Week 7
Video Transcripts

Video 1 (12:24): Getting data: part 1

Today, we're going to look at how we can get data, because you can't do data analysis without data. And getting data can be a very varied process, depending on where it's coming from. So, the basic problem is very simple. Data exists in many different sources and different formats, and we need to actually be able to figure out what source we're using, what format we're using, and depending on the source and the format, we might need to do something different when we want to recover the data and use it in analysis. So, let's take a look at what the basic formats are. There are many different formats, but generally, you will see local data, which is data that 'son your machine itself, and that typically comes in some kind of text file, either a CSV or maybe a PDF or maybe an XLS. Each of these has their own issues. For example, a PDF file is actually an image of the text of the data, and extracting data from that can be quite complicated. However, luckily for us, there are Python libraries that deal with all this stuff. So, we have the Python library for CSV files, PDF files, as well as XLS files. We will look at, a little bit later–we will look at CSV files, which are really, really simple to deal with, but for today, we're going to focus more on web data. Web data is any data that comes from the Internet. And typically, you will go toe server somewhere and pickup data from there, and the data that comes back is going to be either in JSON or XML or HTML format, and we have to deal with all three formats when we're recovering data from the web.
There are also database servers, and oftentimes, particularly when you work for an organization, you're going to work with databases sitting on a server, and typically, the servers are relational, and these two servers, MySQL and PostGres are relational servers, and we will look at that, too, but not today. Or they may be what are called on-relational No SQL databases, and we'll look at one of them Mongo DB, but there are many other databases like that, as well. However, they all have similar characteristics. So, we should be all right study in those. So today our goal–today and next week–our goal is web data, which is where the vast majority of data that is available–resides these days. It's all in the Internet. So, the first thing about the web is that increasingly, most webservers use what are called Restful web services. REST stands for Representational state transfer, and the idea is that you have a standard mechanism for getting data from a webpage. And typically, in a Restful web service what you end up with is a website for, let's say, you are here, you're a client, and then, there's a server over here. The server can have multiple pages. So, let's say page 1, page 2, page 3, page 4, and these are all interconnected in the sense that you go to page 1.
That links you to page 2, which links you to page 3, which links you to page 4.So, that's page 1, 2, 3, and that page 4.And maybe page 1 is also linked to page 4, etc. So, you can think of a server as actually being a network of web pages, and to get data from the server, you use various HTTP commands like GET and POST and, you know, various other stuff. So, the idea is with think kind of a standardized interface, when you want to write an application that accesses data from the web, you know that they're going to be using the same basic setoff commands, and you can use that to your advantage, so to speak. So, just to summarize, Restful web services deliver resources to the client, and there sources are from web pages or could be a JSON file or an image or whatever, and everything is associated with the URL. Whether it's an image or an HTML page or a JSON file, there's a URL and HTTP method. HTTP stands for the Hypertext Transfer Protocol that's associated with that image, JSON, or HTML, and you can use a URL to directly access data from the web server.

So, any server that conforms to the REST standards is called Restful, and typically, you can expect almost every server these days to be Restful anyways. So, let's take an example. So, for example, if you go on your browsers right now and type www.epicurious.com,that's the website for Epicurious, which is a website that delivers recipes based on keywords or stuff that you provide–search terms that you provide to it. So, you can type any search term you want. I typed tofu chili, for example, but you could do whatever you feel like, whatever kind of food you like to eat. So, you type that, and what happens is you get–you type that in the search box. So, that's the search box. So, you can type here tofu chili. You type that, and then, you click the Search button, and once you click that, what happens is that you get back a new URL that looks something like–that comes up on the page that looks like this. And the URL for that page is this one.

Notice something about this URL. It contains www.epicurious.com,which is the same thing that we had over here. Then it had the word Search, which corresponds to the fact that we were searching over here, right? And then, it contains the two words that I put into researching. You can put in three, four, five, whatever you want, and it says, of course, they're separated by spaces. That's what the percent20 indicates over there. So, that becomes the new URL. So, what this means is that, as far as we're concerned, we can actually go directly to this page. We never really need to typeepicurious.com if you want to find recipes, for example, for say, zucchini omelette. You could just typehttp://www.epicurious.com,and then search, and then–what did I say–zucchini, percent 20, omelette and go, and we'll get a page that includes recipes for zucchini omelettes.

That's what the implication of a Restful service is, that we have networked pages. So, this points to this, and we can construct our own URLs to get data. What this Restful search is using is it's using the HTTP GET method to get data, which is why we can see our search terms right there, tofu and chili, right there on the page itself, in the URL itself. So, let's take another example. Let's take the New York Times, and this is a login example here. To log into the New York Times, when you go to the New York Times homepage, there's a little button that says, "Log in here if you aren't already logged in," and then, what you can do is, you can go and–when you click on it, it's going to take you to this URL. Maybe not exactly that, but effectively that. You might have to search around for it, but trust me, this is what it's going to do. So, you can actually go directly to this URL and type in your login information. You don't have to go to the New York Times page. So, here, when you go to the New York Times page, to their login page, you get this box over here, and in the box, you can type your email address and the password that you have associated with your New York Times account, if you have one and then click Login, and once you click Login, then the request goes back.

There's an HTTP request that goes back to the New York Times server in the form of a post request. Post requests are little bit more private, and we will see this in HTML later, a little bit more private. So, when you do this, it's not going to return to you like we saw in the Epicurious example. It's not going to return to you the key words that you entered, chili and tofu. It's not going to return those to you in the URL, the last thing you want to do is you want to see–you don't–the last thing you want to see is your password showing up on the URL of the page that you get back from your login request. So, what happens with the post request is, typically, a post request impacts the database of the server somehow, and in this case, what it does is it checks to see whether you are actually a legitimate user of the New York Times and you have an account there, and once it does determine that, it logs you in and send a token back that will now go back and forth with every request that you make, and keep getting checked against the database, and it'll make sure that you're logged in and that you can get all the logged-in benefits of the New York Times, which is not a whole deal, except that you don't get locked out of articles really, really soon.

So, that's second the kind that we get, and in those these examples, that is, in the Epicurious example, as well as in the New York Times example, what we get back is HTML in some form or the other, along with, in the case of the POST request, along with the fact that they're logged in,

and in the case of the GET request in Epicurious, It's just essentially sending the data back. It might keep track of session information, but that's independent of our being there. The next thing we can try is we could try the Google geocoding API. Now, APIs are very, very useful for getting data, and we'll see later that they are actually very nice to have around, and what happens in an API is that you send a request, and the request, because it's a Restful service, is going to go in the form of a URL, and you get back data. Typically, the data will come back in either JSON or XML format. That's really useful for us, because what we can do is we get a lot of data back. In fact, in some cases, we can get many gigabytes of data back in one single request.

Whereas if you're going to HTML pages, you would have to go through multiple pages just to get that maybe thousands, actually, in the case of a gigabyte of data, to get the same amount of data, which means sending lots and lots of requests and responses. And every time that you send a request out onto the Internet, you are slowing an application down, because the largest latency is the time that it takes for an HTTP request to gout of your computer and the response to come back into your computer. So, let's take a look at the geocoding API. So, the geocoding API from Google is a Restful API. Now what that means is that, again, it's sort of network. You can think of it as being network and structured in some way. So, in this case, we're going to use the mapping API. That's what produces the geocoding information. So, they get maps toGoogleAPI.com/maps, and then it tells us, very helpfully, that what we are doing is using an API, and it tells us that the API we're using is geocode, and it tells us that the format of data that we want back is JSON. I'm sorry, it could be XML or JSON, and we specified JSON.

We can change it XML, and we get XML data. And then, we have the parameters of our request. The parameter here is that we have a variable called address, and that address value is Columbia University, New York, New York. I put underscores over here, because, typically, a URL–always, a URL has to be without spaces inside it. We'll see that Python can handle that, but for now, you know, we'll assume that a pure URL is not going to have anything but spaces. So, if you take this URL and type it into your browser, then what you're going to get is you're going to get a response from Google, and the response will come back in JSON format. Try it and see what you get back. We'll look at JSON in a few minutes. In fact, we'll look at this very URL in a few minutes. But for now, just try it, and take a look at the response you get back and think about Python a little bit and think about what kind of data you're seeing, in terms of what Python object it's most likely to resemble, alright? So, that's pretty much it for getting data, for the basic structure of getting data.

Video 2 (13:40): Getting data: part 2

So, what we have seen so far is that we can send HTTP requests of either the get type or the post type and get a response back. And the response will be either HTML or JSON or XML. So, now our job is to figure out how do we send these requests? Clearly, we can make a URL. If you want to make a URL to get tofu chili recipes from Epicurious, we just write out that URL. We can do that. But we want to programmatically send the request, get the response, and then parse the response to get the data out of it. So, what do we need? Well, we need, of course, an ability to create and send HTTP requests. We need to be able to receive and process these responses. And finally, once we've sent the request, got the response, we want to be able to take the data that is in the response and extract it somehow. So, we need to be able to manage JSON, XML and HTML formats. So, let's take a look at the first figure, we'll look at how we send an HTTP request and get a response. So, to send an HTTP request and get an HTTP response, this is Python, so what do we do?

We find a library that does the work for us. The two libraries that are typically used, the basic library is this one, the second one that I've listed here in Python3, its urllib.requests. And in Python2, its urllib.2. And this handles the business of sending a request and getting a response. However, there's a very nice library that sits on top of all of this called requests. And that's what

we're going to use. It makes the whole process lot simpler for us. And our main interest is in data analysis, so we are not going to make things hard for us, basically, right? That's the idea. So, the first thing we want to do is look at the request library. Then once we get the data back and we figure out that our request and response cycle has worked well and we actually have something we can extract data from, then we're going to use Python libraries to get hold of the data itself. There is a library called JSON that deals with JSON files, a library called LXML that deals with XML files, and there are two libraries that are useful for scraping pages. Scraping pages means when you get an HTML response, you want to be able to get the data out of that HTML page itself. And the two libraries for that are Beautiful Soup and Selenium. Web scraping typically means that you are extracting data from an HTML page that you've got back from the Internet. So, these are our web scraping libraries.

And these two are our library that we'll typically use APIs for. So, the request library is Python–Python library for handling HTTP requests and responses. There's a lot of documentation on it over here. But we are going to use–it's not really very complicated. So, we will focus on how to use it effectively for our purposes. And the basic idea here is that we will use, import the library, of course. So, you first import the library. Then you construct a URL. Then you get a response. You check whether the response is valid or not. And then you move forward after you've checked that. So, let's take a look at how this works in practice and move through our iPython notebook page. So, I've set up a notebook here for you. You can open up that notebook and start moving. So, the first thing we do is we import requests. That's our step one. So, let's import request. Once you've imported it, we can take a look at what it looks like. Let's see what that is. So, what is that–here, if we go here? Let's say request. It does as it's a module, right? So, we have a module requests, and this is a whole bunch of stuff inside there. Then we want to use this library to get our web page.

So, the web page that we're going to use is for starters, for just for practice initially, is the Epicurious page. And the Epicurious page we're going to look for tofu and chili recipes. The function that gets HTTP response back, what it does is actually sends the request and gets a response back. It's called .get. As simple as that, right? So, we'll be calling the get function. And this–from our request library–and we get a response back from that. So, let's run that. And once we get that back, the first thing you want to do is check to see if everything went as planned. So, when you get a response back, what you get back along with it is a status code. So, let's take a look at the status code of this. And that says 200.The status code of200 typically means that everything went according to plan. It doesn't mean that you got your data back. It just means that the HTTP request and response worked as planned. When won't it work? Well, for example, if I do change search to PQR search and send this, and now look at the status code, I get a 404 response.404 means page not found.

So, in this case, what's happened is that my request went out, it reached www.epicurious.comsuccessfully.But when it asks for a page called PQR search over there, that page said–the server that there is no such page, and it gave me a 404 response. So, generally, if you get a response in the 200s, that's good thing. If you get–for most of what we do, we want our response to be exactly 200, which means that it worked perfectly fine, there are no proxies, no funny stuff going on, and you're actually getting your data back as asked for. But there could be are direction or a proxy server or something else that is coming in the way of your request, and the final server, and you might get a response of 201or 200 and something else. So, be aware of that. But typically, we really want just a 200 response. So, we check that. So, once we know that we have our correct response, a 200, we know that our stuff worked. We want to get the actual contents of the page itself. So, the contents are going to come back from an HTML server. It's going to come back in the form of HTML.

Anything that usually comes back as a web page is going to come back as HTML. So, what we're going to do is we're going to take that HTM Land convert it into a Python string. So, let's take a look at what this gives us. We find, yes, we have an ice little page here. So, once we've got our data back, we want to take a look at the actual data itself. And we use this function called

.content. So, response. Content is going to give us, remember, response is just the name of the variable we give it. We could have called it X or Y or Z or whatever. So, response. Content is going to give us the actual contents of our HTML page. Typically, when you use data from the web and you're getting HTML stuff back, the page might actually come back in the form of a particular special kind of representation called a byte string. So, if for example, if I looked here at a response. Content, then we notice that there is a B in front of this thing here. Over there, there's a B. If we can see this. There's a B there.

That B, what that indicates is that the contents are a byte string. A byte string is a special representation that's used on the Internet. And what you need to do is you need to decode it into Unicode. And to decode it into Unicode, we use the function decode, and we give it the coding scheme. The coding scheme can vary. There are lots of coding schemes. But utf 8, or utf 16, are the most common. So, generally if you are going to an English language web page, you can expect that the result is going to come back in utf 8 format–is going to come back and needs to be decoded using utf 8 as your decoder, alright. So, that's what happens here. And we can see that B has disappeared, and we start with a single code. So, this is a normal Python string. So, once we've got this, you move on, and we know we have succeeded. So, we go back to this stuff here. So, we've checked our response codes, so that's good. We have a 200.And we don't want to be in the 400s or anywhere else, okay–300s or 400s–not good. But 200s, good, okay? And generally, 200 is the right response code that you want. Once we do that, we want to get our data.

And for that, we can use response. Content to get the actual content. And if it is in the form of byte string, and you can always see that, when you're testing on your code, if there's a B in front of the string, you know it's a byte string. You decode it using utf 8 as a decoder. And once you've decoded it, you get a Python string. If that didn't work and you know, again, once again, you will actually see what the thing looks like once you've decoded it, if it doesn't make sense, or it's not a Python string, or it's got some funny characters, you know that it hasn't worked. So, test it all out before you package your program and run it as a matter of course. And the general rule of thumb, like I said, is that utf 8 is probably what you want. So, let's try this now. This is your goal at this point. What you want to do is go towikipedia.com, and rather, construct a URL on your Anaconda browser. And this is the URL that you want to use. Send in a request, get a response, check the status code, and see what you got back. If it's a 200, great. Once you've got that, get the content, decode it, and then search the page for the string, "did you know".

So, if you can find did you know, and to find that, you use an STR find function, then what will happen is that you will get a integer, a positive integer that will tell you where that string is located. And if it can't find it, then it will return negative 1. So, if you've got negative1, then that's a problem. If you've got a positive integer, then that's good. So, let's take a look at how we can do this. What I want you to do is actually pause your video right now and see if you can do it yourself, and then come back and move forward with the–I'm going to show you how it's done right here. So, let's say we start with a URL. So, we say URL equals blah, blah. We've got that. And then we want to import requests. Once we've got that, then we want to actually get it.

So, what we can do is we could say wiki_page_response, just to show you that the variable can be anything at all, equals requests. Get (URL). So that gets us the wiki page. And then we want to decode it. So, we could say wiki_text equals wiki_page_response dotcontent.decode (utf8), and run this. So far, so good. And then to this, we can add let's check our status code, actually. I never checked that. So, let me see if I can look at the wiki page response.status code. That is 200.That's great. So, we have a good status code over there. We'll get rid of this. And we have decoded wiki text. I want to search for did you know. So, I do wiki_text. Find, and give it the string, "did you know". And that tells me that byte 14, 822, which is where pretty much it is. So, if you look at the wiki page–the Wikipedia page here, if you go to Wikipedia, we find that yes, there is a "did you know" in it right there. So, that is there, and we found it, and that's a byte14, 800 and whatever. So, hopefully you got that. But that's good practice for actually getting the data itself. So, let's go back and move on from here now.

Video 3 (11:11): Web data formats

So, now we're going to look at the data formats we get stuff in. We know we can now send an HTTP request, get an HTTP response, and actually take the response and convert it into a Python string. So, my question now is how do we extract data from that string? And the way we extract data from that string will vary depending upon whether it's XON–XML, JSON, or HTML. So, HTML is the most common but we will postpone that for now. We're not going to talk about that right now. What we're going to look at is our JSON standard.JSON is really useful because most API's these days are going to return stuff in JSON. So, it's kind of useful to know that. So, let's start with that. JSON stands for JavaScript Object Notation and it was originally developed to exchange data between JavaScript programs or to store data. So, it's really something that's called–it's a standard for something that's called serializing, and serializing means that you convert data into text form.

So, you have a structured data object that contains detain some kind of form like your name, social security number, age, that kind of stuff, and you want to save it and then recover it so that it creates the same object that you had stored all over again without you're having to go through the process of creating it in your program. So, that's what serializing is, and so, JSON has become a standard for serializing, and the idea is, again, that you either store it in a file or you transmit it over the network. It's very nice because it's human readable. You look at a JSON file, you pretty much know what data is inside of it. You don't have to–you don't have to be a rocket scientist to understand that. So, it's very useful for data interchange because it's human readable and you can see it and it's textural so, you can just send it and almost every system that you have has some methodology for dealing with it, whether it's a cloud computing platform or whether it's SPARK or whether it's Python or whatever, everything can deal with JSON.load, it's also useful for representing and storing semi-structured data. So, you can store it in that format and recover it.

Of course you can use key value pairs and stuff like that. And it's told in plain strings or UTF strings–UTF-8 strings, so that's you know that's good for exchanging and storing stuff. So, what is inside JSON? Well, JSON has a bunch of constructs–that define data definitions really. So, JSON has things called numbers, strings, and null, true, false, object, and array. And if you look at this, we know that there are Python equivalents for each one of these. So, for example, a number could be either an integer or a float. If it has a decimal point it's a float. If it doesn't have a decimal point it's an integer. A string is a Python STR. Any JSON null type can get translated into a none-type in Python.JSON true, false–notice that it's a lowercase t and lowercase f–becomes Python true false, that is uppercase T, uppercase F. A JSON object is a Python dictionary, a key value pair.

There's a key and there's a value associated with it in a JSON object and that's exactly what a Python dictionary is like. And finally a JSON array is a list in Python. So, we have one to one correspondence between JSON–one to on each correspondence between JSON and Python objects. So, it's really easy to pretty easy to take a JSON object and convert it into a Python object which is what we want once we get data from the Internet. So, let's take a look at that here. That's easy to do in Python. There's a library called JSON. You import that library and then there are two functions. That's pretty much all you need. There are other functions too but this is pretty much what you need. There's a JSON.loads function. What that does is it takes a JSON string or a string that is called data in JSON format, and converts it into Python objects.

There's a JSON.dumps function that does exactly the opposite. It takes a Python object and converts it into a JSON formatted string and that's pretty much it. We can exchange information using that kind of stuff. So, let's take a look at JSON on our–so, when we're dealing with JSON, we want to first import the JSON library. So, we import the JSON library here. Now let's say we have a data string that is of this form here. So, this is our–this is a string and notice it's a string

because it starts with a single quote and every substring in it also has double quotes but that won't matter. As far as Python is concerned, this is one single string that contains this piece of text that happens to have double quotes inside it but it's not important, right? So, that's the–in fact, let's take a look at that. So, we see that's what data string is. It's just a string.

Now what we do is we say Python data equals JSON.loads data string. So, this is going to date the data string and load it into Python which means that it'll take everything inside the string, assume it's a JSON object and convert a JSON formatted string and convert each element into an equivalent Python element. So, that's what we get here. We converted this into a string–into a Python data structure called Python data, and we get this over here. And we notice here that in our data string, it started with a square bracket. So, the Python assumes that that's going to be Python list because a square bracket in the JSON string indicates that what you're getting is an array. So, if we look at the type, for example, of Python data, we will see that it is of type list, right. So, that's a list type of object because it's a list. Inside the list is a–let me run this one here. So, inside the list we have a–first we have a list. So, you think data string is type STR.

Python data is of type list. Then inside the list, the first element in the list which is essentially this, right, becomes a dictionary because it starts with a curly brace. So, we get a dictionary that has a key B and a value 2, 4, and then another key Can do another value 3.0. Another key A with a value uppercase A, right? So, then we get three key value pairs inside our dictionary. And inside each key value pair, every data item gets converted to its appropriate type. So, for example, the 2, 4which is our Python data 0B, so, Python data 0 is this entire list–this entire dictionary, I'm sorry. That's Python data 0. That's Python data0 and then the key B for that produces the 2, 4. And if you want to find out, for example, what the last element is that I could say here for example print type Python data, zero and the third key is A. So, I'm going to give it an A here and then look at that, and its class STR. That's a uppercase A right. So, if I took this out and took the type out–I get the A, okay. So, that's the value that you're finally getting.

So, what's happened is that the Python.loads–the JSON.load function has taken the entire string, JSON string, converted it into its equivalent Python object, in this case a list, and then recursively gone into the elements of that string and converted them into their equivalent Python objects. So, you've taken the sort of exploded it out and taken the whole thing and exploded it into different Python objects and each of them get combined together into this final Python data variable over here. So, that's JSON.loads. You have to be a little bit careful that if your format of a JSON string is incorrect, then you're going to get an error. So, for example if I do JSON.loads, hello, I get an exception. Why do I get an exception? Because the exception I get is–I get an exception, JSON decode error exception and the reason I get an exception is because here, I have a string but it doesn't contain JSON object. To contain a JSON object, it should have a string inside it.

So, what I need is a string that contains, hello, with double quotes around it because it's not an STR otherwise. So, that's the reason why this one fails and if I want to do it correctly then I have to make sure that it actually is a JSON object inside it. So, let me comment that out and uncomment this, and I run this and that gets converted accurately into a Python string. So, that's the JSON loads function, and the flip side of that is JSON.dumps. What JSON.dumps does is it takes the Python data object and converts it into an equivalent JSON string which is a pretty straightforward operation. So, if I take my Python data, which we saw was this entire list with dictionary inside it, and convert it into a data string using the .dumps operator over here, then I get a data string and we can see that the data string is of type STR and actually is this entire thing here, okay? I mean it doesn't look like a string. It doesn't have quotes in it but we could easily change that by getting rid of the print here and looking at the representation of the data string object.

There we go. So, now we have code. So, we see this is a string that contains a whole bunch of stuff and that bunch of stuff adjacent objects. So, this makes life easy for us. Given that JSON–correctly formatted JSON object, we can convert it into a Python object and recursively every element of that string will also become a Python object. We can then extract them, play around

with them, do calculate means, averages, whatever we want, variances, everything we feel like, draw graphs that do all kinds of light stuff for that. And if you want to do the opposite, for example, we want–we have some data and we want to send it over the Internet to somebody else whose doing something with it and we want the format of the data to be retained, we can use JSON.dumps, make it print to a string, send that string across the Internet and then the other person can just do JSON.loads and they are all set or use whatever other language they use in JavaScript or C or whatever–Whatever they can deal with JSON.Whatever they have something to do with JSON.

Video 4 (13:30): JSON, Google API: part 1

Luckily for us that even–we don't even have to do this. The request library has a function that automatically loads a JSON string into Python. So, for example, if we go to the API that we saw earlier for a Google API's geocoding, we can– and we send our request instead of having to do request, sorry, response.content.decode and all that kind of stuff. When we get the request back, we can just call the JSON function on it and it'll automatically load it, assuming of course that it is a proper JSON string. So, let's go back and look at our recording here. Let's make sure we dont miss anything. So, we've done this, we did this, and we are here now. So, what we are want to do is we want to send our requests. So, we'll create an address over here that is our Columbia University, New York, New York, or whatever you feel like, and then create a variable called URL and in the variable URL, we're going to take that address and add it to the back of our URL string, and that's the URL string that we want to send for any API.

If our API a– geocoding API–is going to return to JSON string, then we can just use this entire thing as our frontend and pop the address in at the back. The nice thing about the request object is that we don't have to worry about replacing spaces with anything else in the URL. So, though this URL is going to actually be a URL with Columbia space University, new space, York, space NY, we don't have to worry about it. Requests will automatically take care of that by putting its percent 20or whatever it sends to URL, and we'll be all set. And then once we do that, we just call request.get URLJSON and we're going to get our response in a proper Python object that we can then use to deal with and do whatever we feel like with it. So, what we want to do is make sure–now we've seen as we went along that stuff can go wrong, and whenever you're writing a program that actually interacts with the outside world, you have to be ready to face the fact that things will go wrong. For example, you're sending an HTTP request and you forgot to turn your Wi-Fi on, or the Wi-Fi stopped working for whatever reason. You just didn't have an internet connection.

That's not going to work. That's not going to happen at all. Or you send an incorrect URL, or the server that you are trying to reach is for some reason down. Maybe there is maintenance going on. Maybe there is some kind of, you know, power outage. I don't know; whatever. So, the server is down. Or you send the wrong page or the page that you are accessing is no longer available. For example, let's say Epiqurious changes from search–it changes its page name to B search. It can happen, you know. So, then your code won't work anymore. So, you should always be ready to face the fact that your code may not work. You may be expecting JSON object back but the server instead sends you a malformed JSON object. Be ready for that too. So, always check for exceptions and that's what we're going to do now. We're going to make sure that we have our everything properly checked over here. So, here's our basic checking mechanism, and what we're going to do is we are going to construct our address, that's this one here. We're going to construct the URL, and that's this one here, and then we're going to–at various stages try out the process and how we're going to get an HTTP error?

Is something going to happen in the middle? Is the status going to be something other than 200? You know, all those kinds of things. Is the response not in valid JSON format? And for this, we use the try and except mechanism that Python provides us. So, let's take a look at how this

works in practice. So, let's see how this works out in practice. So, we have on our little piece of program fragment or whatever they call it, we've got the address. We've got the URL which we've constructed over here. And so, for starters, let's take a look at the URL and make sure that worked. Always a good idea to make sure things work before you run anything. So, let's see what the URL looks like, and that looks good.URL is all this stuff and notice that there are spaces over here which wouldn't work if we actually tried this in practice, but they will work when we use our request library. So, the first thing we do is we put in a try accept over here. This is going to catch every possible problem that we might face with our request thing. So, if this doesn't work, then the exception–whatever exception we get is going to come here and we're going to get a method that says something happened.

So, we want to make sure that we can somehow deal with that. And then inside the try except, we'll try to catch individual problems that may occur. So, the first thing we do is we go here and get our response. So, we get a response and we check the status code. If the status codes 200, that's great, but if it's not we want to know that there is something happened. So, what we can do is we can print the status code and see what that status code was and take action accordingly. For example, what happens if the URL changes. So, if it works, great. Now we're going to try to convert our response into JSON.load, all we need to do is take the response and call the JSON function on that, then we get response data. And if that works, great, otherwise we say that the response is not invalid JSON format because that's where it's going to fail. And so, this essentially does our stuff for us and we get our response data, which is the final thing here.

And at the end of everything works nicely, we get a response data and we notice that it's of type dictionary. So, let's take a look at what this dictionary looks like. So, if we go here and we take a look at this response data. So, let's look at the response data. So, the response data contains we can see the dictionary. The dictionary has a key called results and another key called status. The status is okay which is kind of nice which means that it worked, and this is an important point because so far everything we've done has to deal with our HTTP request and response. So, the fact that we got a status code of 200, all that meant was that our request response worked. The fact that the result that we got back was convertible into JSON just meant that the result was JSON. It doesn't mean that Google actually gave us the data we wanted because if Google–if the Google part doesn't work, they're going to send back a JSON object with a result of an error inside it. So, the status here will be bad instead of okay.

So, you probably want to check to see whether the status is okay or bad. But assuming it's okay, we can see that our results is this entire thing here, and from this, we want to dig out the latitude and longitude of Columbia University New York, New York. So, that's what we have to do next. So, let's see if we can get --do this, and our goal in this exercise is to write a function def, kind of function called get at long address string that returns the latitude and longitude for any address string that we give it. That's our goal here. So, let's see how we can do this. Well, this should be reasonably straightforward. So, what we want to do is–keep that up a little calleddef.get_lat_lng bit. The first thing we want to do is to insert above there–okay great, so, now let's see what those results looks like. So, clearly we have a dictionary, right?

Our response data is a dictionary. So, we want to investigate this and see how we get a latitude, longitude out of this. So, the first thing we know is that assuming the status is okay–we should probably check that–we're not going to do that right now–assuming it's okay, we know that the result, the actual latitude and longitude is going to be inside the dictionary under the key results. So, we can start by looking at this key here, right? So, that gives us our step one. So, we know that that's our response data results–gives us our results. So, we look at this thing here and we find that this results is actually a list because it has this square bracket there.

The reason it's a list is because Google geocoding API will give you typically every possible result that matches their dress string you've given it. So, for example if a dress string is Main Street, it's going to give you lots of results. It may or may not be every main street but it'll give you quite a few of them. So, you get back a list of address components, but in our case, we have only one because as you know there's only one Columbia University. So, we get only one back

but it still comes back in the form of a list just in case. So, we find that that's the list. So, the next thing we can do is we can extract the first element of that list and we do that here, and we get this. So, the first item on the list is now a dictionary.

Remember that when we did our JSON.loads, when our request.JSON did its JSON.loads, what it did is it took every–recursively every object inside the JSON strength and converted it into an equivalent Python object. So, the equivalent Python object here is a dictionary. So, we got this dictionary and this dictionary has a whole bunch of keys in it. Address components, long name, long name, etc., etc. So, let's see what all these components are and we can do this–examine this a little programmatically. We could say for–let's say thing in response data zero which we know is a dictionary. So, this is going to pull out each key in turn. Print thing. Where thing is actually the key, right? So, we pull this out and we get it has four–five, sorry, different keys.

Address components, formatted address, geometry, place ID, and types. So, we have all this stuff here and let me go back to this. It's another cell there. So, now we can take a look at each one of these in turn and decide what we want. But we don't need to do that because we can see that amongst these things that one of them is called geometry. So, let's take a look at geometry and see what that pulls up just because that maybe that's what we want to look at. Geometry–geometrical location, all that kind of stuff, right? Helpful wording for us. So, response data. Results. Zero pulls them all-out, and if we do geometry, then we get all this stuff here.

So, that's another dictionary as we can see, right? We know it's a dictionary because it has this helpful curly brace in front of it. So, that's another dictionary, and inside the dictionary we've got a key called location, a key called location type, a key called viewport, and a key called viewport–viewport which is a dictionary that contains a key northeast and a key southwest, and each one has latitudes and longitudes. So, this viewport is probably our bonding box for Columbia. It probably shows us where the northeast corner is and the southwest corner is. So, it gives us a little box that locates Columbia. But, all we really want is the latitude and longitude so, we could just look at location. That's pretty much what we're interested in.

So, we could go here and add location to our recursive dictionary access that we're doing, and we get latitude and longitude and if we take this, we find that's a dictionary again with two keys. So, we can take this and do latitude and that gives us the latitude. And if we did longitude, we would get the longitude. So, what we can do actually is we can save these. I could say here latitude equals this, and–if I say print, latitude, longitude, then I get the latitude and longitude of Columbia University. So, our goal is to write a function and typically this is what you want to do. You want to take everything that you work with and pack it into a function so that you can use it anywhere. That's the idea.

Video 5 (1:45): JSON, Google API: part 2

So, this is what you have to do now. We have all the steps in place. What we have is, we have–over here we've got our–in box 27; it could be a different for you depending on you ran this. We have our address construction–URL construction mechanism, and then we have our error checking. And once we do all this, we've got a proper response that we know is going to work. Then we have–we can–we've got the response data itself. From the response data we want to extract the latitude and longitude, and we can do that using this stuff over here. And instead of printing, you want to actually return it. Just remember that before you write this you want to make sure you import requests and you'll be all set, okay? And instead of, of course, you won't actually be typing in the address anymore because the address will come in as an address string in the function down here.And instead of address in this location you need to put in address string.

And once you've finished that first problem, then try to extend the function so that it takes a possibly incomplete address. Now remember that we can get more than one address back from a URL request. So, we could get multiple addresses back. And from those multiple addresses, you want to extract the complete address, latitude and longitude from the JSON that gets returned. I actually have the solution right here but try to do it without looking at the solution and seeing if you can figure it out.

Video 6 (10:31): XML: part 1

So, the next format you want to look at is XML which is short for extensible mark-up language, and the thing about XML is that it follows what is called a tree structure and it has tagged elements. It has a lot more detail that is attached to it than JSON does. So, it's sort of harder to look at to read because it can be kind of a little bit hard to read but all the information is actually there. And along with each detailed element–along with all element you also get a set of attributes. So, this is very much like HTML actually because they both have the same root language which is a little bit like Spanish and French are both Latin-based languages like Italian is. You know Latin-based languages similarly XML and HTML are both have the same root language behind it. And the way this works is that you get a tree of tags or attributes. So, you get a tag, another tag, another tag, another tag, another tag, and then under this is the actual text that is the data that you want. The data is always on the leaves and the leaves could be in multiple places.

You could have data right here, for example. But the data is always in the leaves. So, to get data in an XML tree, what you have to do is you have to traverse the tree in whatever direction you want to go until you get to data. That's the basic idea with XML. So, let's take an example. So, here's an example, and you can see that it's not–it all makes sense but it's sort of hard to read because it's a lot of peripheral information around here. So, this example what we have is a bunch of books; and for, each book we know the title of the book. That's the title, for example. New York Décor. And we know the first and last name of the authors, and we know where they live, and we also have some other information like the weight of the book, not that you should judge a book by its weight but just in case, the price of the book, and the ISBN of the book.

So, that's the basic information that we have, and all this is sitting inside our XML tree, and our goal really is to extract this information from that tree, alright. And we know it's a tree because the tree has a root at bookstore, and we can take a look at it. So, let's take a look at an XML example in the form of a tree. So, in the tree we find that there's the root of the tree is a bookstore and under the bookstore we've got two books, book one and book two. And each book has a bunch of attributes. So, here I haven't listed them all here but we can see that this book for example has an ISBN, a price, and a weight. And then under the book, there are two nodes or elements. One is the title here and the authors, and then is the actual data itself, the title of the book, New York Décor, and under the authors are the list of authors of that book. So, we can follow the tree along here as well. There's a book. It probably has a bunch of attributes not listed. It has a bunch of authors and the authors are–each author is one sub tree of the authors sub tree. And author one has a first name, a last name, and then under each first name and last name element are the actual names of the authors. So, we've got Bill and Harris.

So, we have an author called Bill Harris, first name Bill, and last name Harris. So, this becomes our tree, and this is just another way of looking at this XML structure over here. That's the root. It has two sub-elements. A book and a book. Each sub-element has an open element book, and a closed element/book. And for the elements of book, for the book element, we have a bunch of attributes and the attributes are ISBN number, price, and weight, etc. So, we can go down here author for example. Authors, for example, have an author residence; New York City, and author residence, Beijing. Author residence, New York City, you know, and all that kind of stuff, right? So, as you can see that there's an open tag. The open tag contains the attributes and the closed tag, doesn't contain any attributes–it makes sense. So, this is our tree structure. Our goal is to traverse the tree structure and extract the data. The data being in this case, for example, the name, let's say, what is the first and last name of the author of the book, New York Décor? So, what we would need to do is we'd need to find this element, the entire book element, and from the entire book element, we figure out yes, its title is New York Décor, and the first name of the

author is–look at the author's sub-tree, and look at the author's sub-tree. This one here. And then extract the first name and last name as the leaf nodes of that tree and then get the data.
So, that's the whole idea on XML. It's a little bit more complicated than JSON because JSON is easier for people who know Python, programmers who know Python because you can convert the JSON object directly into a dictionary or a list, and then, you know, we know how to deal with lists and dictionaries and we can traverse them very easily and get data out. But here, we are getting a structure that's slightly different so we don't need to actually figure that out there. So, as always with Python when you want to do something there's a library that will do it for you. There's a library for everything in Python. There's a library for whatever you want, right, for finding flights to Mexico or whatever. So, let's say we want to start with this library. The library that we're going to use LXML, and from there we're going to import this object E-tree. Object definition E-tree. E-tree is the object that will contain all the data type that'll contain the entire tree–XML tree that we're going to get from a file or wherever it shows up from or the Internet or whatever. So, we got the E-tree object. So, the first thing we want to do is we want to figure out the root of that tree.
And this is just a variable name. Remember it doesn't mean root we call if we like, but what we're going to do is we take our data element, and data if you recall is nothing other than this string. So, this is an entire Python string. So, we have this Python string and that becomes our data. We take that data element and we give it as an argument to the function XML in the E-tree library, in the E-tree module, and that produces for us the root of the–because the whole entire tree actually and returns just the root of the tree. And then we can print the tag of that. And once we have that, we can do various things to actually examine the tree itself. So, let's take a look at this in our Python notebook. Go back to our notebook here and–so, we have our XML object over here. So, this is our data string that contains our entire XML object, right, and exactly what we had over there. And then we import E-tree which is what we want over there, right, and then we can get the root, and we look at the root itself and take a look at it, and it tells us that we haven't actually done this part here so, let's do that. And it tells us that the type of tag that we have is of type STR because that's the whole thing over there. So, now because of XML structure is–you can think of it as containing the data which is this entire structure here and containing–extracting the structure from this string and that's what we do by the end of the XML function itself. So, let's go back to this here and we can look at pretty printout tree itself, and we find that it's essentially this printing the tree structure, okay, and we have to decode it from UTF-8because using UTF-8because it's byte string. So, we got that, then–so, what we want to do is–the first thing you might want to do is iterate over the XML tree.
So, if you want to extract all the elements in the tree, we can use an iterator. An iterator in Python, which is very useful in general, it's a generator that doesn't actually–something like the range operator we saw when we were looking at Python last week, it doesn't actually generate–it doesn't instantiate every element but it generates them as and when we want them. So, iterators are very useful for that.00So, what we're sort of doing is we're saying we have our tree structure and the tree has a bunch of nodes in it, and what we want to do is we want to construct an iterator that is going to sequentially go over these–go over these nodes and return them one by one to us so that we can do something with them. So, let's take a look at what this does here. So, I get the root.iterator, and in fact what I could do is–let's take a look at that in a second. I can print each element and what it tells us is that our tree contains a bunch of elements, a bookstore, a book, a title, authors, author first name, last name, etc., alright.
So, we get all these different elements in our tree, and we can take a look at what kind of object or root iter is. So, if I say root.iter, it tells me it's a element depth first iterator. So, it's going to be a depth first iterator which means that it's going to–in our tree it's going to go down one complete branch and then come back up one step and do the next thing. So, that's what we get here. We get bookstore, book, title, authors, author, first name, last name, and then it's going to go back all the way up to the next book and get that. Let's take a look at what the thing does here. So, if you look at our tree here, the iterator is going to go from bookstore to book, book to title, title to

authors, author to author, authors to author, then go back up. It's done all this, done all this, go back there, done all that, and take the next book and then go on and so forth. So that's what that first iterator does.

Video 7 (9:31): XML: part 2

We can also, if we add a particular node, we can also go down and get just the immediate children of that node. So, we're looking at a tree. When you look at a tree, you can think of an element as having–as having children, and then each child having its own children. So, from a sub tree, rooted at an element, what we end up getting is a bunch of descendants of that sub tree, right, of that node. So, given that we are at a particular node, we get a bunch of descendants of that node. So, if we want the immediate children of that node, all we can, we need to do is iterate over the root itself, so, over the element itself. So, here we say, starting with the root of the tree, we say, give me the children. So, what are the children of the root, which is, in this case, bookstore, the children of bookstore, that's our root, our book, and book? And let's take a look at that. And that's what we get. Book at this. Book at that. So, we get two books there.

For any element, given an element, we can access theta at that element. So, if I do–for–let's say, child and root, I'm going to get first this child and then this child, and I can get the tags associated with that. And it tells me that book and book, that's the–we can see what the tag is, if you want to do something with it, alright? So, we got that. So, now let's go and see if we can find the first name and author and last name of an author, given an author tag. So, what we're going to do is we're going to use our iterator. So, that's the iterator that we had, root.iter. And we're going to say that instead of iterating over the entire tree, of course it's going to iterate over the entire tree. Give us only the nodes or the elements that have authors their tag name, okay? Not all the other ones. And for each author as a tag name, we want to find the first name and last name associated with that author. And remember, these are first name and last name–are also–if we have here–what we're going to do is we are going to go through our tree and we look for any tag that has the name author, and then stop at that tag and say, get me the tag first name and the tag last name that belongs to the author. And this is only going to look down at the children and nothing else. We're not going to look at any descendants. In this case, there aren't any.

But it's not going to actually look at descendants. So, if we do that and we go back here and run this stuff, and then the last step here, of course, is we're going to see say, give me the text associated with that. So, now we are saying that for an author, find the first name and get the leaf–text from the leaf for that. So, we're actually getting data here.And similarly, for the last name. So, what this tells us is that, what this will give us is all the authors in our XML tree. So, we have three authors; Richard Berenholtz, Bill Harris, Jorg Brockmann, and those three authors show up over here's, sometimes it's useful to know the structure of the tree that is who the parent and what parent and child combination would produce a certain answering the tree. For example, let's say I have this tree here again, and a part of my tree, instead of author, first name, last name, I also have here a title. And the title is a leaf that says, say Mr Bill Harris, right? So, the title of a person like Mr, Miss, MS, Mrs, whatever, right? So, Dr., Professor, all that kind of stuff. Since I have this, and if I look in my tree and I say find, do a root.iter, iterative over the tree, looking only for tags that have the name title omit, the tag has the name title, and I'm really looking only for the titles of books, I'm also going to get Mr, along with New York Decor and 500, whatever, of New York, right?

So, that's not so great. But what I do know is that this, if I'm interested only in book titles, then the thing that I'm interested in is in the format book and title. This structure, right? Not, this is good, and not the structure that contains author and title. This is bad, okay? So, what I want to do is, I want to say, instead of saying find title, I want to say find the structure book slash title.

This structure is called an XPath. And XPaths are very useful because of exactly the reason that you've seen here. So, let's take a look at how we can do that. And what we can do is we could say for element in root.findall. So, root.findall is going to find all tags that match this XPath. Root.find will find one tag that matches, the first tag that it matches. And root find all finds them all. So, this is root.findalltag, all the tags that match book.title and print (element.text). So, let's look at how that works. So, here, we have the same structure here. For element in root.findall (Book/Title): print (element.ext). So, what this should print for us is all the books that we have in our set up here. Let's look for the book title combination and print it out the text that is associated with the node that has this book.title combination. This is that bottom of this sub tree, or this XPath that you can think of, okay? So, let's say, let's see if you can do this. Find the last name of all authors in the tree root using XPath. So, let's go back to our diagram here, just so that we have something to refer to.

And here, we see that to find the last name, then what we need is an author last name combination. We technically don't need that, but let's see how we're going do that. So, find it– find all this using the XPath that goes up to last name, alright? So, let's see if we can do that. Try that, and then I'm going to try it here. Pause this and try it. Seeming you paused it and tried it, let's see if we can work this out. So, for element in root.findall. And now you want to give the XPart that takes us to last name. So, that would be book/authors/author/lastname. And then we can print element.text. And that tells us the lastname of all those authors is Berenholtz, Harris, and Brockmann. We've got three such authors there. We can also extract attribute values. We saw that in our tree, we've got a bunch of attributes. So, book, for example, has an ISBN number, a price and a weight. And we want to extract these values as well because this is data as well, right, for us. That's pretty straightforward. All we need to do is we need to do root.find. So, remember, find will return just one item, not all of them, the first item that it finds. You give the thing you're looking for, book, or the XPath, or whatever. And so, we are looking for it here. We have an entire XPath, actually.

So, we've got this entire XPath. And we are saying in our XPath, because this it's going to technically return, you know, in our case, it'll return only one, but if you had find all, it'll return many of them. We want to say that only those books that have a weight of 1.5, whatever 1.5 is, so, the weight is specified that is to put in square brackets. And then at the rate of, the name of the attribute, followed by equal to, followed by the value of the attribute. So, this gives us the value of that book. And that's the author's first name for the book that has a weight of 1.5, which we can see in our tree is this one here, right? So, if we can go back up one, here we see that in our tree that the book with a weight of 1.5 has an author with the first name Richard. And that's what we end up seeing in this place here. Richard, okay? So, that's where we have done that.

So, now, as our final exercise for this class, what you want to do is see if you can find the first and last name of all authors who live in New York City for the tree that we have over here. So, note that in our structure, and I'm not going do this for you, you can try this, the answer would be posted online, in our structure over here, we have authors have an attribute called residence, and that residence is New York City here. So, what we want to do is we want to–in our–from our data structure here, we want to pull out the first name and last name of this author and the first name and last name of this author. So, what we should see is two lines. The first line should say Richard Berenholtz, and the second line should say Jorg Brockmann. That's it. Thank you.