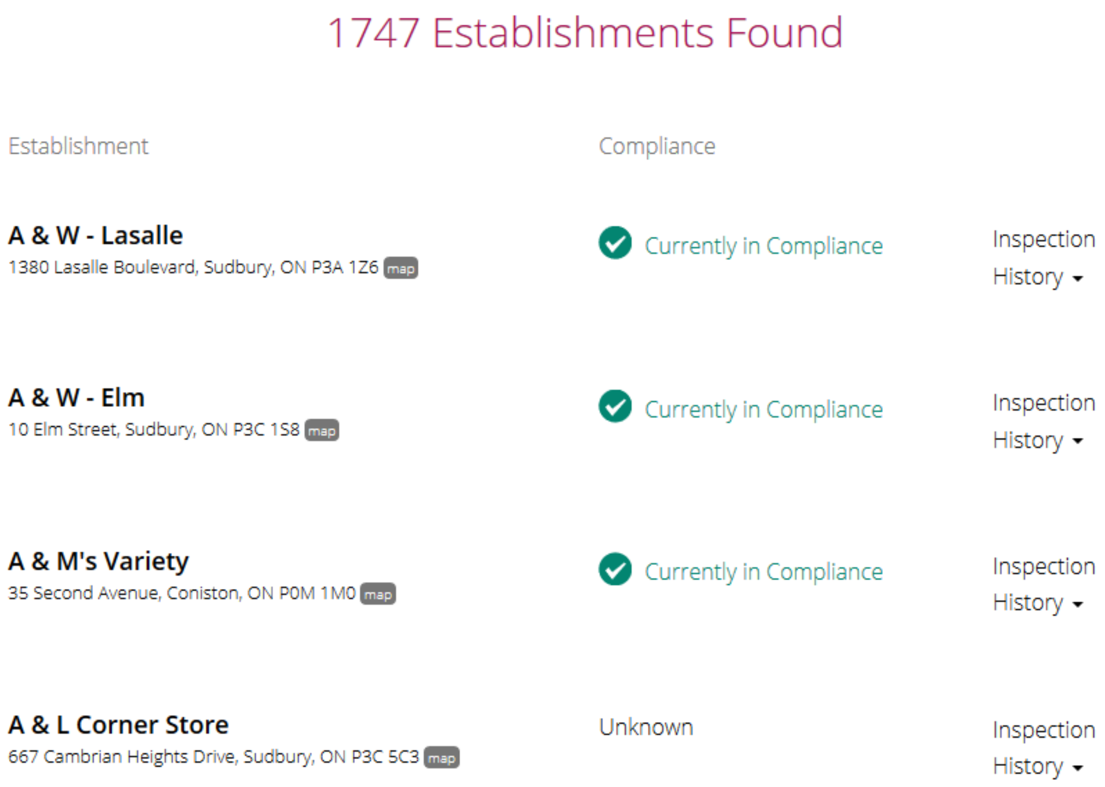
As of August 2017, the website used for this tutorial had been archived by the Sudbury and District Health Unit, and was soon to be replaced. This tutorial has been updated to use the embedded Firebox developer tools. With the coming phaseout of the Sudbury food site, this tutorial will be replaced by a revised tutorial scraping a different live site. Once complete, the new tutorial will be posted to thedatajournalist.ca/newtutorials and will be posted on the Oxford University Press ARC site during the next refresh in summer 2018.

Skills you will learn: How to make GET requests to obtain data that updates a web page using Ajax; building a more complex multi-layer Ajax scrape; creating functions to reduce duplication in your code; using the JSON module to parse JSON data; Python dictionaries; Python error handling; how to use Firebug as part of the development of a more difficult scraping project.

Getting started

Like many communities throughout Canada and the U.S., the Sudbury and District Health Unit provides basic details about health inspections to the public via a website. You can go to the site at https://www.sdhu.com/inspection-results#/

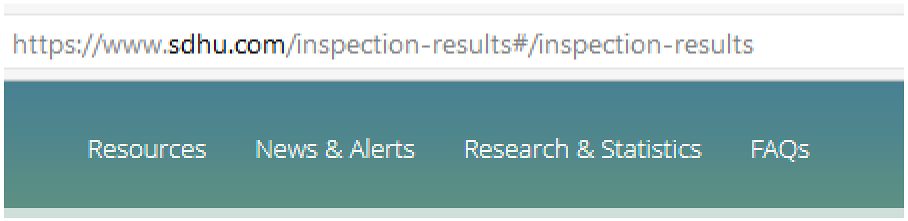
If you click on the Inspection Results icon you'll see that the page soon populates with basic results of inspections.


1747 Establishments Found

Establishment	Compliance	
A & W - Lasalle 1380 Lasalle Boulevard, Sudbury, ON P3A 1Z6 map	✓ Currently in Compliance	Inspection History
A & W - Elm 10 Elm Street, Sudbury, ON P3C 1S8 map	✓ Currently in Compliance	Inspection History
A & M's Variety 35 Second Avenue, Coniston, ON P0M 1M0 map	✓ Currently in Compliance	Inspection History
A & L Corner Store 667 Cambrian Heights Drive, Sudbury, ON P3C 5C3 map	Unknown	Inspection History

If you are a journalist in the Sudbury area, you'd probably love to analyze the performance of establishments in health inspections. But while the data is online, there's no easy link to download it. You may have to scrape it. Trouble is, this page is a toughie.

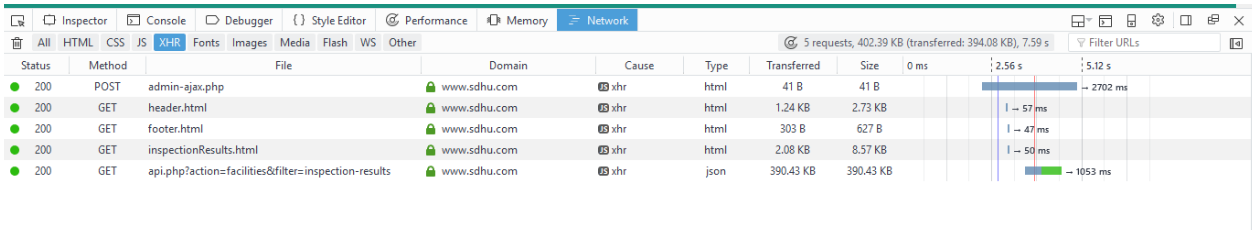
If you take a look at the HTML of the page using Page Source we see soon enough that the results we see on the screen are not in the page source. There is a lot of JavaScript code, and some skeletal HTML for contact information and social media icons, but little else. If you click to get details on one of the establishments, the browser updates with the information requested, but URL for the page stays the same.


https://www.sdhu.com/inspection-results#/inspection-results

This means that if we were to write a scraping script that tried to parse the inspection results out of the HTML page sent by the web server, we would be stopped in our tracks. The page is using JavaScript and the AJAX protocol to populate the page we see in the browser (if you are unsure of what Ajax is, see the explanation in Chapter 9 of *The Data Journalist).*

Fortunately, there are ways we can grab the data being sent by the server to update the page.

We'll begin by having a look at what is happening using the network tab in Firefox developer tools. If you are unsure about the basics of development tools, see the tutorial **Using Development Tools to Examine Webpages** on the companion site to *The Data Journalist*.



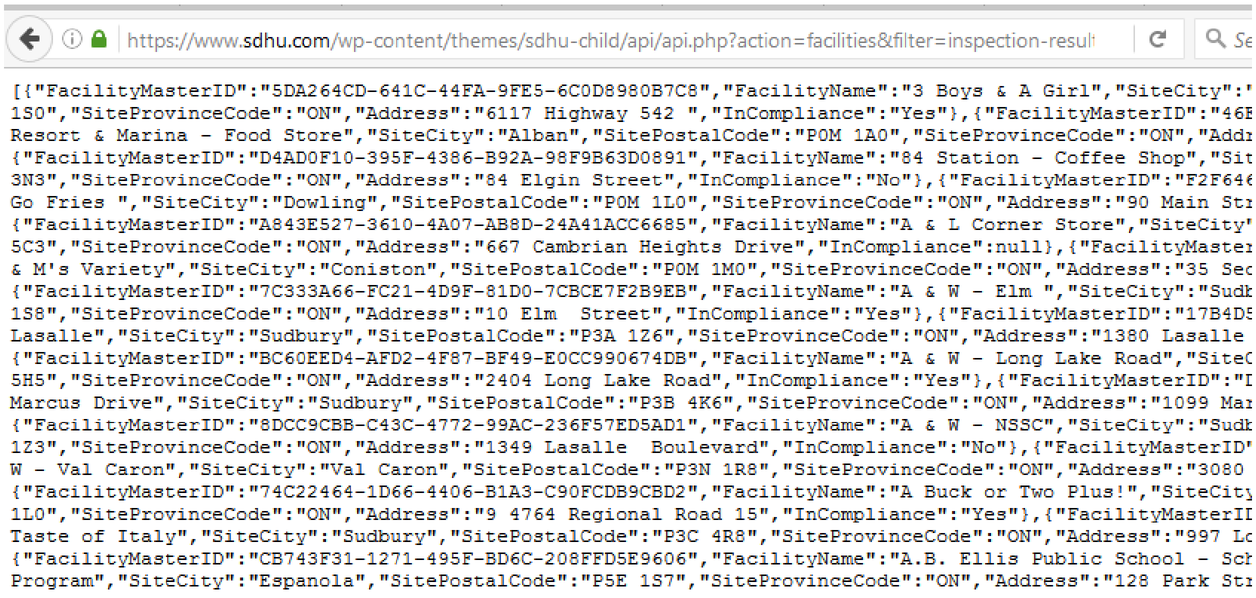
```
5 requests, 402.39 KB (transferred: 394.08 KB), 7.59 s

Status  Method  File                                          Domain         Cause  Type  Transferred  Size       0 ms   2.56 s   5.12 s
200     POST    admin-ajax.php                                www.sdhu.com   xhr    html  41 B         41 B       → 2702 ms
200     GET     header.html                                   www.sdhu.com   xhr    html  1.24 KB      2.73 KB    → 57 ms
200     GET     footer.html                                   www.sdhu.com   xhr    html  303 B        627 B      → 47 ms
200     GET     inspectionResults.html                        www.sdhu.com   xhr    html  2.08 KB      8.57 KB    → 50 ms
200     GET     api.php?action=facilities&filter=inspection-results  www.sdhu.com   xhr    json  390.43 KB    390.43 KB  → 1053 ms
```

We can see that there was one XHR request made for JSON data and it was a GET request. That means that we can make the request ourselves by using the same URL.

(If the site used a POST request, one that sends the data for the request as part of the HTML headers of the request, we'd have to handle things differently.

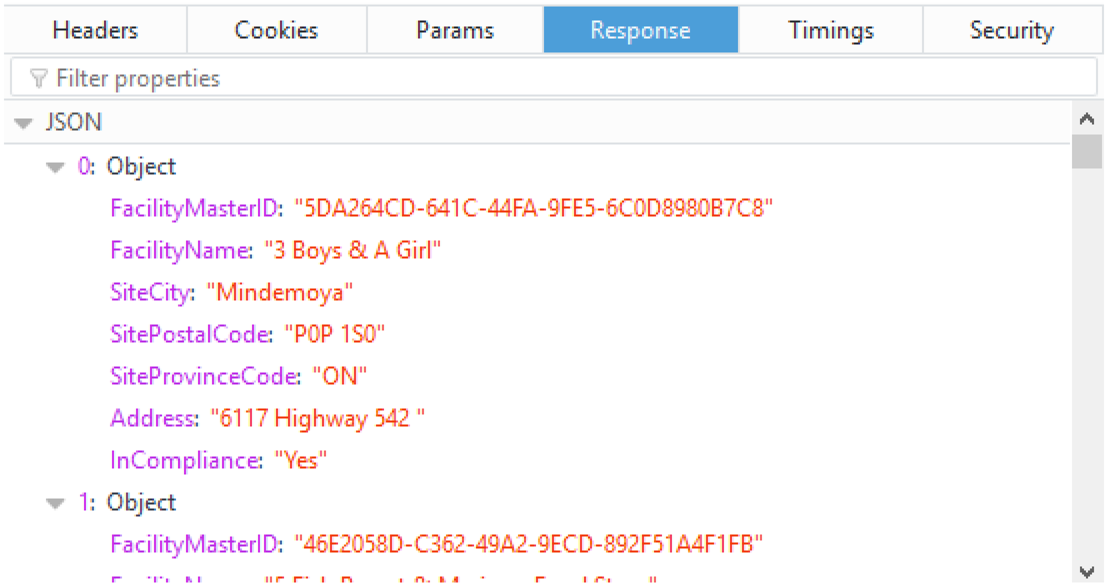
In fact, if we paste the URL into a new browser tab, we can see the response, which is some JSON.



```
https://www.sdhu.com/wp-content/themes/sdhu-child/api/api.php?action=facilities&filter=inspection-resul
```

```
[{"FacilityMasterID":"5DA264CD-641C-44FA-9FE5-6C0D8980B7C8","FacilityName":"3 Boys & A Girl","SiteCity":"
1S0","SiteProvinceCode":"ON","Address":"6117 Highway 542 ","InCompliance":"Yes"},{"FacilityMasterID":"46
Resort & Marina - Food Store","SiteCity":"Alban","SitePostalCode":"P0M 1A0","SiteProvinceCode":"ON","Add
{"FacilityMasterID":"D4AD0F10-395F-4386-B92A-98F9B63D0891","FacilityName":"84 Station - Coffee Shop","Sit
3N3","SiteProvinceCode":"ON","Address":"84 Elgin Street","InCompliance":"No"},{"FacilityMasterID":"F2F64
Go Fries ","SiteCity":"Dowling","SitePostalCode":"P0M 1L0","SiteProvinceCode":"ON","Address":"90 Main St
{"FacilityMasterID":"A843E527-3610-4A07-AB8D-24A41ACC6685","FacilityName":"A & L Corner Store","SiteCity"
5C3","SiteProvinceCode":"ON","Address":"667 Cambrian Heights Drive","InCompliance":null},{"FacilityMaste
& M's Variety","SiteCity":"Coniston","SitePostalCode":"P0M 1M0","SiteProvinceCode":"ON","Address":"35 Se
{"FacilityMasterID":"7C333A66-FC21-4D9F-81D0-7CBCE7F2B9EB","FacilityName":"A & W - Elm ","SiteCity":"Sud
1S8","SiteProvinceCode":"ON","Address":"10 Elm  Street","InCompliance":"Yes"},{"FacilityMasterID":"17B4D
Lasalle","SiteCity":"Sudbury","SitePostalCode":"P3A 1Z6","SiteProvinceCode":"ON","Address":"1380 Lasalle
{"FacilityMasterID":"BC60EED4-AFD2-4F87-BF49-E0CC990674DB","FacilityName":"A & W - Long Lake Road","Site
5H5","SiteProvinceCode":"ON","Address":"2404 Long Lake Road","InCompliance":"Yes"},{"FacilityMasterID":"
Marcus Drive","SiteCity":"Sudbury","SitePostalCode":"P3B 4K6","SiteProvinceCode":"ON","Address":"1099 Ma
{"FacilityMasterID":"8DCC9CBB-C43C-4772-99AC-236F57ED5AD1","FacilityName":"A & W - NSSC","SiteCity":"Sud
1Z3","SiteProvinceCode":"ON","Address":"1349 Lasalle  Boulevard","InCompliance":"No"},{"FacilityMasterID"
W - Val Caron","SiteCity":"Val Caron","SitePostalCode":"P3N 1R8","SiteProvinceCode":"ON","Address":"3080
{"FacilityMasterID":"74C22464-1D66-4406-B1A3-C90FCDB9CBD2","FacilityName":"A Buck or Two Plus!","SiteCit
1L0","SiteProvinceCode":"ON","Address":"9 4764 Regional Road 15","InCompliance":"Yes"},{"FacilityMasterI
Taste of Italy","SiteCity":"Sudbury","SitePostalCode":"P3C 4R8","SiteProvinceCode":"ON","Address":"997 L
{"FacilityMasterID":"CB743F31-1271-495F-BD6C-208FFD5E9606","FacilityName":"A.B. Ellis Public School - Sc
Program","SiteCity":"Espanola","SitePostalCode":"P5E 1S7","SiteProvinceCode":"ON","Address":"128 Park St
```

We can also see the JSON in a neater, easier-to-read format in Firefox developer tools:



```
JSON
  0: Object
    FacilityMasterID: "5DA264CD-641C-44FA-9FE5-6C0D8980B7C8"
    FacilityName: "3 Boys & A Girl"
    SiteCity: "Mindemoya"
    SitePostalCode: "P0P 1S0"
    SiteProvinceCode: "ON"
    Address: "6117 Highway 542 "
    InCompliance: "Yes"
  1: Object
    FacilityMasterID: "46E2058D-C362-49A2-9ECD-892F51A4F1FB"
```

If we liked, and all we wanted was a list of the businesses, their locations, and whether they are currently in compliance, we could simply copy and paste this JSON into an online converter such as http://www.convertcsv.com/json-to-csv.htm, and paste the result directly into Excel. The simple scraper we will build next will duplicate that functionality.

Because the JSON can be fetched via a simple GET request, all our scraper needs to do is send a request in the usual way, then parse the JSON and turn it into a CSV file. The first part is nothing we haven't done in simpler scrapes, and the second can be accomplished using Python's built in JSON module.

The first three lines of our script aren't any different from scripts we have written before.

```
1. import urllib2
2. import json
3. import unicodecsv
4. mainPage = urllib2.urlopen('https://www.sdhu.com/wp-
   content/themes/sdhu-child/api/api.php?action=facilities
   &filter=inspection-results').read()
5. output =
   open('C:\Users\Owner\Dropbox\NewDataBook\Tutorials\Chapter9\9_9_Java
   scriptScrapes_AJAX\SudburyFood.csv','w')
6. fieldNames =
   ['FacilityMasterID','FacilityName','SiteCity','SitePostalCode','Site
   ProvinceCode','Address','InCompliance']
```

The first three lines import the modules we will use in the script. All but unicodecsv are standard library modules, but unicodecsv will have to be installed using pip if you haven't already done so.

Line 4 uses urllib2's urlopen method to make a request to the URL we extracted using Firebug and assign the response object to the name 'mainpage'. In line 5 we open a new file for writing and assign the file-like object to the name 'output.' Line 6 assigns a list containing the file headers for the data to the name ‘fieldName.’ We figured out the headers by examining the JSON in Firefox developer tools.

We could, if we wanted, reorder the fields to whatever order we liked, because the dictionary writer we will create in the next line will order the fields in the output CSV by whatever order we choose for the fieldnames in the fieldnames = argument. The key thing is that the fieldnames listed in the dictionary must be present in the list of fieldnames, spelled exactly the same way. Otherwise, you will get an error.

The next line creates a unicodecsv Dictwriter object and assigns it to the name ‘writer.’ In previous tutorials and in the body of Chapter 9, we used the regular writer object and passed in parameters for the file delimiter to be used and the encoding. But unicodecsv and the standard library csv module also have a method for writing Python dictionaries to csv files. As the output of our JSON module is a dictionary and we don’t really need to alter the output at all, we’ll just write the dictionary directly to the output file. More on dictionaries in a moment.

```
7. writer = unicodecsv.DictWriter(output,fieldnames = fieldNames)
8. writer.writeheader()
```

In line 8 we write the headers using our Dictwriter’s writeheader() method. We set the fieldnames in line 7, so these are the names that writeheader() will use for the headers.

In line 9, we will put the JSON module to use. The JSON module has a method called .loads() that parses JSON, converting each JSON object into a dictionary, and an array (array is the term used in JavaScript and many other programing languages for what Python calls a list) of JSON objects into a list of dictionaries.

```
9. theJSON = json.loads(mainPage)
```

A primer of dictionaries

We haven’t talked much about dictionaries. Dictionaries are a python data type that is what computer scientists call a mapping. This isn’t the kind of mapping we dealt with in Chapter 6, of course, but instead is the mapping of values. In a dictionary, values are stored in key/value pairs. Each value is bound to a key. Think of it as being a lot like a table in Excel or a database. The key is the field name, the value is the field value. This is an example of a simple dictionary given in the official Python documentation at https://docs.python.org/2/tutorial/datastructures.html

```
tel = {'jack': 4098, 'sape': 4139}
```

Here, a new dictionary is assigned to the name tel, for telephone numbers, and each key value pair in the dictionary is a name and its associated phone number.

To see the value for any key/value pair, you use a construct quite similar to that which we saw with lists, except that instead of using an index number, we use the key name to get the value.

```
tel['jack']
```

You can also extract the value for any key value pair using the get() method.

```
tel.get('jack')
```

This sort of dictionary is useful for a very simple task, such as storing a single phone number. More complex dictionaries can be used to hold many different pieces of data.

For example, this dictionary stores information about the author of this tutorial.

```
{'name':'Fred Vallance-Jones','employer':'University of king\'s
college','expertise':'Python','sport':'cross-country skiing'}
```

If we wanted to have numerous authors, we could create either a list of dictionaries, or a dictionary with additional embedded dictionaries. Here is a simple list of dictionaries with the two authors and two contributors of this book.

```
authors = [{'name':'Fred Vallance-Jones','employer':'University of
kings college','expertise':'Python','sport':'cross-country skiing'},
{'name':'David McKie','employer':'CBC', 'expertise'\notice
:'Excel','sport':'cycling'},
{'name':'William Wolfe-Wylie','employer':'CBC','expertise':'JavaScript
development','sport':'cycling'},
{'name':'Glen McGregor','employer':'CTV','expertise':'Investigative
Reporting','sport':'walking'}]
```

We can now use standard list indexing to grab the first element in the list, which is a dictionary.

```
authors[0]
```

Gives us…

```
{'sport': 'cross-country skiing', 'expertise': 'Python', 'name': 'Fred
Vallance-Jones', 'employer': 'University of kings college'}
```

Notice that the key/value pairs were not returned in the same order in which we entered them in the original dictionary. That's a normal behaviour of dictionaries, and it doesn't really matter because we can always extract the value we want using its named key.

If we combine the previous command with our .get() method for the embedded dictionary, we would write:

```
authors[0].get('expertise')
```

```
Which would give us:
```

```
'Python'
```

We could represent the same data in a pure dictionary, with the value for each name being another dictionary structure:

```
authors = {'Fred Vallance-Jones':{'employer':'University of kings
college','expertise':'Python','sport':'cross-country skiing'},
'David McKie':{'employer':'CBC',
'expertise':'Excel','sport':'cycling'},
'William Wolfe-Wylie':{'employer':'CBC','expertise':'JavaScript
development','sport':'cycling'},
'Glen McGregor':{'employer':'CTV','expertise':'Investigative
Reporting','sport':'walking'}}
```

We could then extract the information for any one of the authors using the get() method:

```
authors.get('Fred Vallance-Jones')
```

Would give us….

```
{'sport': 'cross-country skiing', 'expertise': 'Python', 'employer':
'University of kings college'}
```

To retrieve a single value from the above dictionary we could write:

```
authors['Fred Vallance-Jones'].get('expertise')
```

Because the value for each key/value pair in the main dictionary is another dictionary, we use the get method to extract the value from the specified key in the inner dictionary.

```
'Python'
```

This kind of statement is at the heart of parsing dictionaries, something we will have to do when we use the JSON module to turn the JSON returned by the web server into a dictionary.

Alright, let's go back to our script. Remember that we had written this in line 9:

```
theJSON = json.loads(mainPage)
```

If we were to print theJSON to the screen, we would see a list of dictionaries much like we examined a moment ago. Here is a short excerpt:

```
[
{u'InCompliance': u'Yes', u'SitePostalCode': u'P0P 1S0',
u'FacilityMasterID': u'5DA264CD-641C-44FA-9FE5-6C0D8980B7C8',
u'SiteProvinceCode': u'ON', u'SiteCity': u'Mindemoya',
u'FacilityName': u'3 Boys & A Girl', u'Address': u'6117 Highway 542
'}, {u'InCompliance': u'Yes', u'SitePostalCode': u'P0M 1A0',
u'FacilityMasterID': u'46E2058D-C362-49A2-9ECD-892F51A4F1FB',
u'SiteProvinceCode': u'ON', u'SiteCity': u'Alban', u'FacilityName':
u'5 Fish Resort & Marina - Food Store', u'Address': u'25 Whippoorwill
Rd'}
]
```

To make the structure obvious, we have put the square brackets that surround the list on separate lines here. The small u characters before each text entry indicate that the text is encoded as Unicode.

As you can see this structure is the same as our list of author dictionaries. Each item in the list is a dictionary.

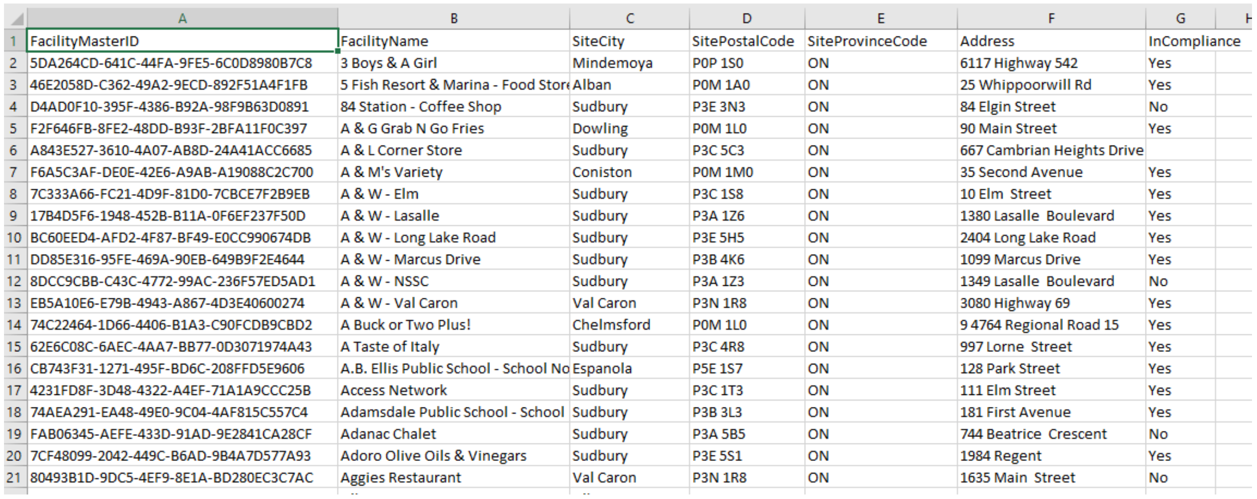
From here, our script is straightforward.

The last two lines will loop through the list of dictionaries, and write each line to the CSV user the unicodecsv DictWriter we created back in line 7.

```
10.   for dict in theJSON:
11.         writer.writerow(dict)
```

Line 10 initiates a standard for loop which will iterate through the list of dictionaries that we named theJSON. We'll call our iteration variable dict to make it clearer that each item in the list is a dictionary. Finally, in line 11, we'll use our DictWriter to write the dictionary to the CSV.

The result, opened in Excel, looks like this:



	A	B	C	D	E	F	G
1	FacilityMasterID	FacilityName	SiteCity	SitePostalCode	SiteProvinceCode	Address	InCompliance
2	5DA264CD-641C-44FA-9FE5-6C0D8980B7C8	3 Boys & A Girl	Mindemoya	P0P 1S0	ON	6117 Highway 542	Yes
3	46E2058D-C362-49A2-9ECD-892F51A4F1FB	5 Fish Resort & Marina - Food Stor	Alban	P0M 1A0	ON	25 Whippoorwill Rd	Yes
4	D4AD0F10-395F-4386-B92A-98F9B63D0891	84 Station - Coffee Shop	Sudbury	P3E 3N3	ON	84 Elgin Street	No
5	F2F646FB-8FE2-48DD-B93F-2BFA11F0C397	A & G Grab N Go Fries	Dowling	P0M 1L0	ON	90 Main Street	Yes
6	A843E527-3610-4A07-AB8D-24A41ACC6685	A & L Corner Store	Sudbury	P3C 5C3	ON	667 Cambrian Heights Drive	
7	F6A5C3AF-DE0E-42E6-A9AB-A19088C2C700	A & M's Variety	Coniston	P0M 1M0	ON	35 Second Avenue	Yes
8	7C333A66-FC21-4D9F-81D0-7CBCE7F2B9EB	A & W - Elm	Sudbury	P3C 1S8	ON	10 Elm Street	Yes
9	17B4D5F6-1948-452B-B11A-0F6EF237F50D	A & W - Lasalle	Sudbury	P3A 1Z6	ON	1380 Lasalle Boulevard	Yes
10	BC60EED4-AFD2-4F87-BF49-E0CC990674DB	A & W - Long Lake Road	Sudbury	P3E 5H5	ON	2404 Long Lake Road	Yes
11	DD85E316-95FE-469A-90EB-649B9F2E4644	A & W - Marcus Drive	Sudbury	P3B 4K6	ON	1099 Marcus Drive	Yes
12	8DCC9CBB-C43C-4772-99AC-236F57ED5AD1	A & W - NSSC	Sudbury	P3A 1Z3	ON	1349 Lasalle Boulevard	No
13	EB5A10E6-E79B-4943-A867-4D3E40600274	A & W - Val Caron	Val Caron	P3N 1R8	ON	3080 Highway 69	Yes
14	74C22464-1D66-4406-B1A3-C90FCDB9CBD2	A Buck or Two Plus!	Chelmsford	P0M 1L0	ON	9 4764 Regional Road 15	Yes
15	62E6C08C-6AEC-4AA7-BB77-0D3071974A43	A Taste of Italy	Sudbury	P3C 4R8	ON	997 Lorne Street	Yes
16	CB743F31-1271-495F-BD6C-208FFD5E9606	A.B. Ellis Public School - School No	Espanola	P5E 1S7	ON	128 Park Street	Yes
17	4231FD8F-3D48-4322-A4EF-71A1A9CCC25B	Access Network	Sudbury	P3C 1T3	ON	111 Elm Street	Yes
18	74AEA291-EA48-49E0-9C04-4AF815C557C4	Adamsdale Public School - School	Sudbury	P3B 3L3	ON	181 First Avenue	Yes
19	FAB06345-AEFE-433D-91AD-9E2841CA28CF	Adanac Chalet	Sudbury	P3A 5B5	ON	744 Beatrice Crescent	No
20	7CF48099-2042-449C-B6AD-9B4A7D577A93	Adoro Olive Oils & Vinegars	Sudbury	P3E 5S1	ON	1984 Regent	Yes
21	80493B1D-9DC5-4EF9-8E1A-BD280EC3C7AC	Aggies Restaurant	Val Caron	P3N 1R8	ON	1635 Main Street	No

Fantastic! We scraped a site that uses JSON.

But this site has more than one layer of information, each one reached by clicking on more links/icons that fire more JavaScript and more AJAX calls back to the server.

From here, the logic involved in scraping this site gets a lot more involved. We will write loops within loops within loops, and write to three different output files. If you are just beginning, some of what follows will bend your brain into contortions you didn't realize were possible. We will introduce several Python constructs we have either never seen before, or just discussed in passing, including functions and error handling. It's a lot of material, and more advanced than we have seen till now, so don't despair if you have trouble understanding it the first time through.

Alright, let's get back to our layers of JSON.

For each establishment, a visitor to the website can click on an Inspection History dropdown arrow:

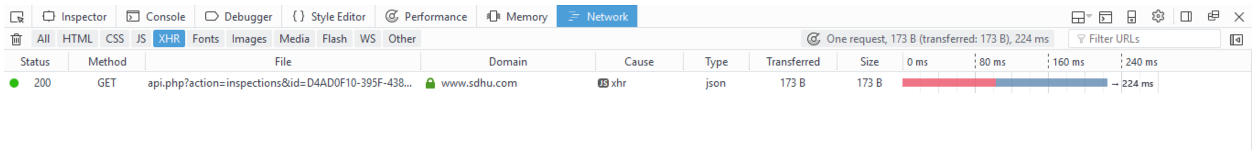
84 Station - Coffee Shop
84 Elgin Street, Sudbury, ON P3E 3N3 map

Not in Compliance

Inspection History

That then reveals details about when the establishment was inspected and the overall result.

Details are only provided for the establishment you clicked on, because the page is being changed through another AJAX call. Remember that we can see what is going on when we interact with a browser by using Firefox developer tools. Let's see what shows up in the network tab when we click on the Inspection History link.



One request, 173 B (transferred: 173 B), 224 ms

Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms 80 ms 160 ms 240 ms
200	GET	api.php?action=inspections&id=D4AD0F10-395F-438...	www.sdhu.com	xhr	json	173 B	173 B	→ 224 ms

As you can see, one GET request was made to the server to get the detail on the inspection. It had this URL when we scraped it:

https://www.sdhu.com/wp-content/themes/sdhu-child/api/api.php?action=inspections&id=D4AD0F10-395F-4386-B92A-98F9B63D0891

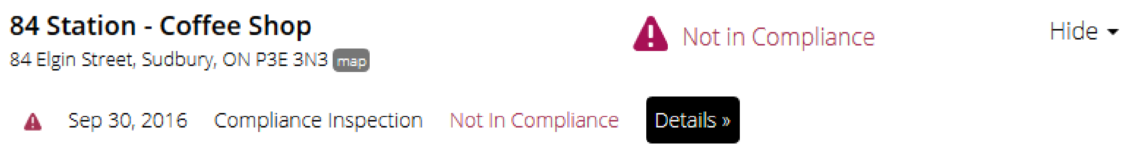
The most interesting thing about this request is that it contains a long alphanumeric ID at the end that looks just like the facility master ID we scraped for each establishment, from the main page. It looks as if that ID is being used to tell the server, when you click on the Inspection History dropdown, which details to return. Everything that follows id= is the unique ID. That being the case, we can use the ID to fetch the details for each establishment, in a script. Let's look at the response that comes back, which is again, JSON.

[{"InspectionID":"FEEDB8CC-02F9-4B6A-8FE0-B705DE810D7B","InspectionDate":"2016-09-30 00:00:00","RowStatus":"A","InCompliance":"No","InspectionType":"Compliance Inspection"}]

We see the same pattern as we saw in the JSON for the main page, an array of JSON objects, each of which has a unique inspection ID, an inspection date, something called RowStatus, a field indicating if the establishment was in compliance, and the inspection type. In the above image you can see three separate JSON objects in the array, separated by commas. We know from our previous example that if we use the loads() method in the JSON module that this array of JSON objects will be converted to a Python list of dictionaries, from which we will be able to extract the details we want.

Also note that the data structure that is being used mirrors a relational database, and was probably extracted from one. For each establishment there is a main ID, and for each inspection, an inspection ID.

But we're not done. If we go back to the page in our browser, you can see that for each inspection, there is a link for even more detail.



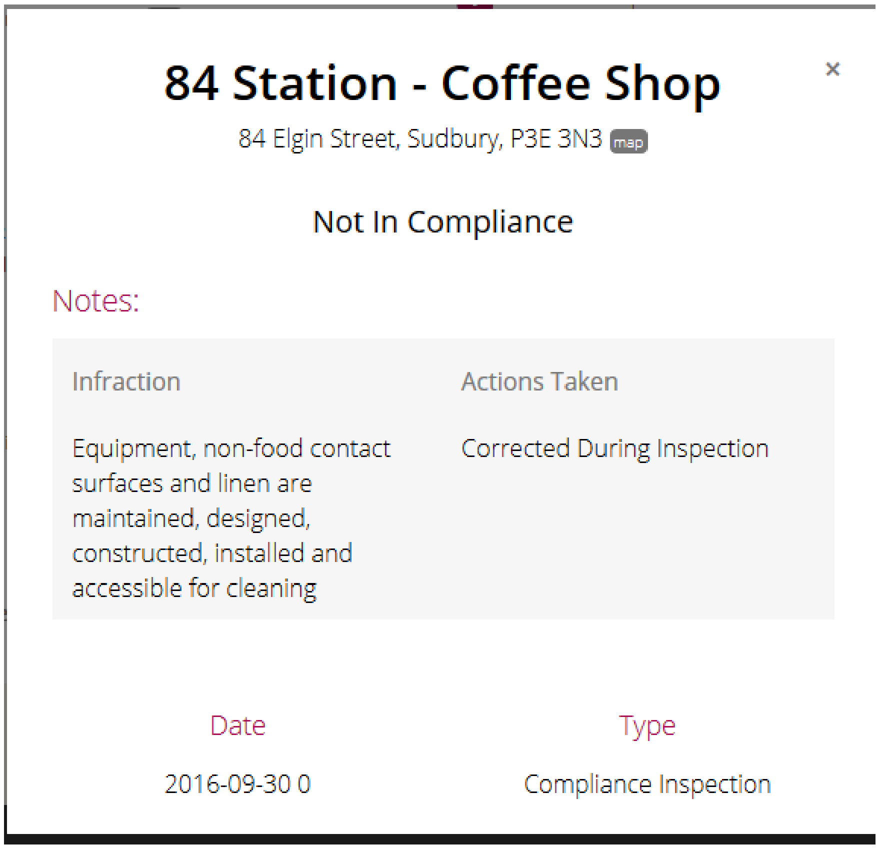
84 Station - Coffee Shop
84 Elgin Street, Sudbury, ON P3E 3N3 map

Not in Compliance

Hide

Sep 30, 2016 Compliance Inspection Not In Compliance Details »

If we click on one of the Details icons, we get details about the specific infractions, if any, along with detail about the establishment:



84 Station - Coffee Shop

84 Elgin Street, Sudbury, P3E 3N3 map

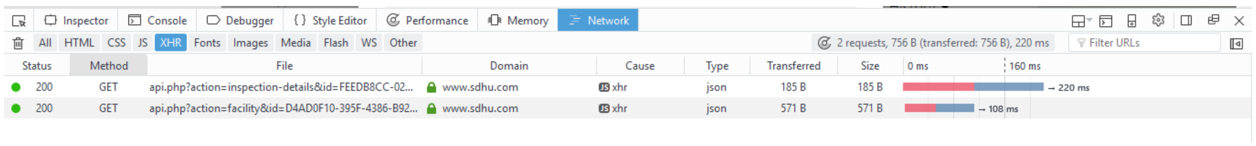
Not In Compliance

Notes:

Infraction	Actions Taken
Equipment, non-food contact surfaces and linen are maintained, designed, constructed, installed and accessible for cleaning	Corrected During Inspection

Date	Type
2016-09-30 0	Compliance Inspection

If we look at the Network tab in Firebug, we can again see what request(s) was/were made to the server to fetch this information.



2 requests, 756 B (transferred: 756 B), 220 ms

Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms	160 ms
200	GET	api.php?action=inspection-details&id=FEEDB8CC-02...	www.sdhu.com	xhr	json	185 B	185 B	→ 220 ms	
200	GET	api.php?action=facility&id=D4AD0F10-395F-4386-B92...	www.sdhu.com	xhr	json	571 B	571 B	→ 108 ms	

This time, we can see there were two GET requests. If we look at each in turn, we see that the first one requests the inspection details, using the unique inspection ID.

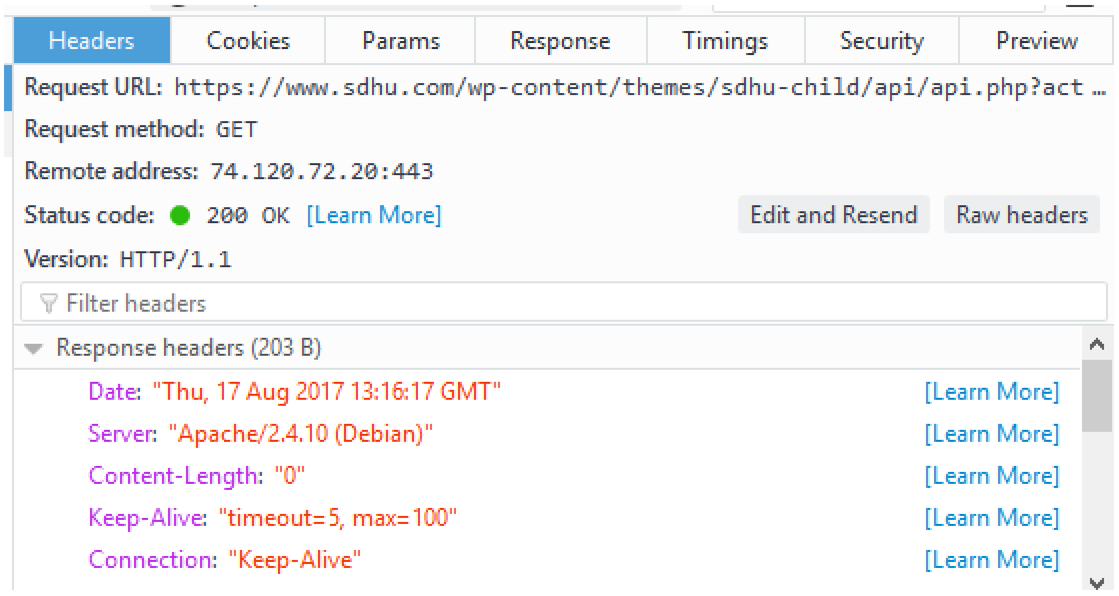
The returned JSON looks like this:

```
[{"Question":"Equipment, non-food contact surfaces and linen are
maintained, designed, constructed, installed and accessible for
cleaning","InCompliance":"Corrected During Inspection"}]
```

This is pretty simple. We get the “question,” which is the aspect the inspection, and “InCompliance,” which is the result of the inspection for that aspect. We can scrape that easily.

If there is more than one issue, then there are more dictionaries in the array.

A close inspection reveals that this information is only generated when there is an issue discovered during the inspection. There is a response from the server, but it is empty of content, when the establishment passed. A look at the headers for the response for a passed inspection shows that indeed, nothing is returned. There is also no JSON.



Request URL: https://www.sdhu.com/wp-content/themes/sdhu-child/api/api.php?act …
Request method: GET
Remote address: 74.120.72.20:443
Status code: ● 200 OK [Learn More]
Version: HTTP/1.1

Response headers (203 B)

Date: "Thu, 17 Aug 2017 13:16:17 GMT" [Learn More]
Server: "Apache/2.4.10 (Debian)" [Learn More]
Content-Length: "0" [Learn More]
Keep-Alive: "timeout=5, max=100" [Learn More]
Connection: "Keep-Alive" [Learn More]

The second request when you click on the details link uses the main facility ID again to request details about the establishment to include in the details popup.

This request is made for both requests with issues and those without. The JSON in the response looks like this:

```
{"FacilityDetailID":"57AF5646-76CD-412A-B165-
D46CAF9C853B","FacilityMasterID":"D4AD0F10-395F-4386-B92A-
98F9B63D0891","FacilityName":"84 Station - Coffee
Shop","FacilityCategory1":"Food,
General","FacilityCategoryStyleID":"D7A1A2AC-5D5C-4E87-88B2-
011F08A93AC5","FacilityCategoryStyle1":"Cocktail Bar \/ Beverage
Room","SiteUnitNumber":null,"SiteFrom":"84","SiteStreet":"Elgin
Street","SiteCity":"Sudbury","SitePostalCode":"P3E
3N3","SiteProvinceCode":"ON","SiteProvince1":"Ontario","SiteTelephone"
:"662-3757","Address":"84 Elgin Street","FacilityNumber":"SUD-160-
0000011"}
```

A close look at the response shows that it actually gives us some more detail about the establishment that was not included on the main page, such as the type of establishment, a general category in which it fits, and a phone number. There is no reason we can't scrape some of that detail into our final results (though as you will see, we will leave that task to you as a challenge).

This examination of all the responses and requests suggests we can scrape the data out into at least three related tables, one to contain the master details about the establishment, the second the overview of each inspection, and the third the details of each inspection for which there was an issue. It makes sense to proceed this way because for any establishment there may be several inspections and for any inspection there may be several issues found. Trying to put everything together into a single flat file table would result in significant duplication, which as we learned in Chapter 5 is something we try to avoid when designing data tables.

Let's try to figure out how we would perform this rather more difficult scrape. Since we've already written the code to scrape the master data, we can simply incorporate that into our new script. That said, we'll need quite a bit of new code.

Conceptualizing our scrape

In Chapter 9, we indicated that it is a good idea to conceptualize what a scrape will look like before writing the code.

Based on the exploration of the site above, this is what our scraper would need to do.

1. Go to the main inspection page and retrieve the array of JSON objects that contains the main details for each establishment and whether it is currently in compliance.
2. Convert the array to a list of dictionaries using the JSON module.
3. For each dictionary:

    a. Write it to the main details CSV file.
    b. Grab the facility master ID
    c. Request the inspection overview for that establishment, from the server, using the facility master ID.
    d. For each inspection in the inspection overview:

        i. Convert the returned array of JSON objects using the JSON module.
        ii. Write the data to the inspection overview CSV file along with the facility master ID.

iii. Grab the unique inspection ID.
iv. For inspections that were not in compliance, request the inspection details using the inspection ID.
v. For each issue in the inspection details:

   1. Convert the returned array of JSON objects using the JSON module
   2. Write the details to the inspection details CSV file, along with the inspection ID.

There are a lot of layers involved, but it is easy enough to see what kind of Python structure will be required. There will be a main request, then for each main item, we will need to iterate through all the inspections, and for each of those inspections, we will need to iterate through each of the details. The code will be a series of for loops, each nested within the next, in a manner that echoes the conceptual structure we have above. There will also have to be a conditional structure, otherwise known as an IF statement, to skip the request for inspection details when an establishment passed.

Let's take our conceptualized scrape, and convert it to some code we can run. The completed code for this tutorial is available as AJAXScape.py on the companion website to *The Data Journalist*. Here, we will walk through the code from top to bottom. Note that it is possible that with a change in the website, the code will no longer work. The tutorials will be updated at least annually.

```
1. import urllib2
2. import json
3. import time
4. import unicodecsv
```

The first part of the new script imports the necessary modules, and is the same as the simpler script that scrapes just the main page, except that we are also now importing the time module, which will allow us to implement something critical in scraping scripts, and that is a delay between requests to the server. If we don't put in a delay, our scraper can go back to the server dozens if not hundreds of times per second, which could bring down some smaller servers and would likely be interpreted as a criminal attempt to attack the server, even by larger entities.

The next section of the script defines two functions. We have not used many functions before in our scripts, so we'll explain what is going on here. These two functions consolidate the opening of the output file and the instantiation of the unicodecsv writer so that when these actions need to be performed repeatedly, we don't have to write out the code each time. As well, if we have to change the code in the future, say to change the location where we are saving the output files, or to change the field delimiter used by the writer, we can make the change in one place. There is a principle in programming that goes by the acronym DRY, which means "don't repeat yourself." These simple functions prevent unnecessary repetition.

To define a function, use the def keyword, followed by the name you are giving to the function and a set of parentheses. Inside the parentheses, you put the arguments for the function, which stand in for values that you will pass into the function at run time. The passed in values are then used within the function, and the return keyword designates what will be passed out of the function after it runs (to run a function, we "call" it from within our program). So you pass in some values, the function does something with them, then the function returns something back to the script that called the function.

As a side note, when you use a method, either one of the built-in methods that come with Python such as the str() function, or a method defined by a module you import, you are actually running a function, just one that was already written for you.

In this case, the function we are writing takes a string representing the name of a file as its only argument. Within the function, that passed in file name is concatenated with the rest of the code used to open a new file for writing. The opened file object is then returned by the function. Note that the body of the function is indented in the same way as the body of other code blocks, such as those within conditional statements or loops. Indenting is Python's way of saying "this code belongs to that construction up above." When you stop indenting, Python assumes the subsequent code is no longer part of the function, conditional statement or loop. If you fail to indent, or indent where you shouldn't, you'll get an error.

```
5. def makeFile(fileName):
6.     theFile = open('C:\Users\Owner\Dropbox\NewDataBook\
       Tutorials\Chapter9\9_9_JavascriptScrapes_AJAX\\' + fileName +
   '.csv', 'wb')
7.     return theFile
```

The next function takes the name of a file-like object as its only argument, and returns an instantiated unicode csvwriter object. When we run it, we'll pass in the file created by the makeFile function.

```
8. def makeWriter(handle):
9.      theWriter = unicodecsv.writer(handle, delimiter=',')

         return theWriter
```

Line 10 is the same as line 4 in our previous script; it uses the urlopen method of urllib2 to fetch the main inspection results. Remember that we are fetching the JSON that populates the page, not the main health unit page, because the site uses AJAX to populate the page with results.

```
10.     mainPage = urllib2.urlopen('https://www.sdhu.com/wp-
   content/themes/sdhu-
   child/api/api.php?action=facilities&filter=inspection-
   results').read()
```

Next, we see the first use of the .sleep() method of the time module. This will insert a wait of 10 seconds after the request to the server. Ten seconds is generous, and should ensure we don't put an onerous load on the Sudbury servers. That said, for much larger sized servers, shorter intervals can probably be used, but never make more than one request a second to a server unless you have permission from the server administrator. Chapter 9 of *The Data Journalist* has a longer discussion of the ethics and legalities of web scraping.

```
11.     time.sleep(10)
```

The next three lines use our makeFile function, defined near the top of the script, to open three CSV files for writing our three tables.

```
12.     output1 = makeFile('SudburyFood5')
13.     output2 = makeFile('SudburyFoodInspecOverviews5')
14.     output3 = makeFile('SudburyFoodInspecDetails5')
```

In line 15, we create a list with the field names for SudburyFood5. These are the same field names we used in our original script.

```
15.     fieldNames1 =
    ['FacilityMasterID','FacilityName','Address','SiteCity','SiteProv
    inceCode','SitePostalCode','InCompliance']
```

The next line of the script echoes our original script as well. We are creating a DictWriter instance for writing the main file. It takes as its arguments the output1 file handle and the fieldnames list we just created. In line 17, we write the fieldnames to the header of the file.

```
16.     writer = unicodecsv.DictWriter(output1,fieldnames =
    fieldNames1)
17.     writer.writeheader()
```

Lines 18 to 19 use our makeWriter function to instantiate CSV writers to write to our other two files, then lines 20 and 21 write the headers for the two files. The reasons for the field names will become apparent later in the script.

```
18.     writer2 = makeWriter(output2)
19.     writer3 = makeWriter(output3)
20.     writer2.writerow(["FacilityMasterID","InspectionID","Inspec
    tionDate","RowStatus","InspectionType","InCompliance"])
```

```
21.     writer3.writerow(["FacilityMasterID","InspectionID","Issue"
   ,"Outcome"])
```

The next four lines are slightly modified lines from our original script. Line 22 uses the JSON module's load method to create a list of Python dictionaries from our downloaded data in mainPage.

We then use the same loop we used in the first script, but in addition to writing the data to the main file, we also use the .get() method to grab the facility master ID from each of the dictionaries.

```
22.     theJSON = json.loads(mainPage)
```

```
23.     for dict in theJSON:
```

```
24.         idValue = dict.get('FacilityMasterID')
25.         writer.writerow(dict)
```

Next, we see, for the first time in our scripts, an error handling routine. If an error occurs while running a script, and you have not done anything to handle that error, the script will terminate and the Python interpreter will report the error. In this case, we could encounter an error when the script tries to access the inspection overview for an individual establishment. The error could be caused by your Wi-Fi connection going down, an outage with the server, or less likely, a bad URL. There could also be an error caused by a problem with the JSON returned by the server, causing the parsing routine that follows to fail.

To handle possible errors, use the try/except clause.

First, enter the keyword try, followed by a colon. As with any other code block, what follows the try keyword is indented one tab stop or four spaces (don't mix tabs and spaces.

Within the try block, place the code you want to run.

Then, at the same indent level as the try clause, write the except clause, again followed by a colon.

In this case, the except clause is actually at the end of the script because the code block we are trying to run encloses everything else that follows, including the nested loop that extracts the inspection detail information. Lines X and Y tell the interpreter that if the block above fails then to put the message for the error into a variable, then print it to the screen. That way, when the script has finished running, you'll be able to review any reported errors to see if you need to change anything in your script.

```
26.     try:
```

Lines 27 and 28 use the urlopen method to fetch the inspection overview information for the current establishment, then wait 10 seconds. The URL comes from the examination using Firebug of the request made to the server when a user clicks on Inspection History. That URL changes with each establishment to include the unique ID for that establishment, so we concatenate the part that doesn't change with the unique ID that we extracted in line X, then apply the .read() method to the result, storing it all in inspecOverview. Notice that in the concatenation of the idValue, we are applying the str() function to ensure that it is a string value and not a number. You can't concatenate a number and a string value.

```
27.         inspecOverview =
   urllib2.urlopen('https://www.sdhu.com/wp-content/themes/sdhu-
   child/api/api.php?action=inspections&id=' + str(idValue)).read()
28.         time.sleep(10)
```

In line 29, we use the .loads() method of the JSON module to convert the data we just retrieved into a list of dictionaries. This is followed by a block in lines 30 to 39 that iterates through the list of dictionaries, and for each iteration creates an empty list called detailList, uses the get() method to grab specific data elements from each dictionary, creates a list called detailList containing the elements just extracted, uses the unicode csvwriter to write that list to the output file, then overwrites the list with an empty list. This last step is an extra precaution to ensure that contents of the list from a previous iteration are never written to a later line in the file. Notice that we have given the iterator variable oDict a name that gives some sense of what it is, in this case a dictionary containing overview information.

```
29.         overviewJSON = json.loads(inspecOverview)
30.         for oDict in overviewJSON:


31.             detailList = []
32.             inspectionID = oDict.get("InspectionID")
33.             inspectionDate = oDict.get("InspectionDate")
34.             rowStatus = oDict.get("RowStatus")
35.             inspectionType = oDict.get("InspectionType")
36.             inCompliance = oDict.get("InCompliance")
37.             detailList =
   [idValue,inspectionID,inspectionDate,rowStatus,inspectionType,inC
   ompliance]
38.             writer2.writerow(detailList)
39.             detailList = []
```

Now that we have written the overview information for this inspection to the overview csv, we need to do the script equivalent of clicking on the Details link on the webpage. But we only want to do it when the inspector found issues. We know that if the establishment passed, there is nothing further to be obtained from the details. We can save a request to the server this way, which with the delays will add up to a lot of saved time over the running time of the script.

Line 40 uses an if expression to test to see if the inCompliance variable, extracted in the last part of the script, points to the string "No." You'll notice that in this comparison, we are using not a single = sign but a double == sign. This is because a single = sign is the assignment operator (see Chapter 9 *of The Data Journalist* if this terminology is unfamiliar to you) and a double == sign is the equality operator. Do the two things on either side of == sign have the same value, in this case the string "No"? If they do, then the actions within the code block that follows will be run. If not, the script will simply ignore everything that is within the if statement block. As you can see below in lines 42 to 51, the entire routine to fetch the details for an inspection is within the if block and will only run if inCompliance equals "No" .

```
40.             if inCompliance == "No":
```

In line 42, we use our now familiar urlopen() method to make a request for data from the Sudbury food inspections site. Again, we are using a URL that we found by using Firebug to see the request made when we clicked on the Details link. Because the request URL has a part that is always the same, and a part that changes, we use concatenation to join the unchanging, base URL, with the inspection ID that we extracted in the previous routine. The inspection ID is needed to get the details for that inspection. We then apply the .read() method to read out the returned JSON data and store it in inspecDetail. After that, we again use time.sleep() method to make sure the script stops executing for 10 seconds after the request to the server. Note that we are once again wrapping our routine for fetching and processing the detail JSON in a try/except statement, so any errors in fetching the data or processing it will be caught rather than bringing our script to a halt.

```
41.             try:
```

```
42.                 inspecDetail = urllib2.urlopen('
   https://www.sdhu.com/wp-content/themes/sdhu-
   child/api/api.php?action=inspection-
   details&id='+str(inspectionID)).read()
43.                 time.sleep(10)
```

The next routine essentially repeats what we did when we extracted the overview information. Again, we use the .loads() method of the JSON module to convert the fetched JSON data into a list of dictionaries. Remember that the fetched JSON is always an array of JSON objects, which is why we end up with a list of dictionaries. We then iterate through the dictionaries stored in the list, and from each one extract the data we want, before writing these to a list along with the master facility ID and the inspection ID, and writing the list to the CSV file.

```
                    detailJSON = json.loads(inspecDetail)


                    for dDict in detailJSON:
                        fineDetailList=[]
                        question = dDict.get('Question')
                        inCompliance = dDict.get('InCompliance')
                       fineDetailList = [idValue,inspectionID,question,inCompliance]
                       writer3.writerow(fineDetailList)


                       fineDetailList = []


               except Exception, e:
                   print e


        except Exception, e:
            print e
```

At the conclusion of this innermost loop, the script then goes back to the next inspection overview dictionary, if there is one, scraping that data before again testing to see if the inspection failed, and if it did, running the innermost loop again. Once all of the overview items have been scraped, the script goes back to the main loop and starts again with another establishment, fetching its overview data, and so on. In this way, all of the data at all three levels is collected and written to the three output files. Any errors encountered along the way are printed to the screen, so we know what happened. If we were so inclined, we could change the script to write those error lines to the appropriate output file, so we would have a permanent record of where the errors occurred.

More functionality could be added to the script. For example, you could decide to grab some of the additional information about the establishments that is fetched by the second request to the server when a user clicks on the Details link. You could also have the script skip writing any overview or detail data for inspections previously scraped. To do this, the script would need to check the previously written overview file to see if an inspection id was already present, and if it was, move on to the next one. The only caution with this approach is it is possible inspection results could be revised to correct errors, and you would never get the corrections. Of course, you could also get the script to check not only to see if the inspection id was already present, but whether the recorded results were the same.

All of which is to say, you have a great deal of power to write your scripts as you want, and include functionality as you please. This is one of the reasons why writing your own scripts is so often preferable to using off-the-shelf scraping programs.