Data Wrangling with MongoDB

Lesson 2 Notes

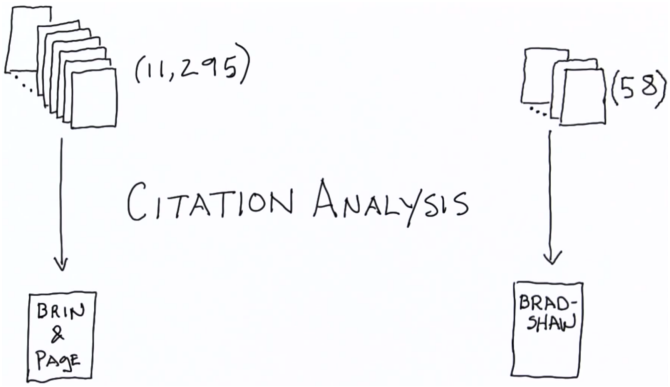
Data In More Complex Formats

Welcome to Lesson 2

In the last lesson, we looked at extracting data from some very common formats, CSV, Excel and JSON. In this lesson, we're going to continue with that theme, moving into extracting data from XML. We'll even look at scraping data as HTML from webpages.

Intro to XML

Great. Let's dive into XML. Over the course of my career, I've used XML in a number of different data science projects. One of these was working with a large collection of research articles. So, just as an example, here's the original Google paper from Brin and Page when they were still grad students at Stanford. Now, what I was doing in this project is what's known as citation analysis. In citation analysis, What we're doing is comparing the relative importance of papers based on how many other research articles cite them. So for example you could compare the Google paper with some of my work. Which receives a much more modest number of citations compared to the 11,000 that Brin and Page's paper got.

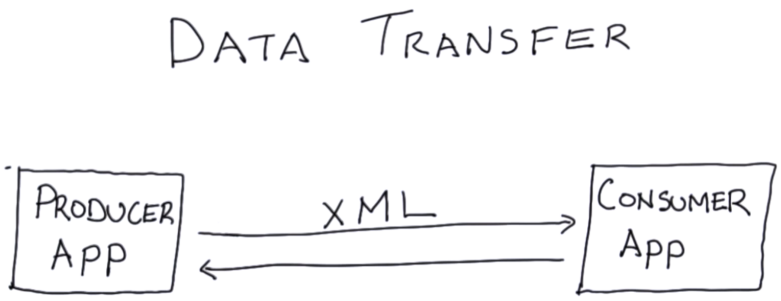


Now when I was doing my work most of the data that I used was not publicly available. But nowadays the same type of data is available and it's encoded as XML. There are quite a few open access publishers, like bio-med central. These publishers produce every article they publish both in a print form like this, and in .xml. Now, in order to do something like citation analysis, what we need to do is access the bibliography for each article. So what I want to look at as an example, is how easy it is when you have your data encoded as .xml, to pull out that type of data, and use it programmatically. So let's take a look at the references for this paper. Here are all of the other papers that this particular research article cites, now let's take a look at the XML version of this paper. Here is that exact same paper only here, instead

of being designed for reading, it is instead encoded as data, let's jump down to the bibliography for this paper. And here it is, the very beginning of the bibliography. If we take a look at the print version of the article again, we can see that it does, in fact, align with what we're seeing here. So, this type of use of XML is very much what the designers of XML had in mind, where you have documents that have lots of text, but text. That you want to encode so that portions of it, at least, can be used programatically, like we might want to do with a bibliography of a research article. Or the author list and other data that occurs throughout a document like this.

XML Design Principles

XML was designed with a number of goals in mind. One of the most important for purposes of this class is, it was designed to provide data transfer that's platform independent. What does that mean? Well the idea here is that you can have a Producer App. That's written in any programming language on any operating system and any type of hardware. And the consumer app implemented in any other programming language operating system or hardware. There is no binding between how the consumer app or the Producer app is implemented, because they both agree to speak XML to one another. And of course in addition to consuming XML from the Producer app, the consumer, would also write XML to the Producer app.



Another important goal for XML is that it would be easy to write programs to read and write XML. The designers also wanted a data format that could be validated. So in XML, we write a specification for a particular type of document. And then any specific examples of that document that are produced can be validated against that specification. So BioMed Central has a specification for the research article format, and any articles that are produced are validated against that format to ensure They adhere to the rules for that data model.
XML is designed to be human readable, and as we saw in the example, we can get a pretty good idea of what information is contained with in an XML encoding just by looking at it.
And finally, XML is designed to support a wide variety of applications. We've seen one application of XML, we're going to take a look at several others. That essentially span a number of different ways in which XML can be applied to exchanging data between applications. If you're interested in more information about this data format, I encourage you to take a look at the W3C site.

We're going to talk a little bit about what having a standard means. One of the most important benefits of there being an XML standard, is that we have robust parsers in most programming

languages, Python included. What this means for us as data scientists, is that we get to focus on our own applications. We don't have to worry about writing parsers for some ad hoc data format. Previously each messaging system had its own format and all were different which made the type of messaging that we do now very messy, complex and expensive to do. If everyone uses the same syntax, it makes writing these systems a lot faster. And much more reliable.

Another advantage for XML is that it's free. Now, that's free as in beer but also free from legal encumbrances. It's not a format that any company owns and may change out from under us. XML information can be manipulated programmatically. So we can build databases to support specific types of queries. Or, we can piece together data from different sources or take it apart to be reused in different ways.

XML documents can also be reliably converted into other formats with no loss of information. XML lets you separate form, or appearance, from content. So, your XML file contains your document information, all of your text and data and identifies its structure. Formatting and other processing needs are identified separately in a Stylesheet or processing system. In the BioMed Central example, it is actually the XML that is transformed into HTML for rendering on the website or into PDF for download. Using a Stylesheet. And stylesheet processing system. The two are combined in output time to apply the required formatting to the text of data identified by its structure. This structure might define location, position, order or any other aspects of the data.

XML in Practice

We can represent most things by a list of properties and their values. Think about how you would represent you or your cell phone, the email in your inbox. Maybe even events, such as the last party you attended. Mostly a list of key value pairs, right. Key value pairs or dictionaries as they're called in Python, make a lot of sense as a way to represent data. Primarily because it's so natural to represent so much of the data we care about this way. Let's think about the “you” example for a minute. In my case, some of the features might be: Eyes: brown. Hair: brown. Height: 6'2. Weight: I'll leave that as an exercise for the viewer.

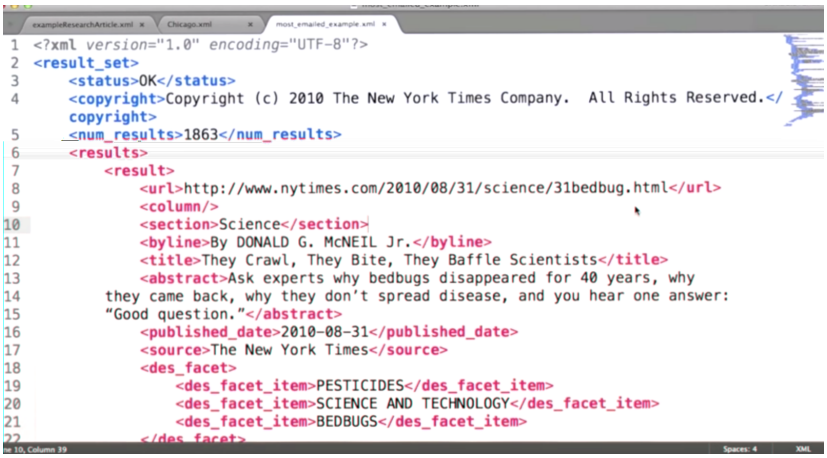
Note that name pairs are very different from documents. With the document, we have sections, containing subsections, containing paragraphs, containing sentences. This type of nesting or tree structure is exactly the type of structure that XML was designed to work with. In practice you'll find that XML is used for many purposes, even those for which it's probably not the best choice, and others for which one could argue it's not appropriate. In order to ensure you're ready to wrangle XML in all its forms, we'll give you some practice working with applications of XML, to both documents and key

value data.

Fundamentals of XML

In case you're not terribly familiar with XML, let's spend a few minutes talking syntax. Even if you are familiar, it might make sense to follow along with this little review.
So in XML, elements are the basic building blocks of an XML document. Now an XML element is composed of an open tag and a closed tag.

This is some data drawn from the New York Times developer API. I encourage you to have a look at this site. We are going to look at some data from the most popular API. These are for example articles that are most frequently emailed among readers of the New York Times.



Okay, so let's look at a couple of examples here. So, the first thing that we might notice about this particular document is that we have some tags for num results or some elements that have to do with the number of results. So, this is actually result set from having done a query to the most popular API and we've got a, an element that tells us how many results were identified by our query. And then the list of results follows. Now this happens to be a single result here. And we can see that this result begins right here with this open tag and closes right here with this close tag. Okay.

Now just as a couple of other examples of the data within this particular result, we can take a look at the byline, note that it's got a close tag, as well. And some of the other elements, here if you note the title for example, this happens to be an article about bedbugs. Okay. So, this provides an example using some really nicely named tags. We know what these mean. Now, there's another aspect of XML that we need to concern ourselves with especially given some of the exercises that we're going to have later on. And those have to do with attributes for XML elements. Now, this document provides a number of very nice examples of elements in XML. But what we don't have here are any examples of attributes for any of these elements being used.

So what I'd like to do here is talk about essentially the two types of data that we're going to look at that have been encoded in XML. One is this more documented oriented type of XML, which is originally the type of data that XML was designed to encode. But then we can also take a look at something like this.

Okay, now this is actual data from the Open Street Map project. This is a pretty close zoomed in view from Open Street Map of West Belmont Avenue. Particularly the 1000 block. And you can see right here, there's a Giardano's Restaurant here. Giardano's is a famous pizza chain in Chicago. So, this is data that is essentially from a layer on top of that particular map. This is data that is human created. So, users of Open Street Map have actually added this data on top of the map data. And what I want to point out here is that this is very much not document oriented. This is just data. Okay? And a lot of times you see HTML used in this way, you'll see that attributes are heavily used. So in this particular example, this is the node that represents the Giordano's restaurant. We can see that there is a number of attributes specified for this particular element. Common among them are the latitude and longitude attributes that this particular annotation applies to. So, essentially what this data element provides is a mapping from geographic coordinates to more common street address coordinates. Okay? So this is a good example of attributes in XML and there's one other thing that I want to point out here. And that is this type of tag here. Now in this particular data they're doing something that I probably wouldn't do, but it is the type of thing that you're going to see as a data scientist and likely already have. Essentially, they've just got a bunch of key value pairs that are encoded in something called a tag element. And, in this case, none of these tag elements have a close tag. Instead, they use this special xml syntax where you can simply create what are called empty tags, that is tags that don't have any content. All of the data for this type of tag is contained directly within its attributes.

So the most emailed example here provides us a nice example of document oriented XML with lots of content inside the elements. And this particular example from the OpenStreetMap project provides us with other end of this spectrum which is very data oriented XML where all or almost all of the data is contained within attributes for the individual elements and in this types of cases, you often have mostly or at least many empty elements within the XML data that you are looking at.

Parsing XML

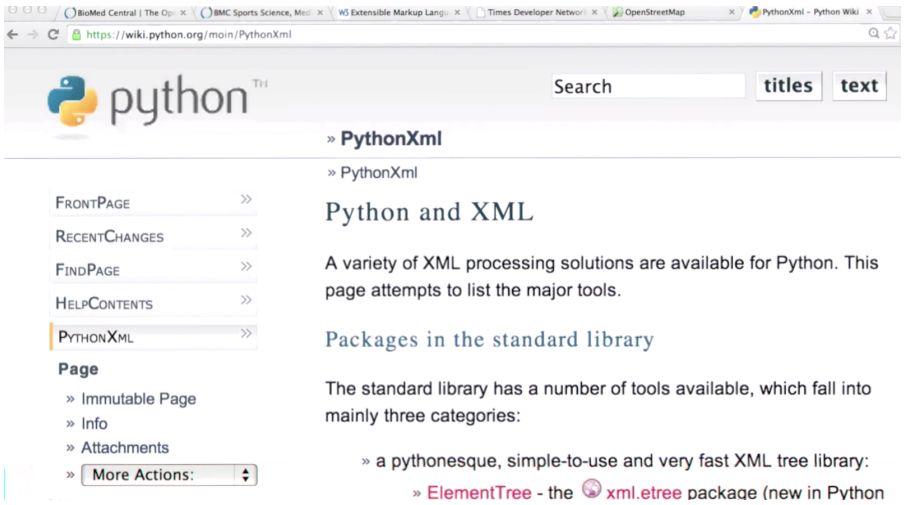
JSON maps almost perfectly onto Python dictionaries and arrays. Parsing XML is a lot more complex, so there are several options to choose from. We'll look at some of the most common. I also recommend that you have a look at the Python XML page to see what your options are.

Let's first look at parsing XML into a document tree. The fundamental difference between this and event driven or, what we might call sax parsing of XML is that, here, we're going to read the entire XML tree into memory. We're going to use the XML e tree Element tree module, which is part of the Python standard library. With element tree we can parse data in a couple of different ways. We can parse it by file name. We can also use it from a string method of the element tree module to parse XML stored as a string value in our program. As we think about parsing XML, using element trees, it makes most sense to work with documented oriented XML data. So let's return to the research article example.

This data set from Bio med central is both an excellent example of XML used to encode documents and one type of data that data scientist, specifically text minded folks, are likely to use in their work. In extracting the data we need from a tree, we're mostly going to be working with element objects. Elements permit us to iterate over their children. For example here, we've parsed our research article into a tree and then from this tree, we're getting the root element. Then we're going to iterate over the children of that root element, and in this case use the tag attribute to print out the tag name. Of each child element. So let's see what this looks like when we run it. Okay, now let's take a look at that xml document again to see where these values come from. Okay, and here we can see that this is the root element, and we are getting out all of the elements that are nested immediately within that element or the direct children of that element, ui, ji, and fm. And we can see that those are coming out here. Now, if we want to extract the title of an article we can see that its found in the bibliography section of the front matter section. Let's see how me might extract that. Element tree supports basic XPath expressions. Because, in data wrangling we're often pulling most of the data out of an XML document, this level of support is going to be sufficient for many if not most of our needs. Returning to the problem of extracting the title, we can do that as follows. So, here, you can see that I'm using the find method on the root element. And I'm using an XPath expression to specify where I expect to find a title element. That simply means, start at the current element. And then work your way down into the fm element and it's child the bibl or bibliography element and then finally into the title element itself. Note that this is very much like the way that you specify a path in a URL or in many file systems. Now it turns out that in this data, many of the text elements are actually wrapped in paragraph tags. And we can see that that's the case actually for the title here. So what I'm going to do here is with the title element that I got by using this xpath

expression in this find method, I'm simply going to iterate over all of the children of title. Very similar syntax to what we saw before when iterating over all of the children of root. And for each child, I'm simply going to take the text of that child and concatenate it on to my title text.

Now, so far for elements we've seen two different attributes. The first one was tag and now we're seeing the text attribute. There are a number of attributes for elements that are useful. These are two of the most common that you'll end up using. Okay, and then here we're simply going to print out the title. So, let's go ahead and run this and then we'll return and take a look at what this piece of code is doing for us. Okay, and here we can see that the title is actually selected and printed out properly. Here we're actually printing out the author email addresses. Let's take a look at how we did that in the code. Here we're going to use the find all method. Instead of the find method, find all with return all of the elements that match this XPath expression. And so, we're simply going to iterate over each one of those elements and then for each one of those elements, we're going to do a Find in order to locate the email tag. So, let's go again then and take a look at the data. Okay, so my XPath expression gets me this far, down here into the author group. And then for each one of these authors, I'm doing a find for an email element, and I'm selecting the text of that element, here, and simply printing it out. So again, this selects all of the individual authors found in the author group that's part of the bibliography section in the front matter.



Programming Quiz: Extracting Data

It's more likely that we would extract all the author data from documents such as these. How would we do that? Assuming this document was illustrative of the entire biomed central schema, write a program that will use the element tree module to extract all the data for a given research article. The data for each author should be stored in a python dictionary, and all the dictionaries should be stored in a list. By all the author data, I mean all of the values stored here for each one of the author tags in the author group that is part of the front matter section. Now this is a lot like what we did with CSV files, building python dictionaries to represent the data we're extracting from, in this case XML files. Now, one thing to keep in mind as you're doing this is that this is a lot like what we did with CSV data, building python dictionaries to represent the data we are extracting from. In this case XML files. One

final point here is that you may ignore the INSR tags, simply leave them out of your data processing

Programming Quiz: Handling Attributes

As we saw with the OpenStreetMap data, XML tags often contain attribute name value pairs. One example of attributes in this data set is the iid attribute of this institutional relationship tag, or insr tag. The iid value here actually defines a relationship between an author and institutions with which they are affiliated. Institutions appear later in this institution group, here, as individual elements of their own. Now in the last exercise we ignored these insr tags. Let's address them here. We're going to update our solution to the last exercise to use the data in these insr tags. See the code I've provided. However, rather than an insr field in your author dictionaries, store an iid field for each author. Note that some authors will have multiple insr tags. The value for the iid field then for authors needs to be an array composed of all the iids for a given author. For each author, create an iid field. Note that some authors are actually affiliated with multiple institutions, so the value for your iid fields needs to be an array.

Intro to Screen Scraping

For some analysis tasks, it's not easy to get the data we need. The data's there somewhere, but not in a form that we can simply download. In these situations, the data is usually found in bits and pieces that you need to wrangle together. Sometimes all the pieces are on a single website, and sometimes you need to bring them together from multiple sources. As data wranglers, we need to be nimble and creative in gathering the data we need.

Introducing the Working Example

Let's look at an example. A lot of times where you find yourself in a situation, where you need to get creative, you're probably going to need to scrape data from a website. In my experience, this is most often the case when the data is provided by a government agency of some kind, or maybe a small grassroots organization.

In this case, the example I would like to look at is from the US bureau of transportation statistics. So, imagine we have a need for among other things to get statistics on how many flights individual carriers have to and from the airports in the United States. Okay, now this particular page provides us with a way of getting at that type of flight data. That is, how many flights say Virgin Atlantic has

into and out of Boston Logan Airport. But in order to get it we have to go through a lot of work and a lot of manual effort if we want to pull all the data for all the carriers. So this is a great example of a situation where we would problematically deal with this problem. Okay now I've got this zoomed in big enough so that we can see what's on the page and as a result the layer is falling apart a little bit so bear with that. So, let's take a look at how we would use this site. Now right now, this is providing for us, the number of domestic and international flights for all carriers, US and foreign, with a destination of any airport. So, if I want to get just the data for Virgin America, and all of the flights for Virgin into Logan Airport in Boston, I can make those selections and click submit. And here we have just that data.

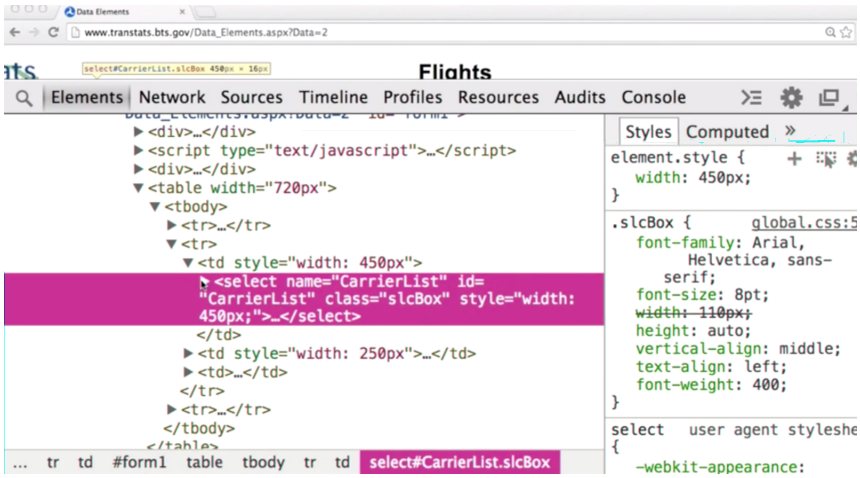
Example Details

Okay so let's tease out some of the details here. What I'd like to do is assemble a complete record for carrier and airport for all of the data maintained in this database. Okay. So what' I'm talking about here is doing the necessary queries against this site. In this case I would need to do two queries for each carrier and airport pair. And end up with a table of data that looks something like this. Now I may decide to store it as a table or JSON. The format doesn't matter here. So much is the data wrangling from the site. Okay. So, for every carrier and airport in a given month, I want to know both how many departures and arrivals there were for that airline.

Wheres the Data

Okay, so let's do a little exploring and think about the mechanics of this problem. The first thing I'd like to know, is how do I go about requesting the data I want from this website? Again, I'm thinking about how I get to a point where I can actually do this programmatically. So most web browsers give you the ability to actually inspect individual elements of a webpage, so in this case, let's just take a look at that selector. And if we do that, we can see all of the options right here in the HTML for the page. And based on our understanding of HTML, we know that it's these values here that we would need to submit as part of our post request. Alright, the options for airports look very similar. So instead of looking at that, let's take a look at The data that's here. Now, a couple things that I want to point out here, before we actually look at the HTML again. One is that for any given airport, this is reporting both domestic and international flights for that airport. Now in my case, I'm not actually interested in that distinction. I simply want to know arrivals and departures. So where does value in both columns, for a given month? I'm

simply going to add them together. So this is one place that I'm doing a little bit of reshaping of the data. It's also the case that I've got these totals here in this columns, and then at the end of the rows for any given year. I'm simply going to ignore those. Again, a little bit of reshaping. Okay. So let's take a look at these elements. Now as you might expect, these are laid out in a table, and if I scroll to the top, I can actually see there's a class attribute for this particular table here. And this is going to help me when it comes time to actually parse this HTML in order to extract the data. So we've looked at, both how to go about getting the values we're going to need to submit in a post request in order to get the data we need. And then we've looked at once that data is presented to us, or in this case, to our program that's going to be accessing the site. How do we go about finding that data and pulling that data out of the HTML? Or at least where is it located in the HTML file?

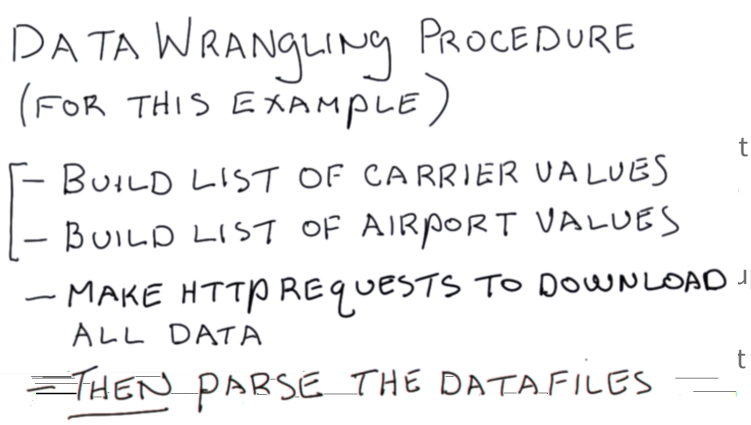


Procedure

So, let's talk a little bit about our procedure here. The first thing we want to do is build a list of all carrier values. We could do that by hand, might actually be a little easier to do it that way just by looking at the HTML. We then need to build a list of airport values. Now there are a lot of values here. So what we probably want to do is actually write a little script that will actually pull those out. Okay. So all pages are going to have exactly the same list for both of these. So we can just use the browser to download an example page and pull those values out. Next, what we need to do is make HTTP requests to download all the data. I'll talk about why we want to download it all in just a minute. Then what we want to do is parse the data files. The reason why we want to do it this way, is because, in building our parser we want to make sure we're working with data that isn't going to change. And after the fact, once we do a little bit of data cleaning, we may discover that the reason why we've got some dirty data is actually because we have a bug in our parser. Much easier to figure out where that bug is if we've still got the original data we were using to parse. I should also point out that it really doesn't make sense to download the data over and over again as we're figuring out how to parse it. Something else you

might want to keep in mind is that for years prior to the current year the data isn't going to change, so there's no reason to retrieve it more than once. So this is actually a bit of a best practice. When you've got a situation like the one we have here and when you've got a scraping task it's often going to look something like this. You really want to grab all the data you need first and then do your scraping into separate process. So what we have for this particular problem is essentially three different steps. We first have to build all the values we're going to use to make HTTP request. We then need to make all the HTTP request, and download the data we need. And then finally, we're going to parse the data we want out of those data files, shaping it into the particular pieces of data, the particular items that we want to use.

Extracting Entities



Okay, so let's get started on actually solving this problem. We'v t the complete list of carriers and the co here is downloaded this page and these lists. So let's take a look at sc p. In particular, we're using Beautiful own machine using any one of a nu t already have BeautifulSoup install BeautifulSoup gives us the ability to do is parse the document tree, the HTML document tree in this case. And, it's similar to elementary for parsing XML. Okay. So let me walk you through how this code works. So what I've done is does the bulk of the work. I'm calling options and note that what I'm calling it with is this soup value. And soup is what I get in response to having called beautiful soup on the local file that is a saved copy Of this one. In addition, I'm passing this string Carrier List. Now what BeautifulSoup passed me back was the top level element for this particular HTML document. And then what I can do is I can use the find method here. The meaning of this is find me the the first descendant element in this document tree where the ID attribute for the element is CarrierList. Okay? So let's go look at the HTML, and see why we wrote it that way. Okay? Again, I can get to the HTML by using the Inspect Element feature of Google Chrome. Okay. So here's the CarrierList, and note that the ID attribute for this particular element here is CarrierList. Okay, so CarrierList, then, will have the HTML element that we just looked at as its value. Then what I'm going to do is call Find all on CarrierList. Find all does something very similar to what find does. It finds the descendants that are option tags or option elements. Now unlike find, find all will actually find all of the sentence instead of just the first one. Okay, so here what's going to happen is I'll just iterate through them then, and I'm going to just build up a list of all of the values. Now what I'm actually doing here Is finding each one of these in turn and I'm pulling out the value of their value attribute. Because, it's this value that I need to pass in any subsequent http requests for data having to do with, say, air Tran or Alaska airlines. So that's how we get the carrier data we need. I use a very similar process, in fact it's an identical process, for getting the full list of airports.

Beginning to Build Our http Requests

Okay so now let's run this code to see what the output looks like. Okay, so all this does is print out a complete listing of all the airport codes and above this, all of the carrier codes. Now it's these values then, that we're going to use in order to mine the data we need from this particular data source. So if we're going to do that, we need to have a good understanding of exactly how requests need to be formulated to this website. Okay, let's take a closer look then. If we scroll up a little bit, we can see the form that's used to submit any requests based on these selectors and the submit button. Now any time you are doing a scraping task like this, you absolutely have to understand exactly how the site expects requests. Some sites are pickier than others. Some sites have more security procedures built in or more constraints that you're going to have to deal with. So our first step is figuring out what URL we have to access, and what HTTP method we need to use. So the HTTP method is post and the URL we need to access is this data elements, ASPX URL, passing it to parameter data equals 2. Now, it turns out that that is exactly this URL here. So submitting this particular form simply executes a request against exactly the same URL.

More Work Than We Thought

In order to mine this site for the data we need, we need to look at how we programmatically construct requests in order to pull each page of data that we're going to need. Remember each time through, we're going to be passing a carrier value and an airport value. So the best way, I think, to do this is to use the browser to tell us how it makes requests to the site. Because the browser is obviously successfully making those requests and receiving the results. Now to do that, We can take a look at the network tab. Now I've just opened this so at this point there are no network requests captured. In order to get those what we're going to need to do is submit our request again. Now its possible when you do that if you've had the page open for a while that you'll actually get an error page instead if that's the case just copy this into a new tab and look it as if you're loading it for the first time. Simply trying to reload the page won't work for you. Alright, so let's scroll to the top here. And we can see right here that a post request was submitted. If we click on that, that will allow us to actually take a look at the HTTP that was sent across, the request itself, as it's encoded as HTTP, in order to retrieve this page full of data. This is the same page we've been looking at all along. Now, what I'm interested in is exactly what data is submitted for this particular form. So going back to the Network tab, what I want to do is scroll down until I find Form Data, and right away I realize that we have more work to do. Than we might have originally expected. This is a perfect example of why we want to use the browser to figure out how to construct our requests. To this point, we've been operating under the admittedly naive assumption that all we were going to need to submit were the carrier and airport as the form data in order to fetch each page we're interested in. But we can see right here that there are at least three more fields that we're going to need to submit, and it's not clear at all to us at this point

where this value comes from. So that's more work to do for us. But this is exactly the type of situation that this particular class is about. In some sense, we're essentially bending the web to our will. This is where we're really getting into the meat of data wrangling.

Programming Quiz: How Many Form Elements

Okay so let's do a quiz. What I'd like you to do, is use the developer tools in whatever browser you're using, to figure out exactly how many parameters we're going to need to pass as form data, in order to successfully execute requests against this website. You can do that, by following the steps I followed to get here, and scrolling down further, to see all of the form data.

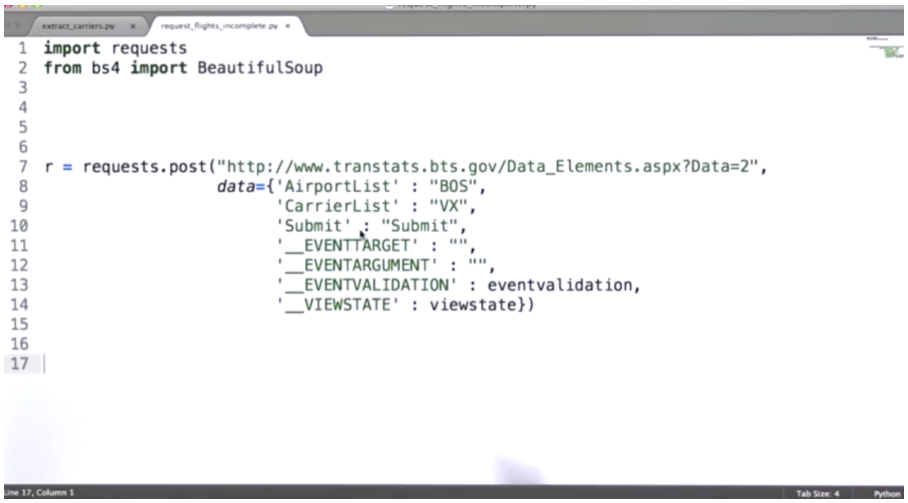
Answer:

Okay. And the solution for this quiz is if you look through all of the form data here, you'll see one, two, three, four, five, six, seven, including the submit button. So, we need to pass seven parameters along as part of our post request.

Using Beautiful Soup

Okay, we're in the home stretch here. Let's figure out how to build the HTTP request such that we can pass all seven of these form elements. Now, the question is where do these elements come from? In most cases when you see form elements showing up that aren't obviously part of the form itself, and by that I mean aren't part of the user interface. Now what's gotta be going on here is that we have hidden form elements. So, let's figure out where they're at. So, here's the start of the form. If, we look down through, I don't obviously see any other elements. So, I bet it's in some of this stuff that's zipped up here. So, let's take a look. And, sure enough, right here we find three hidden form elements. So, now we've accounted for six of the seven elements we've got to pass along, the Carrier, the Airport, the Submit button and then these three. And we can see that right here, Viewstate okay, is one of those with a very long value. If I open this up, I see that here we have one of those other values. The other one in fact, that makes scrolling through all the form data in the network tab such a long and tedious process. Okay, so

now we've identified where all seven of the form fields are. The question then becomes, how do we build our HTTP requests, such that this data is included? Because that's really what we have to do here. In addition to the Carrier, the Airport, and the Submit button, we've gotta send these along as well. So in this exercise what I'd like you to do is write some code here that uses the Beautiful Soup Module, and assigns values to Event Validation and View State such that they have the correct values. The values that this data source is expecting to see come across. In a little bit, we'll talk about why these values are necessary. For right now, let's just focus on building the request properly.



Broken Requests

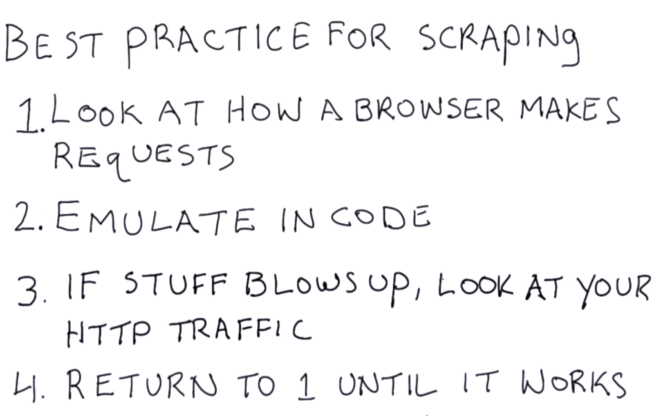
Okay, so now that we've used beautiful soup to parse this HTML, so that we can extract the values we need to pass along in our post request, let's talk about why these values are here. Well, as many websites do, this particular data source uses these values to validate each request that comes in. And if you're really interested in specifically how, for example, View State is used, I'd encourage you to look in the ASP documentation, to see how those types of servers use that particular type of value. Okay, so just in case it's not clear. What we have to do first is submit an initial request and then use the response that we get back, specifically the values for view state and event validation in order to make any subsequent poster request. So let's take a look at the code. Now, one thing I want to point out, something that I didn't mention last time is that here, we've actually hard coded in the value for airport less and carrier list. But of course, what we'd be doing is actually looping through all possible combinations of values in our airport list and carrier list, from the data that we extracted using this code. Okay, so in this code, we're making an initial get request, parsing the response to get the viewstate and eventvalidation, and then submitting a subsequent post request. Let's run it! Okay. Here we're writing this out to a file called virgin_and_logan_airport.html. Let's go take a look at that file. Here it is. And of course we're going to lose style information because relative references will all be broken. we're not actually pulling the CSS or javascript files that are referenced in this HTML file. So, instead of the data that we wanted, instead we're getting a syntax error. So we need to figure out what's going on there.

Best Practices for Scraping

So I mentioned earlier, it's this kind of thing that can really make you want to tear your hair out. As you can see, I've been in lots of these situations. Okay, so we're facing a challenging situation, in that we've got to do some scraping from a site that's really making us work for it. Let's talk about a best practice for screen scraping. So the first thing we want to do here is as I mentioned is, look at how the browser itself makes requests. And with the developer tools, we have quite a few ways in which we can do that. We can also look at the wire traffic if we really needed to using something like wire sharp. Then we want to emulate that in our code. Now if everything blows up, we're going to need to take a look at our HTTP traffic in some way. And then just return to one and work through this again until we get it right. If we follow this process, we have a pretty effective strategy, for dealing with any problems we have making requests from sites we'd like to scrape.

Scraping Solution

Okay, so let's do this. We've taken a look at our HTTP traffic from our client application. Let's go back to the browser and see what it's doing differently. We going to go look at the network request again and if I scroll up, there's a cookie here, there's some session data that is being maintained by the browser and passed along. That's potentially at least part of the problem, so. What we can do then is actually maintain session state in our code. So what I've done here is modified what we looked at before. And in this case, we're going to use a session object instead of just a request object to do both our get and our post. We'll be using the same session object, so any session information that we get back from this first request, will be maintained and passed along when we make this request. Alright, let's run this. Let me go back to the browser and see if we've got a correct response or a response that gives us back the data we're hoping to get. And now we could see that the data we get, is actually what we were expecting. So, note that this is actually a local file and is in fact the local file that we were writing out here. And we've got the right settings here, what we would have expected had we actually submitted this through the browser. And, we're pulling back the data that we hoped to get. Still missing the CSS files, but we don't care all we care about is this data right here. Whoo hoo! We did it!



Wrap-Up

The last step of course would be to put all the pieces together. So, we would take our carrier and airport lists and run a number of queries to download all of the data. We then want to write a parser that will allow us to extract this data, and put it into the tables we would like to build for flight information, for all airline and airport combinations. This is a somewhat larger process than it makes sense to go through in this lesson, but you'll find some fun work to do built around this example in the problem set.