Week 8
Video Transcripts

Video 1 (10:00): Getting data from the web: part 1

Today, we look at how to get HTML, which is essentially going out to the web and scraping data from web pages. So, that's our goal for today. Essentially, what do we need to know? Well, we need to know, to be able to do this, probably we need to know some amount of HTML and CSS. Not a whole lot, but just enough so that we understand where the data is on our page and how to get it out of there. We need to be able to, of course, send and receive HTTP requests, so, we need this whole stuff over there. And finally, once we get the HTML back, we need to be able to parse the HTML that's returned by the HTTP request and the issue to request process. So, these are three things we need to know. So, let's start with the HTML, and that stands for Hypertext Mark-up Language. It's actually a very simple language, because the whole idea in Hypertext Mark-up Language is to take a piece of text and format it. And just like Word or pages on a Mac formats a document, similarly, Hypertext Mark-up Language formats an HTML page, takes the text on an HTML page and formats it. The only difference is that you, when you're writing the page, when you're writing the–structuring the page, you actually have to write the HTML code yourself unless you use an HTML editor, or some sort of HTML page generator, I should say. So, the basic idea here is that the HTML page consists of content that's text. And all the text lives inside tagged elements.

The tagged elements are what actually mark up the text. And the elements can have attributes, and the attributes often contain formatting commands. Like, for example, the colour of the text, or the font size, or those kind of things. We don't really care but it's derived from HTML. That's the overall language, if you want to look at that. It's closely related to XML. And we've already seen XML, so, we see that there are similarities, structural similarities between XML and HTML because they both are SGML-based. And the only other thing that we need to worry about is that in an HTML page, you can also find runnable scripts, typically written in JavaScript. And that complicates matters a little bit because a script is essentially a program that generates, or a program fragment that generates an output. And until that output descriptor is run, you're not going to get the output. So, that complicates things a little bit. We talk about it, but we're not going to really delve into it in this class. We're going to assume that our HTML pages are well structured in HTML itself. So, we'll briefly look at HTML over here. And I will point you to resources so, you can find out more about it. But if you look at this, a typical HTML page, this is what it looks like. Essentially, we have a bunch of tags. So, here, for example, this tag says html, and that says slash html. So, what this is saying is that all the stuff that is between these two, that is all this, there's an HTML, piece of HTML code, so to speak.

It's not strictly necessary to put an HTML tag because it's an HTML page, but it's a good idea to do it for forward compatibility. So, that's the basic idea. So, everything, the content is always going to be included inside an HTML tag. Let's look a little bit deeper into this. The page is divided into two parts; a head and a body. The head contains met information about the page. And typically, that includes the title. And the title is what shows up on the top of your page over there. So, if you look at an HTML page, we have here, for example, our notebook for today. And we notice that it has a title, which says over there, I don't know if you can see it, but it says web scraping. That's the title. So that's included inside a title tag inside your HTML page. So, that's the purpose of the title. And then it contains some Meta information. And more important than anything else, it contains what's called the style of the page. The style, and we'll see an example later, is typically CSS style. And that tells HTML how-to format individual sections of the page.

So, how do you want the head–the heading to look like? How do you want a body to look like? Those kind of things. So, style is kind of important for web scraping. And so, it's important to note that the style is really where we're going to scrape data from, you know, we're going to use it to scrape data. Then there's a body part. And the body contains the actual contents of the page. So, this is what really shows up on the page itself. So, in the body, we've got various things. So, there's an h1, which is a bold, you know, big letters, heading kind of stuff over there. And this is where you are in this world or whatever. And you'll see that in big bold letters, somewhat like this anatomy of an HTML page. Not that this is HTML, but that's the idea of it. It's a heading. Just like in Word, you can fit a heading or a body, you know, a heading, a subheading, or stuff like that. Then there's a script in this page here. And the script is a Google API script because this particular example that I'm looking at here embeds a map inside. Or actually, no, it uses an API to get some information from Google. So, actually, it does embed a map. It embeds a map in the page. Then there's a div tag. Div tags divide your page up into sections. So, the idea is that you have a page. You want it to look nice, you know, like in a brochure, you might have a panel on one side, a graph on the other side, a picture or an image in another corner, and you want to decide beforehand what the different segments of your page will look like. So, typically, you put them inside a div tag.

And in a div tag, we see here it has–so, the important thing to note here is that every tag has an open and a close. So, we've got an h1and a slash h1. And the text that's inside is what is marked up. Some tags, like actually any tag, can also have additional information in the open tag part of it. So, this is the open tag here. And it contains an additional piece of information, which says class equals list box. Class is a CSS selector. So, what we are doing is we are saying here that use the formatting commands that are inside a CSS definition, CSS is cascading style sheets. Inside a CSS's definition called listbox, use that format for whatever follows in this div tag. So, this is kind of important, because if you think about it, when you're structuring the web page, and you're designing a web page, you're going to define different segments of your page. And for each segment, you're going to have a different format. For example, you might have a you might divide your page up into parts, and you might say, this part, I should probably change this, so you might say this part contains an image, this part contains a panel, and then this part over here contains text, right? So, now, the first thing you're going to do is you're going to say, how big is this part? How big is this part?

So, each part has its own size. And when it has its own size, you're going to define that size, and that's what is going to show up in your CSS listbox, or selector, or the value over there, right? So, whenever anyone–any organization that's delivering HTML to you, is designing the page, they're going to include certain formatting information inside the tag so that you can so that the page can format it appropriately. And we can use that formatting information to decide whether there is valuable information or not in that section, okay? We'll see that later on. So, that's the general idea here. So, let's look into the head over here. In our head page over here, we contain–it contains a style tag. Inside the style tag, we're defining the different formatting commands that the page might have. So, let's take a look at one of them over here. And the one we're going to look at is a listbox. And we're seeing here that a listbox, a div tag that contains a listbox class should have a background of light green and a width of 500 pixels. That's what we're seeing here.

And, what this works up to be in our body itself, we say, when we define the div tag, we want the map to appear. So, in this case, like I said, it's a Google map that's going to show up, embedded map that's going to show up. We are saying that the div tag here contains a class equal's listbox. And this is going to use the formatting that we gave on our page at the back over here, which means light green and 500px.And it's going to use that over there. So, in this example, the listbox tree doesn't contain anything. But what we want to do is we want to say, okay, if the listbox, for example, might contain a list of places, let's say the map is actually a list of restaurants in your neighbourhood. So, the map here contains, shows you the map itself, and there's more to this on the right, which is not showing up on the page. The map is the actual Google map. And then, under that, we have a listbox that contains a listing of all the restaurants. So, this would be substituted by a listing of all the restaurants. And we will know that when we

are scraping the page, we will know that we need to look for a div tag with a class equals listbox to find the list of restaurants that we want to look for. And that's the idea here. There could be more than one div tag on the page.
In fact, there could be many div tags on the page. However, each div tag is formatted in a particular way, depending on the content. And we can use that formatting, for example, the listbox formatting, as a mechanism for figuring out which section of the page contains useful information for us. So, here, listbox, class equals listbox contains that, and so that's what we would use to actually find the data that we want.

Video 2 (8:32): Getting data from the web: part 2

Here, we have anh1 as heading format. And, this is if you have an h1 tag that contains a heading format attribute or value, then, that should be coloured blue violet. So, we can take a look at that here. So, here, we have a second example. In this, we have anh1 tag with, you know, several tags, actually. It says that if you have anh1 tag with a heading format as a CSS selector value, then the colour should be blue violet. So, this is what this, so the way this would work here is it'll say h1 class equals heading format, and then CSS examples is going to be coloured blue violet. We also have another tag here, the p tag which is a paragraph tag, and it says it has an ID equal’s cursive green font. The main difference between ID and class is that class applies to specific tags. So, this is saying div. explain_ box. Therefore, when we want to format something in explain box, we have to use it with a div tag and give the class equals explain box value with that. However, what this says is hashtag cursive green font, and this is not attached to any particular HTML tag.
It’s an independent one, so we can use this in any HTML tag. And, here, we're using the paragraph tag, and in this we only use it in any HTML tag. We have to give it an ID value. So, any CSS selector definition that starts with a hashtag, it’ll be, is used with an ID. And, any CSS selector that starts with a HTML tag is used with a class. That’s the main difference. But, we, you know, we don't really have to worry too much about this. Our goal is to understand that in HTML, there are tags. There's a tag, let’s say p, and then there’s a–there, it can have a bunch of attributes. So, for example, you come up with an attribute that says article equals something, xyz. Then, there's text, and then, there's a closed tag. And, by using–by trying to find this section here, we can identify the location of our text and extract it from the page. That’s our goal, okay? The last kind of tag we want to look at is the A tag. Actually, not. It’s not the last tag. We have one more tag to look at. The, we want to look at the A tag. The A tags are very important. With A tags are what are called annotate tags, so this is–this tag is an annotate tag. And, the annotate tags connect us to other webpages.
After all, what is a webpage? A webpage is a bunch of text, and inside the text are embedded links. And then, you can kind of take your cursor and hover around over that link and then click it. And, you go to another webpage. So, the page has to know what the next page is going to be when you click on it. And, that's the embedded inside an A tag, and it uses the value of an attribute called a Href to know where to go when you click on it. So, when you get the way to think of this is that an A tag really is a href equals http–must start with an http–with an http link, slash, slash something. That’s the open tag. And then, there's a link. Click me. And then, there’s a closed tag. So, this sets up our clickable link. What’ll show up on the screen is click me. That's all, right? And then, when you, and it typically is coloured blue or some other colour depending on whether you’ve clicked it before or not. And all the browser does, and then you take your cursor, and you hover around click me. And, what happens is that when you, and when you click on it, then at that point, the HTML page goes and picks up the link that it's supposed to go to and goes to that link, basically.
Right? So, if you're hunting in a page to find all the HTML links, you really need to look for all the A tags. And, we'll work through an example where we do that. The only other thing left here is

the forms tag. So, often, when we ‘reusing HTML, we are also entering data into the page. For example, you might log into a page, or you might have, might be providing some information about yourself or entering a bunch of keywords for a search on the page. And, when you're doing that, you have to send data from your browser which is a client, to the server which is sitting remotely somewhere. Like, for example, maybeamazon.com or something. It’s sitting in Seattle, and you could be in New York or Paris or Beijing or anywhere, right? So, that's the goal there is to send the data across. The data from any–from a client to a server–typically is collected in a form, and then it goes across in an http request object back to the server. So, the format of a form is fairly simple.
The idea in a form is that you’ve got the keyword, the tag form and closing tag slash form. You have an action. The action is what your page is going to do after the user clicks the submit button, right? So, the user clicks a submit button, and we have seen what that is. Then, your, then, you know, typically, you know, you send the data to, back to the server, and that server is going to run some kind of a script or do something like that, okay? So, here, I have after formed the HTML, but it might contain some other URL reference which would typically contain the URL reference. So, we're not worried too much about method and all that stuff here, so we’ll ignore that for now because we are essentially data scrapers. But, the key thing here is the input field. All the data that you enter is collected in, through the input tag in your form. Input tags come inside a form, and they collect the data. So, this, there are various kinds of data that you can give. You can give text data. You can do radio buttons. You can have a date box. You can have numbers.
You know, all kinds of things. And so, the key in an input tag is there's a type which tells it what kind of data is expected, and there's a name. The name is very important because the name is like a variable name that is sent back to the server so the server knows that that's the value that is, it's collecting. So, typically, when you send this back, it’s going to send the back data will saying name equals John or something like that if you entered John over there, right? So, that'll go back in the http response. And then, the server will know that the–sorry–not name. Your name. Your name. Yeah, because it has to use the name of the variable. Your name equals John. So, then, the server will know that it has to, that the value of your name that the user is sending back to the server is John. And then, it can use that in a database query or calculations or whatever else it’s doing to get the data back.
So, when we, as web scrapers, sometimes we need to send information to the server as well through the program. For example, if you want to log in programmatically, you’ll have to send your username and password through your program. And, to do that, you need to figure out what is the variable that contains the–what is the variable that contains the name, the username or whatever, and what is the variable that contains the password. And, put that into your http response, and, sorry, http request. And, send that to the server so that you can log into the server itself. We’ll take an example of that shortly, but that's what we, you know, we want to be able to know that we need to figure out what the input variables are inside a form. If you have further interest in Homeland CSS, which you should do. It’s a good idea to know these things. You can go to Khan Academy, and I have put the link over here that you can use. So, if you use this link and got to Khan Academy and study the HTML CSS. Do the first two topics, intro to HTML and intro to CSS, and you will be, you know, well prepared for web scraping because you'll understand what format the data that you're getting back is coming in. And, that's kind of useful.

Video 3 (10:09): Web scraping

So, for starters, what is web scraping? Web scraping is very straight forward. It’s automating the process of extracting information from web pages which means that we are programmatically going to a web page and picking up information from there. Like, that's the idea. Instead of, normally, what happens is that you go into a browser, and you type something. And then, you know, give a URL, and then, the URL goes to your server, to the server that you're addressing. And, that server processes it, gets data, does whatever it's supposed to do, and returns the HTML back. And then, your browser takes that HTML and renders it. So, this is what really happens, right? So, we've got a browser. Whoops. We have a browser. It sends it to the server, and the server sends stuff back to the browser. What we want to do is now we want to say we have a program, and the program is going to send something to the server. And, the server is going to send something back to the program. So, now, instead of rendering the page on the screen, we’re going to use the HTML that comes back to extract data from the HTML. That's the goal here.

Before we go into all that, we should take a look at a few of the legal and ethical issues involved because it's, you know, we're doing something that is different from the intent of the web page. So, the first thing you want to do is you want to make sure that it's, that the terms of use of a website don't prohibit you from using programs. In general, they will probably say that, but, you know, if you're using it for educational purposes, you're probably okay. But, if you're going to use it for commercial purposes, then it's worth checking the terms of use carefully before you do anything. In general, if you're getting factual data, for example, if you're scraping a page to find the population of the United States in the latest Census, I think it's okay because, you know, that's a fact. It’s not something that is generated by their website. It’s not intellectual content of that website. So, you are probably all right with that. So, factual, non-proprietary data is generally okay. If you are scraping proprietary data, then it depends on what you're going to do with it.

If you're going to take data that, let's say, you go to a stock trading website and you collect certain technical indicator values from there. So, that website is taking–web server is taking equity market data, for example, and then computing these technical factors. And, if you're going to scrape that, and then, and use it for your own commercial intent, then that's probably not kosher. Check with your lawyer, but it’s probably not kosher. So, you should know what the use you're going to make of it, and sort of figure out whether your use is fair use or not. Very important. You should always check, make sure that there's no damage to the scraping. For example, if you write a script that's not working very well, and instead of going and getting one page, it keeps hitting the server and becomes, you know, a fast crawler that is going through millions of URLs in a second. Then, what's going to happen is that you're going to be damaging the server because it's going to slow their server down. And, in the worst case, you might go get in a situation where that server cannot serve anybody else. That’s called a denial of service situation, and that's pretty serious.

So, you want to be very, very careful whenever you’re writing a script that scrapes the web that you've tested it out very carefully before you actually do it because if you damage the scrape, then you're doing something that is extremely bad. Again, public versus private information. Any information that’s private is probably better not to scrape it. If it's public, then why not? Ask yourself for the purpose. We already talked about that. If you can, try to get the information openly. So, rather than scraping surreptitiously, you might want to see is there an API. If there's an API, then, typically, the server, the company will have some kind of API service that you can subscribe to, and they will have limits and all kinds of stuff. And, you can probably pay money and do those kind of things to get more information. But, if they have an API, you can run tests very, very quickly without having to be doing anything that may or may not be kosher. Okay? And, finally, ask yourself, is it a public interest involved. If there is a public interest involved, then I don't know, it’s probably alright.

So, let's move on and see what other libraries that we have for web scraping. So, our three goals in web scraping are to be able to send a sheep requests and responses, to be able to get the HTML request back, and then to extract the information from it. Python provides three basic libraries that are very useful for this. There are many, many libraries, and many of them will actually build upon these three or provide services that are above and beyond this. But, we're

going to look at only the very basic stuff here. So–and the core stuff–the first library, of course, is request which we've seen before. It handles the entire HTTP request and response cycle that is necessary to send the request and get the HTML back. The second library is called Beautiful Soup. What Beautiful Soup does is, it utilizes the fact that all the content is inside tags, and it can quickly parse the page. And, you can ask, you can query the Beautiful Soup object.
That is, it'll take the page, construct a kind of a multilayer dictionary out of it, and then, you can query it very quickly to zoom in on particular pieces of content that you’re interested in. So, we'll take a look at that. That’s our–the main library that we look at. And then, the third library called Selenium which is originally designed to test a server. So, when you're building a server, you want to be able to test what'll happen when users come to your server and interact with you. And, you can't hire thousands of users to do that stuff. So, what do you do instead is you write a program that keeps hitting your server with requests. And, Selenium is a package that supports that. It’s actually an independent package Python has an API to Selenium that supports that activity, really. That you know, programmatically testing out a server. But, because it’s hitting the server, we can also use it to scrape the server. And, we're not going to study Selenium in this class, but it's particularly useful when a pace contains scripts. Because when a page contains scripts, then what happens is that you send a request to the server, the server sends back a response.
The response is, you know, contains the HTML code that is going to be displayed on your–that'll be used to render the page that you’re going to see. And, inside that HTML, there are–maybe–JavaScript tags or scripts or programs or whatever you want to call them. Now, what happens is that when the pages comes back, you get and HTML with the scripts. Those scripts might be going back to the server, and programmatically getting some more data. Or, they might be constructing data for use inside your program itself, inside your HTML itself. The Python cannot understand JavaScript or J Query or anything else that shows up in these scripts. So, what we have to do is we need to, somehow, make the browser run those scripts, generate the content that’s going to appear in the page, and then scrape it. So, that's a second layered step that we need to do. Beautiful Soup can't handle that, but Selenium can. So, Selenium, what it does is it emulates a browser. It pretends it's a browser, and it can get the data back. It’s also useful sometimes when a web server detects that you are coming from a program and blocks you. That can happen, and then, what you can do is you can use Selenium, in which case you emulate a browser. I see this happening less and less.
It was a more often thing a few years ago. But, then, Selenium can emulate the browser, and the web server can pretend to be Chrome, for example. So, the server doesn't know that you're actually coming from a program. Not a great idea because, like I said, doing things surreptitiously is not a great idea. But, it's something to think of. So, that's the three libraries that we're going to look at. So, let's start withBeautifulSoup4.BeautifulSoup4 is a library that, like I said, what it does is it’s an HTML and SML passer. It makes use of the tags. It creates a parse tree, and it’s a multilayer parse tree because it keeps track of attributes of the various, you know, what we really have is an outer tag like an HTML tag. We have an HTML tag, and then, that HTML tag contains, let’s say, a body, a head and a body, and the body contains a div, maybe another div. This div might contain a div. This might contain a paragraph. The paragraph might contain an A tag, you know, etc., right?
So, we get this big tree structure with multi branches everywhere, alright? And, we want to be able to dig into these many, many branches and find our content. So, what the Beautiful Soup is doing is it takes all this stuff and puts it into one giant tree and keeps track. And, remember, each tag might have attributes. So, you might have a div tag with a class attribute. Class equals something. Class equals something. Paragraph might have an ID equals something. So, they're a second layer of information at each node of this tree. We have various attributes. If you think of it, it looks just like a XML tree that we saw last week in our XML examples. So, that's side you there. There’s a very nice website with documentation on this. So, you can take a look at that crummy. That’s worth looking at. We’ll go through a few examples here in class.

Video 4 (8:04): Beautiful Soup4: part 1

Take a look atBeautifulSoup4.BeautifulSoup4 is the library that we talked about for scraping the page or actually taking the HTML from the page and finding a content on it. To use it, we need to import it, but before that, we will import requests because requests is our main library for dealing with http requests and responses. And then, we import, from bs4, we import the BeautifulSoup object definition, class definition. So, we have that stuff here, and we're going to scrape this page on, in Epicurious, right? So, again, just to, we did this last week, but let me just walkthrough this very quickly again. When we look at a web page, for example, epicurious.com, and we want to find something, we go to the find a recipe over here and type Tofu Chili, and that's what we are looking for, chili made with tofu in it. Vegan chili, so to speak. We get a page that looks like this, and if–more importantly, when we look at the URL, we notice that the URL contains tofu and chili, it contains the words "tofu" and "chili" inside it with a percent 20 in the middle.
The percent 20 indicates that there's a space in the middle. And so, what's happened is that our–the data that we entered in an input box on this page over here is now included in the http request that went to the server, and we can see it over there. So, now, if I want to get the recipes on anything in Epicurious, I don't technically have to go and enter the search box. There's no need to do that even on my browser. I could replace tofu chili by, let's say, beef chili. And, I get a bunch of beef chili stuff, right? So, rather than have to go to the home page of Epicurious, type in the keywords, I can actually go directly to that URL. So, for now, that's what we're going to do. We're going to go directly to the URL, and later, we'll see how to input the stuff. But, for now, we can go directly to the URL and get our stuff there. So, here, we go to this URL, and then, of course, we have our request response cycle. So, we use request. Get with the URL to get the response. And, once we get the response, and we should technically be checking to see whether the status was okay or not, the content was there or not. All that kind of stuff. Well, assuming that worked, then we can go and construct a BeautifulSoup object, give it the content of the response which is the HTML. And, tell it the parser that we're going to use. BeautifulSoup, essentially, uses two main parsers, LXML which is a fast parser which is similar to the XML parser we saw last week. And, html5lib which is a little bit slower but it's also a useful one. We'll see in some caseshtml5lib may be necessary.
And, once you've got that, we have a BeautifulSoup object, and we can see what it looks like by looking at the structure of that stuff. So, let's take a look at it and see how that works. So, here, we've got our two imports, import requests, and from bs4 input BeautifulSoup. We've got our tofu chili. I don't need the percent 20because, if you recall again, in what the request objects does, what the request library does is if you have a space, it'll put in the percent 20 for us. So, we can construct a URL which is not a valid URL because it got a space, but request will handle that. And then, we do the request. Get URL, and we check the status code and make sure that it actually works. And, it says yes. That worked. We got a 200.So, we can generalize this. So, what I'll do here is I'll write an input statement where we can enter whatever keywords we want. And then, see if that works. So, let's take a look at that. So, let's say I want lamb chops. I got something back, right? So, that worked fine. I can take that page, so we've got our response here, and the response contains the entire response, the status code, the content, whatever else comes back in the http response.
And, from there, we know that the content actually contains the data that we want. So, I can construct a BeautifulSoup object, and then print it. So, the prettify function over here will just make it look nice. You'll see in a second. And, this is what we get back here. It tells us this is what we're getting back in our content, and we can see this is HTML, right? We've got here the open HTML tag. There's a header. There's lots of stuff over here. And. Junk here. And, we can actually check and see whether–what I'm going to do is I'm going to go back and run this with

tofu chili, mainly because I know what I'm looking for there. Do tofu chili. And then, run this, and inside here, I'm going to look for, say, noodles. So, we get here, we can see that one of the things we've got back is recipes, food views, Chinese egg noodles with smoke duck and snow peas. It's not exactly vegan, but you know, egg noodles. But, with duck and all that, but still. It's probably got tofu as well in it.
That we got that back so, that's the URL to the recipe itself, right? And, we can see that in our page if I go back to epicurious.com and look here. And, if I do–we see that we've got spicy lemon grass tofu. There's some articles and stuff which we are not interested in. And, there's Chinese egg noodles with smoked duck and snow peas which we can, then, click on. And, it'll pull out the actual page for us. So, that's the basic process. Now, we have an HTML stuff, but of course, we want to do more than that, more than just HTML with it. We want to be able to parse it. So, let's look at the BeautifulSoup functions we can use, and there are many, many functions you can use. But, we're really interested, mainly, in four of them. The four that we're interested in are find, find all, get text, and get attribute. You can also navigate up and down your tree using these four functions, parents, parents, children, and descendants. But, we won't worry about those for our class here. We'll look at just these four. What find does is it returns the first tag that matches a particular tag that you give it. So, if you say find div, it's going to give the first div tag on a page.
Find all will return all of the–all the tags that match. So, if you say find all div, it's going to give you all the div tags in a list format, actually a results set. But, it's like a list. Get text returns the content, the actual text that is marked up. Remember, HTML is essentially a mark-up language that marks up pieces of text. So, get text returns the text that's marked up. What if our text is enclosed inside the tag? So, if you do a get text on an entire HTML page, it's going to return all the text that's on that page, right? If you do it on an inside tag, it'll return the text that's marked up inside the tag. And, finally, get attribute returns the value of an attribute. This is particularly useful for the href attribute, because if you're looking for a link, then you find an A tag and get the href attribute value. And then, it'll give you the next link that we are looking for. So, let's take a look at how this works in practice.

Video 5 (4:33): Beautiful Soup4: part 2

So, we've got our results page over here. We can do find all to get all the A tags. And so, this is going to print all the A tags. And if you notice, it gives us what's called a results set object. But then results set object really is nothing other than our–what's our list type object. So, as you can see it's a list. It starts with a square bracket, and it will end with a square bracket as well, and has various items. So, it has this title, Epicurious, and then comma. Then we can see this contains over here. This is an A tag. It starts with this. That's the open tag. So, we have an A tag over here that has a data reacted attribute, an href attribute, an itemprop attribute, and a title attribute. All these things are inside it. And then, this is the actual text that is marked up, Epicurious.
That's what you actually see on the screen. And we can probably take a look in that and see if that is the case. This is what we are seeing here. This is the tag, right? So, we are seeing the Epicurious. And that is the top of the page. That's what we're actually seeing over there. We can find just one. So, if I look at a div tag A single div tag. This is going to return just one tag for me. So, this is returning the first div tag that it sees on the page. And we can see it's a pretty big div tag. It contains divs inside it. So, this has got nested div tags inside it. But it gives me all the stuff that's inside a regular div tag. And we can see at the top of this thing, there's a bs4 element tag. Pretty much on thebs4 object itself. And we can take a look at that as well, if I do insert cell above, and I do–so, this–our entire page is inside a BeautifulSoup object. And the div tag we got back, or any tag that we get back, is in a element dot tag object, okay? So–and all these find, final, get, and get text are going to work either on a entire BeautifulSoup objector on a element

object but they won't work obviously on a results set object because a results set object contains many, many results, right?
So, that's one result. And then that's another result. So, these things are all independent results. So, I can't actually use–I can’t drill down into that. So, for example, what I could do is I could say here, let’s take this one, and I can do div tag dot find div. That’s going to find me a div tag that is inside the div tag, right? So, for example, here, we had div class equals header wrapper. And if I go into this thing here–okay–hard to see, but this class equals header is a tag that's sitting inside that particular div tag, okay? They actually are hard to see. It’s this one over here, right? So, that's the div tag that we get back because that's the first div tag that's sitting inside this entire div tag. So, we can recursively search our tree using the same commands. That’s the idea with this BeautifulSoup stuff. But we can't–what we can’t do is–I can't do all A tags.Dot find. Anything. I can't do that because it says results set object has no attribute find. Because it's like a list. And the list you can't find. You can't do that for the results set. But you can do it for each element inside the results set. That’s the main thing there.
Video 6 (8:43): Beautiful Soup4: part 3
I can also specify that I want to find certain tags that have certain CSS selector values. So, for example, if I want to find all the article tags that contain class equals recipe-content-card. Then, I can specify that directly by using, giving the tag that I'm looking for, and article in this case. And then, telling it that for this CSS selector class, I want to the value to be recipe-content-card. So, it'll link, exclude all other article tags. That is the ones that don't have recipe-content-card as a value. Note that I need to put an underscore after the word “class" here, and this only applies to class, the class attribute because class is also a keyword in Python. If we, for example, on the browser, if I remove this, I notice it becomes dark green, and that's an indication that this is not something I can mess around with. So, instead, I do this, and BeautifulSoup can figure out–can drop the underscore, basically, and figure that out. So, if I run this, I get this stuff back here which is essentially these two articles. And, I can check the length of it to see how many I get back.
And, that is two. And, I get exactly two back which is what corresponds to what I have on this page. And, a useful way of looking at a page is to look at the page itself and then look at–so what we're interested in in our page is–in this page over here is the ones that are actual recipes. So, if you look through the results, we see there's a recipe here, spicy lemon grass tofu. Then, there are some general articles which we don't care about that much. And then, there is one more recipe, Chinese egg noodles. So, just two recipes. So, we don't want all the other article. We just want the recipes. So, what we do is we look at this recipe here, and if we right click on it and choose the Inspect Element, you might need to turn on the developer menu in your browser. So, if you are on, like I’m on Safari here. I go to Safari preferences, and then choose Show Developer Menu. And, I think Chrome comes with that set as default, but you might need to do that. But, if you can do that and then click Inspect Element, it shows me that the element itself. And, we see here that the element says article class equals recipe-content-card. So, that's–indicates that’s the recipe. And, I can see that that's a recipe, and this is not by right clicking on this and looking at the value here. And, this says article class equals article-content-card.
So, this is not a recipe-content-card. So, essentially, what you want to do is you want to look at your webpage and figure out what uniquely identifies the data that you're looking for. Our interest in this page is recipes, so we want to find out what uniquely identifies recipes, and it turns out that what does that is this which says recipe-content-card. And, that's what we want to look for. So, that's what we’re looking for over here. We're saying article class equals recipe-content-card, and we get our data back from there. You can also send the–instead of–in this case, what we are doing is we are calling find all but two arguments. The tag that we're looking for and then the attribute, the selector, and value pair that we're looking for sent as the name of the selector and the value of the selector equals the value of the selector. You can also send them as dictionaries. This is useful, in particular, when you want to look for multiple selectors.

So, if there's a tag that says class equals xyz, ID equals ABC, and then something else, and then something else. And, you want to send them all, you can put them all inside a dictionary, one after the other, and, you know, the normal dictionary elements separated by commas. And, then, we are saying we only want the tags that correspond to all those different selector values that we're looking for. So, either method is fine, or it's often easier to just do this, do the class equals stuff. But, you should know this is there as well because it's not always going to work. So, that'll give us the same two whenever you're looking for over there. The next thing we can do is that given a particular tag result, we can get the content that’s marked up. So, here, for example, instead of find all, I'm using find, I’m saying, find the first article that has class value of recipe-content-card. Note that here I don’t need an underscore because I'm sending class as a string. So, it's a literal.
It doesn't have any Python meaning anyway. Right? So, then, what I can-do is I find that, and then, from the result, I call the get text function, and I get the contents of that text over there, right? So, that's how I get the marked up text. It’s got all the backslash ends and all kinds of stuff in it, but we can always get rid of that. If I just print this, for example, it’ll print very nicely. And, we'll be all set. So, we get this stuff back here. Okay. And, finally, we can get the value of an actual attribute by using the dot get which is actually the same as what we use the dictionaries to find particular attribute values in the insider tag. So, let's look, now, inside our spicy lemon grass tofu and inspect the element again. And so, we get here our article class equal recipe-content-card. And, inside that, we see that there's a A link which contains the, which has a class equals photo link. Doesn’t really matter, I guess. But, inside this class equals photo link.
Now, that'd be one this one. Class equals a view-complete-item. So, this view-complete-item in this A class over here, we’ve got the link to the recipe itself, right? So, this contains a link to the recipe, and we want to extract this link. But, this link is not part of the text of the marked up text. It’s inside the tag itself, and we want to get that link. So, what we want to do is we want to say find this A tag, get the href attribute from that, and extract its value. And, that'll give us the link to our–the next article if you want to find the actual recipe content from there, from our, from Epicurious. So, that we can do here by using the dot get function. So, let's say first what we’re going to do is we're going to find the recipe information, the entire recipe information. So, we do results page.find. Give the tag name article, and send the attributes that we're looking for, that we want the class to be recipe-content-card. So, that's the first thing we do. From that, we look for the annotate tag.
That gives us the A tag. Then, we can print the link just to be sure that we're getting the right thing. And, we think that from this tag, so, this recipe link is our thing we're looking for. So, from the recipe link, we want to get the value of the href attribute. We aren't looking for value of any attribute, but we're looking for the href attribute. And, we get that, and so, it’s going to return the URL and shove it inside link URL. And then, we print that. So, let's look at this, and that’s what we get here. So, our A tag is the entire tag. Href all this stuff. Spicy lemon grass tofu marked up. The link URL is this. Notice that it's missing the www Epicurious dot company link for it to work actually has to have http in front of it, and we'll need to add that back in. But, typically, on a webpage, you're not going to see the full link because the server knows that it's going to put Epicurious in front of it. And, we see the type of the link is a string. So, both get text as well as get return a string type. And, that's our string there. So, just to clarify all this stuff here, if I take my link URL and add to the front of it. And run this, I get a clickable link. If you notice now it’s a clickable link, right? Because it's got the http in front of it. Anything, to be clickable, has some http in that. So, that's the basic commands that we get or functions we can use with BeautifulSoup.

Video 7 (10:34): Epicurious example: part 1

So, let's take an example of how we can get data from the web. And our goal here will be to go to the Epicurious site and to get a list of recipes for a certain keyword or a bunch of keywords and for each recipe. So, we're going to get a list of recipe dictionaries. And we're going to list our recipes in a dictionary format. And each recipe, we'll get the name, a brief description, the list of ingredients, and the preparation steps. So, just to take a look at what exactly we are working with, let's go back to Epicurious and see what we get there. So, if we type in tofu chili, like we did before in our example, we get this page over here. And the page, we can see, contains a recipe for spicy lemongrass tofu and, further down, a recipe for Chinese egg noodles with smoked duck and snow peas over here. So, our goal is to go–If we enter tofu chili, we want to find these two recipes, spicy lemongrass tofu and the Chinese egg noodles with smoked duck and snow peas. And for each recipe, we want to get the description. The description is, if we look into this page here, and click on quick view, for example, we get this stuff here. No, we don't want to do that.
We want to go straight to spicy lemongrass tofu. And we go to this page over here. So, what we want to do is we want to get that link.Yikes. We want to get this link here–the link to spicy lemongrass tofu. We're going to click on it and then go to the detail page. And in the detail page, you want to pull out the description, the list of ingredients, and the preparation steps. So, these things, we want to pull out from there. And finally, put all this stuff in a dictionary where we have every recipe, in this case, spicy lemongrass tofu and the Chinese egg noodles, with the name of the recipe, with the name of the recipe, the description, which contains all that stuff, while traveling on a train and all that kind of stuff, and then the list of ingredients, that is lemongrass stocks, soy sauce, etc., and the preparation steps. And so, we get a list of these dictionaries, and that becomes our data. And a list of dictionaries, which as you can think, could be easily saved in JSON format for distribution or perhaps in a null SQL database or some such thing. So, that's our goal here.
So, let's see if we can go about doing this. So, the first thing we want to do is we want to write the function itself. So, I actually have on the iPad or notebook that I've given you guys, the solutions are already there. But let me walk through the steps on how we build–just to get some experience with this process. So, the first thing you want to do, of course, is figure out what the function is called. So, we know that our function is going to get recipes. And we're going to first write a function that returns the list of recipes. And on the recipe page itself, we have–the list of recipe page itself–that is when we typed tofu chili and get the list of recipes back, we've got the name of the recipe, the link of the recipe, and the recipe description. So, we can pull that information out from that page itself. And then we'll use the link to open up the next page where the detailed recipe detail is and get the ingredients and the preparation steps. So, let's say we start with a function, get recipes. And what this does is, given a bunch of keywords, that's the argument, it's going to return a list of recipes. So, what we'll do is we'll say recipe list equals list, an empty list because we haven't sent anything yet. And our goal in the end finally is to return this recipe list.
We have to create the recipe list and then return it. So, we have our function template in place. So, this is a good way to start usually, get a function template and work with that. So, and we can test it out. You know, it's not going to do anything, but if you want, we can't test it out. So, let's just test it out. What the heck, right. So, if we do this and then type get recipes and give it a bunch of keywords, like tofu chili, it doesn't return anything it'd return an empty list actually because we have created an empty list here and we're returning that. So, now what we want to do is to actually go to the page, open it, and get the list of recipes. So, that means doing the http request response cycle. So, let's start with that. For that, we'll need to import requests.
As a general rule, it's a good idea to do all your inputs inside the function itself so that your function is self-contained. You can move it from one program to another, and you won't need to have your program handle the imports. So, we import requests. And from bs4, import BeautifulSoup. So, we've got those things. And then, we do the request response cycle. So, we could say response. Well, we first construct the URL. So, the URL, we know, ishttp://www.eipcurious.com/.That's the base URL. And from our experience at looking back at

this thing here, we can see that that the page itself has the URL right in it. So, we've got here the keywords right in them, tofu, and percentage 20 chili. And percent 20 was our–was the browser's way of saying, you know, substituting a space inside the URL because URL's can't actually have spaces. So, we can reconstruct that hereby just adding our keywords. We don't have to worry about the space itself because the request library will take care of any spaces when it sends the http request. So, we do this. And then we've got that. So, now we can get our response by calling requests. Get. Response equals requests. Get URL. And we want to make sure that our response actually is successful because it could be the mistake. The URL could change, for example. Perhaps Epicurious decides to rewrite their search URL and instead of calling it search, call the keyword search or something, and then this is going to fail. So, then it's a good idea to check to see whether we've actually succeeded in this.

So, we have a couple of options. One is we could put it inside a try except so, it catches an exception, if necessary, or we could just check the status code. It's better to use a try except. But here, I'm going to just use a status code. So, if response.statuscode equals 200, which is good. And, in fact, what we'll do is we'll say if not response.statuscode because we don't want to–if we do response.statuscode equals 200, then our–else will naturally be indented and we want to reduce the spaghettiness of our code. So, equals 200, then we return the empty list. And we don't need an else now because, obviously, if we get to this point of the program, then it's obviously going to have our correct status code over there. So, once we've got that, now we can put a try except around this just in case our BeautifulSoup fails. So, we'll say try, and we'll convert our page into the stuff returned into BeautifulSoup response.content.

Remember, response.contentis going to actually return the content of the page and give it our parser. LXML is the default parser. You don't need to give it, but if you don't, you will see a lot of pink stuff on your page, and that's not going to be very nice. So, we got this. So, we've got that. And now, what you want to do is we want to figure out our recipes in the page. And we saw earlier when we were testing this stuff out previously that if we find any tag, article tag with a class attribute equals recipe content card, then that contains a recipe. So, what we can do is we can go here and say get all our recipes. We could do results_page.find_all and give it an article tag. Give me all article tags where the class–and remember–class needs an underscore, equals recipe content card. So, that's going to give us all that stuff.

So, just to make sure we've got it correctly, what I'm going to do is I'm going to print. And this is a good idea, you know, when you're developing something to keep testing it out to see whether it works or not. I'm going to print recipes and run this. Okay, that's not going to work because I need an except, right. Except. So, now I do this, and I get an empty list that's not so good. It's a good idea to test everything out. So, let's test this out. And I do that, I do this, I get an empty list back. So, now after as I said, why am I getting an empty list back? I can see here why I'm getting it because I forgot the word "search" over here. But in general, what you could if you were looking for something is to print steps in the middle or run the debugger or something like that to figure out where you went wrong. So, let's try that again. And now, we get our recipes back. So, that looks good.

Video 8 (9:21): Epicurious example: part 2

Now we notice that in our recipes in the article class tag, we have here–a description, which is over here, editor's note blah, blah, that kind of stuff. And we can see that's sitting inside a paragraph herewith class equals dek. So, we can always get the description by looking for a paragraph tag where the class equals dek. We also see and we saw this before that the link to the next recipe, to the recipe detail page is inside an annotate tag. So, there's an A tag over here. And that contains the link, and it also contains the name of the recipe. So, with these two things, by finding an annotate tag, the first annotate tag in our recipe content card article, and the first paragraph tag that has a class equals dike can get the name, the link, and the

description. So, let's add these three things to our setup here. So, I don't need this anymore. I can get rid of that. So, I could say name, recipe name equals. And in here, we do–Actually, for each recipe, right. So, for recipe and recipes because we have many recipes. We want to take for every recipe we want to do this.
We will say recipe name equals recipe. Find the A tag. And from the A tag, we do get text. And that should return us a text of the recipe. And then we can get recipe link equals recipe.find. Again, we find the A tag. And from the A tag, we get the value of href, the href attribute. And let's test this out just to be on the safe side because we all make mistakes. Recipe name, recipe link. And I run that. And I see that we get the recipe name, spicy lemongrass tofu, and the recipe link, recipe name and the recipe link. Notice that our recipe link is not at this point a clickable link. So, we need to add stuff for that. So, here, I can add that in the front. I could sayhttp://www.epicurious.com.And we notice that it starts with a slash here. The slash is already there. So, I don't need to do that. And I add that to this and test it out again. And now, we get clickable links, okay. So, the next thing we need to do, of course, is to get the description.
And we saw that was in a paragraph tag so, with the class equals dek. So, recipe description equals recipe.find, the paragraph tag, and class underscore equals dek. And once we have all three of them, I can now append that to recipe list. So, instead of doing this, I'll say, recipe list. Append, and I want tuple with the three things, the name, the link, recipe link, and recipe description. And that should be it. Let's check it out. So, I run this. I run that. And sure enough, I get my list of tuples back. And each tuple contains the data that I want, and it's got all sorts of stuff in it. That's great. Okay. So, the only thing, if you notice, is that we've actually, in the description–we forgot to get the text because we have here p class equals dek and all that stuff.
So, what I really want to do is I want to say here .get text and run it again. And now, everything works really nicely, alright. So, we've got our list of recipes with the links in them. So, the next thing I want to do is to go and open and the page and get the list of ingredients and get the list of preparation steps. So, we're going to get back–we're going to pass, write a function that takes an argument, the link–recipe link itself. So, let's say we call it get recipe info, and we give it the argument recipe link, which is the link that we're going to get from our get recipes that contains the link. So, we'll go through the list of recipes. And for each recipe, we'll take the link and then call this function. And this function will return a dictionary that contains the ingredients and the preparation, okay. That's the goal here. So, we do this, and this should be pretty straightforward.
But let's take a look at it. So, the first thing you want to do, of course, is what the function returned. It returns a dictionary. So, let's create an empty dictionary. We'll call it recipe dict equals dict. And again, we want it to be self-contained. We are going to send an http requests, get a response, and construct a BeautifulSoup page. So, we need all these things to be imported. So, import requests frombs4, import BeautifulSoup. And let's put it inside a try except. So, we get a try and response equals requests. Get recipe link. Hopefully, if our get recipes work properly, we construct the link correctly, this is not going to be a problem. But it's always safer to check, okay? And check the response code if not response.statuscode equals 200, return the empty dictionary. Otherwise, we construct our results page. And so on. I'm not going to go through all this.
Let's just take a look at the function itself because this is pretty straightforward. I've just walked through the steps. So, we've done that. So, we've got our recipe dictionary. We've done our imports. We got our page back. If we get to this point, it means that our request response cycle worked correctly. And if we get to this point, we constructed our BeautifulSoup object correctly, right. We've got a HTML in our BeautifulSoup, HTML instruction on BeautifulSoup. And now, we want to find a list of ingredients and a list of preparation steps. So, we can go back to Epicurious and click on this recipe here and scroll down and see whether we can find the list of ingredients, what identifies the list of ingredients accurately. So, I go there. I click on inspect element.
And I'll look at the element itself. So, here, we see that if I go down into this, there's a div class equals recipe content, recipe summary, ingredients info. It's got a h2 there, ingredient groups. So, that may be one thing to look at. Inside the ingredient groups, there's an ingredient group li,

and there's a URL where the class equals ingredients. And then we see that every ingredient itself is inside a litag, which is a list-item tag in HTML. And that li tag has a class equal's ingredient. And this doesn't matter whether we're in one group or a different group. They all have the same structure. So, what we can do for simplicity is if you want to get ingredients by groups, we'll need to go a little bit up to the class equals ingredient group and then pull data from there and then organize it. But what we're going to do is we'll just find all the li tags that have class equals ingredient or itemprop equals ingredients. Doesn't matter which one. And that will give us a list of ingredients. So, that's what we'll do here.

Video 9 (6:55): Epicurious example: part 3

Go back to this. So, we say here for ingredient in result page dot find all li every list item tagged with a class underscore equals ingredient. And we get a list back or result set back that contains all the ingredients. And from each element of that, we get the text and append it to the list of ingredients. At the end of this we should have a list of ingredients. Similarly for the preparation step, again, if you go back here and scroll down, and we see there are steps one, two, three, four. So, I'm going to just again right click on this and do Inspect Element, and it says class equals preparation step and we find each preparation step is in a list item with class equals preparation step. So, we can just pull that out exactly like we did with ingredients. So, now we say for preparation step in result page dot find all, and we get every list item that has class equals preparation step.
And then extract the text from it. And just because there's a lot of text–and we can take a look at this, actually. So, if I print this over here. Let's just print that just to take a look at it. So, I do this. And I need to run that. You notice that we're getting this, a lot of white space in the beginning and, you know, it doesn't look really very nice and we want to clear that. So, we can use this very handy function called strip. What strip does is it strips all white spaces in the beginning or at the end of a string. And in fact I'll strip the backslash N as well. And that makes our results look much nicer. So, if I do this and do this, so, now we get everything, ingredients, all this stuff, and the preparation steps no longer have the white spaces in the beginning. So, we've got that. And then once we do that, we want to create the dictionary that we're going to send back.
We already created the empty dictionary. So, what we do is we add the ingredients–we add a key called ingredients and give its value as ingredient list. And add another key called preparation and give its value as prep steps list. So, now we have a dictionary that contains two elements, the ingredients and the preparation, two keys, and each key has its value there. So, what we want to do finally of course is to add the name and the description to this dictionary, and return it. So, what we can do then is write out one function that runs it all, essentially, and we're going to call that get all recipes. So, what that will do is it'll call get recipes. And remember, get recipes returns that list that contains the name of the recipe, the description, and the link. And then–so that's what'll recipes is. And then from all recipes, we'll iterate through that, take every recipe in turn and get the dictionary that contains the ingredients and the preparation steps and pass to that function get recipe info, pass the URL. So, recipe one is the URL, right, the second element in our trupel.
And then we get the name, which is recipe zero, and we add that as a new key into the dictionary returned by get recipe info. And we get the description with recipe two and add that as a new key into our dictionary. And then append that to the list of all results that we setup initially as an empty list. So, we do this and run that and we get a list of all our recipes that contain the name of the recipe, the description, the preparation step, and all that kind of stuff, can test this out to see what happens if I give it a non-existent set of recipes. So, if I go here and I type get all recipes, and give it the keyword nothing, I get an empty list, because there's no recipes returned. And we can check that out. If I go back here and I go to epicurious.com. I type nothing, I get no recipes. I get gallery, article, menu, article, article, but there are no recipes at all, right.

So, we don't care about articles and stuff like that. So, that's pretty much it. That gets us our solution. And the only other thing that I should point out is that if in some cases the recipe description is missing in a recipe. So, if there's no recipe description, then what's going to happen is that when you try to find the paragraph with the class equals dek, this one here, you're going to get an exception? And when you get an exception, your program is going to die. So, what you want to do is you want to put it inside a try except here. And if we don't get a description, then we'll get an exception over here, couldn't find anything that matched class equals dek. Then we said with recipe description two, nothing, an empty string. Another ways we return the recipe description itself. So, it's a good idea to check and see in your data whether there's what the scope of errors is or exceptions is and then cover for that. Because typically, when you're doing data analysis, you're working with millions and millions of data items. Any one of them could go wrong.
A missing piece of data or a poorly–you know, a character that you're maybe converting from a string into a fringe point number and one data item has a letter inside it or a misplaced space or something like that. And what will happen then is that you'll be run through the first million and then you get this bad data item and your program crashes. You don't really want that. So, it's always worth thinking, okay, what can go wrong, and then protecting yourself against that. And you can look around and see what's there or even just use your imagination and figure that out.

Video 10 (6:20): Log into a web server: part 1

So, sometimes, we want to log into a server to get data. We might need to log in because we're protected. For example, you're maybe bank account or a site like the New York Times. You want to look at the archives, and you want to pull data from there. You need to log in or have an account with them before you can do that. You know, those kind of things. So, they're the wall, a pay wall, perhaps, or some other kind of wall that separates you from the data. So–and you need to log in for that. So, logging in actually involves sending data, sending your username and password to the server. And, it requires filling in a form and sending the form data to the server. So, let's take a look at that, and that'll actually–it a good example because sometimes, you need to send a form anyway. Like, for example, you want to search a page, and you want certain keywords or you want to have a certain search criteria that you want to set. So, rather than having to go to the end page, you can write a program that sets the criteria.
And then, sends those criteria to the server and gets the results back. So, being able to fill a form and send that to the server is actually quite useful. So, the form that we'll fill is a login form. So, let's take a look at how we look into web server, and what we're going to do is we're going to use Wikipedia as our example here. And, what you might want to do is open an account with Wikipedia if you don't already have one and get a username and password just to test this out. I'm not going to share mine with you because I think Wikipedia would be very unhappy if, you know, millions of us log in on the same account and start messing up. So, that's the idea there. So, my, what I'm doing here, and this is not a bad idea when you're logging into–when you're writing a program that involves logging into stuff–is to put things inside a file. And, that file can be a binary file, for example, that nobody can read. Or, maybe an encrypted file or something like that. And, what you can do, then, is you can read the contents of the file using your program. And then, you know, people don't actually see the username and password. But, it gets used anyway.
So, for example, if you 'rewriting a web application and you're logging in from your server, then the client won't see the username and password. It's sitting on your server separately. That's the idea there. So, I'm going to read them in over here, and I will. This will hopefully work, but we won't test it out. So, here, I write a bunch of code over here which says open wikidata.txt which is my file as f. And, again, I think we did this before that what this says is that f is a file that'll be opened only in this block over here. Only in the web block, and once the block ends, the file will

no longer be accessible. And then, I'll read the username as contents–from contents zero of the file and password from line, from line one. So, that's line zero and line one. Okay? And, we’re splitting the file–reading the entire file here and splitting it on backslash n.
So, end of line characters because the first line contains the username, and the second line contains the password. So, I read that, and hopefully at one point, we will know when we'll do that. Now, the next thing you need to do is to figure out what the page looks like. So, let's see what we can do with that. So, the page that we want to go to for Wikipedia's login is this this page over here. So, let's go there. So, this is the Wikipedia login page, and we can see it has here a username input box and a password input box. And then, it has keep me logged in for 365 days, a checkbox and login. And, if you look at the page contents itself, so, let me right click here and see the source. We’ll see that the source contains all this stuff is going to be inside a form.
Yeah. So, here we have the inputs. Remember, a form contains input fields, and the input fields are where the data actually comes in. So, here, we find that there's a bunch of input fields over here. And, here's the username, div4wp1.Input for here. So, this is the username, wpName1, and the name of the field. So, what you want to do is you want to look for the input tags, all the input tags, and look for the name of the field that the input tag is capturing. Because, remember, this is the name that's going to be used by the server to get the value that you send it. So, we need to know that. So, we find the input tag with wpName here, and we actually find there’s another one for password as well. So, that's wpPassword over here, input five wpPassword. And then, there's a lot of other stuff as well. There are actually quite a few input fields.
There’s an input field called wpEdit Token, and that's hidden. Hidden means that we don't see it on the page, but it is there. And, it has to go back. If it doesn't go back, you won't log in. So, you really need to find all the inputs that are there on the page and figure out what they–what values need to go back, whether you’re entering that value or not. And, more important than all this, there is yet another field he recalled wpLoginToken. This one over here. And, that has a value. What Wikipedia does is, it sends this value each time you open the page. So, if–I take a look at this, for example, it ends with db006.And, if I reload this page and go down and take a look at it. Where is it? Here. We see now it's db08E.It's a different token. So, every time we log, we refresh the page, we get a different token value. That’s their way of, you know, some kind of security. This making sure that the page that you're logging from is a page that they gave you, not a page that you created yourself. So, it's a kind of security there. So, you want to figure out all the inputs there, and I've done that already here for us.

Video 11 (8:48): Log into a web server: part 2

So, we see that there are a bunch of inputs. So, you put that inside an object that contains all this data. So, here, I'm going to create a dictionary over here, and I'm going to call that payload. And, in the payload, I’m going to put every input name. So, for every input, we take the name of that input and use that as a keyword in a dictionary, and we give the value as the value of that key. So, we have wpName. That’s the username, and that value comes from the file that I just read. Your wpPassword, and that’s a password. And, that also comes from the file that I just read. There’s a login attempt value, and that has a value called login. And, that, we get from the page itself. There’s Edit Token, and that also we get from the page itself. A title, and that also we get from the page itself. And, authorized action, and we get that also on the page itself. Force, and that's empty, so, we don't care about it. WpForce Https colon one.
That means it's forcing us to use https rather than http, and we send that with a one. And, from http because we are sending it from an http file, so, we set a one there as well. And, finally, we're going to add one more field called the wpLoginToken, and this, like I said before, is a value that is provided by the server. So, we need to extract this from the page that we get from the server when we connect to this, the login URL, this one over here. Okay? So, need to extract

that. So, for now, I'm just putting a place holder, but it's not actually there in my dictionary. So, we've got that. So, I run that. And now, I want to get the value of the login token itself. So, I ran a small function over here that says get login token from a response, http response, a request response thing. And, to extract that, I create a BeautifulSoup object, and in that object, I find an input field which has a name attribute, and the value of the name attribute is wpLoginToken.
And then, I do a dot get and get the value of that attribute which is pretty much what I want here. We can look over here. We see that this is in this line over here. We have input name equals wpLoginToken, and the value equals blah, blah, blah. So, we want to get the value that's associated with this attribute. Right? The value attribute. And, that's what we do.Dot get, remember, gets the value of a CSS selector or more generally, you can think of it as an attribute of a HTML tag. And, we return the token. So, we have this function. We write that. Now, the next thing we want to do is we want to send our login information, and we want to stay logged in into the website. So, it's not a–into the web server. So, it's not just a question of sending the page with the data and then, that's the end of it. That won't work for us because what we send the data, a login data that goes to the server.
The server logs us in, and then after that, we stay logged in for every subsequent request. And, the way that works is that the server sends back a session object that contains, you can think of it as containing the fact that we are logged in. And, that little object will go back and forth with every request after that so that the server knows who we are, that we are the person who logged in initially. The same person. So, we don't have to keep logging in every time we make a request, because then, you’ll get nothing done. We’ll be just logging in. So, we need to–what we need to do is we need to setup what's called a session object. The request library contains a function called session which creates a session object, and that's going to keep track of the session data that comes back with our http response. And, it'll send that along with the http request back to the server, and when the server returns something, that’ll send back a session on it again.
And, that way, we can think of it as, like, a relay baton. That baton is going to get passed back and forth every time we send a request or a response. Or get a response, and that ‘show we keep track of who we are and this kind of thing. The server keeps track of who we are. So, we set up a session object. So, we're going to set up a session object, and we're going to say the session object is valid only inside this block. We don't have to do that, but it's a good idea because you don't want a session object to be valid after a certain point because your program might be serving somebody else. And, you might want to get some data and then stop the access to that session. So, it's always a good idea when you're accessing an external resource to put it inside with a block so that its lifetime is restricted. So, here, we set up a session object, and now, rather than saying request.get, we actually use the session, so, we get a session object s, and we're saying from this session get the login page.
From the login page, so this is the page we got earlier, we use our login token function that we just wrote to get the value of the login token and add it to a payload. The payload we've already created over here, right? We created the payload over here, but we're going to add this new key and its value to the payload so that we have a complete set of data to send with our http request. Now, we send our login request. Logins are a post request that’s most secure than a get request. So, generally, when you ‘resending, for example, search parameters or something, you’ll probably do seat rather than s.post. But, if you're sending login information, you need to use s.post. So, we send a–we send a post request, and in the post request, we give our same login page.
And, we give it the data that we want the request to do. So, this is the data that is getting sent along with our request. And, that data contains all the stuff that we've put here, the name, the password, the login attempt, and all the way down to the login token. So, once we get that, we can check and see if we are still logged in. To see we are still logged in, we–in Wikipedia, once you log in, you get a page that contains, you know, various stuff. And, one of the things it contains is a watch list. And, what the watch list is is a list of recent articles that you, as a user,

have indicated that you want to see if someone has changed it or not. So, this is a way, if you don’t have a watch list, it means you're not logged in. If you have a watch list, it means you're logged in. So, we are going to get that page over there, and once we get that page, over here, we send a get request. And, we're sending you the session. Remember, so, what the session is going to do, going to send our baton that says that we're logged in back along with that. So, Wikipedia knows it’s us, who we were. Right?
And, we look at the content, and in the logged in page there, the watch list, there’s a thing called nw-changes list which tells you the changes. So, we'll get the text of that, and see if that works. So, now that we’ve set everything up, we can actually test this out. Before we do that, we want to make sure that we are–have our imports ready. So, I import request, and frombs4 import BeautifulSoup, and that should be it. So, let's test this out. And, we run this, and it’s going to go there. And, it gets back page, and it tells us I had put big bang and the main page as our two things to check on the watch list. It’s just a dummy thing there, and it says that, yes, we’ve got a change on 17th May, and today is 18th May, just for your information. With the talk main page 22 to 25by the user called Bencherlite, and big bang by River torch on 15th May. And, that's pretty much it.
So, that tells us, yes, we are logged in, and we can do all the login kind of stuff with that. So, this is what you would do if you, for example, want to log in to the New York Times and look at the archives or you want to log in to some other data provider who requires you to log in first and before you can actually get data from them. And, with that, we end our getting data part. So, what have we done? We’ve reviewed how to deal with JSON data and LXML data. Sorry. JSON data and XML data. We’ve looked at how to send http requests and deal with the response that we get back. And, if we are scraping web pages, then we can use BeautifulSoup to scrape the web page itself and get the data out of the web page. Scrape the HTML and get data out of it. If there's JavaScript on the page, then you need to look at a library called Selenium which is not, we’re not covering in this class. But, it's worth knowing that it exists. Thanks.