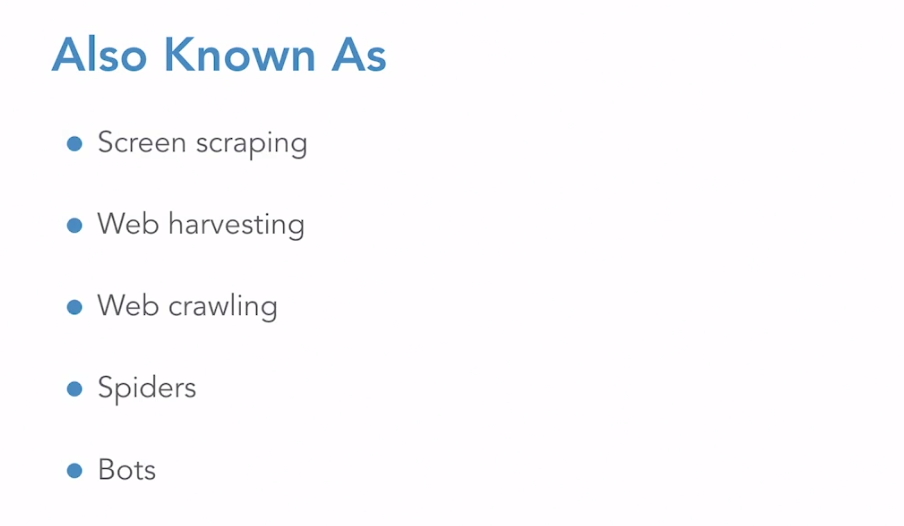
# **Web Scraping with Python**

### **How to learn to stop worrying and love the bot**

Selecting transcript lines in this section will navigate to timestamp in the video

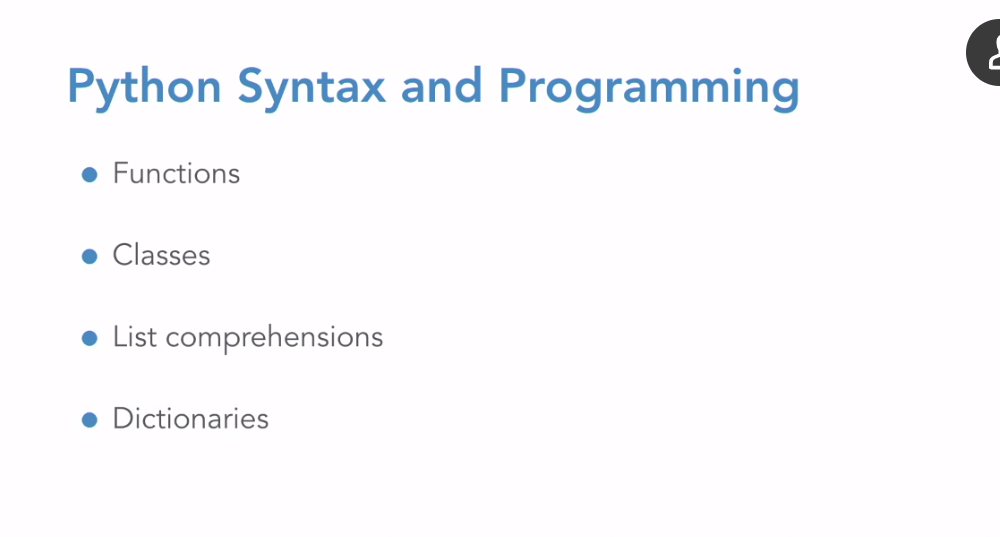
- [Ryan] Are you tired of clicking on web pages one at a time and having to read them with your eyeballs? I was, until I discovered web scrapers. I built my first web scraper in order to create a database of drink recipes. Quickly, my love of potent potables was eclipsed by love of powerful programs. Soon I was web scraping full-time, building bots that pretended to be human, figuring out how websites were put together so I could break them apart, even writing scripts to beat CAPTCHAs. It was so exciting being able to use all these different disciplines of programming and computer science and funnel them often in creative ways into web scraping. Hi, I'm Ryan Mitchell and in this series, I've distilled some of the more powerful and fundamental techniques **of web scraping into easy examples using Python's popular Scrapy library**. So whether you're doing app testing, research, data science or just want a database of exotic drinks like I did, let's get scraping.

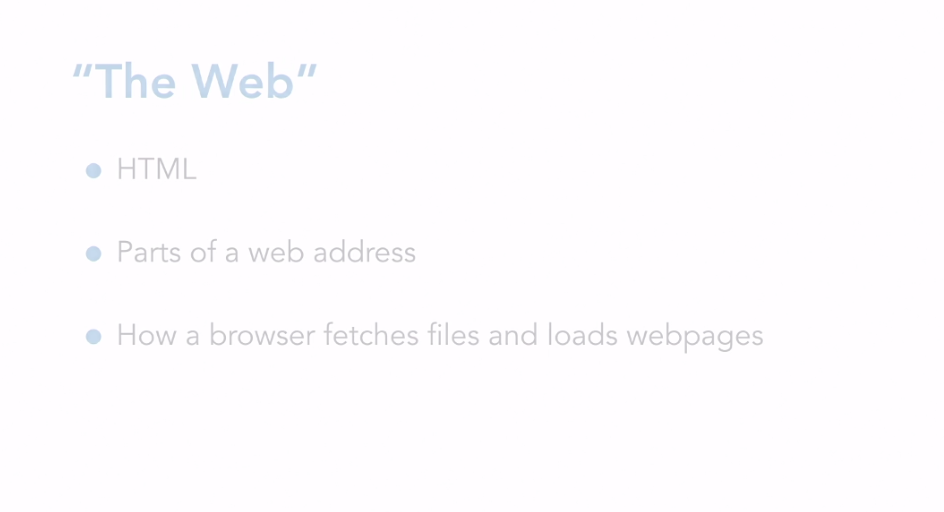


### **What you should know**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] So, this is not a Python course. It's a web scraping course that happens to use Python, but it does teach web scraping. So you should be familiar with the tools of the trade beforehand. You don't need to be a Python expert. I'm not sure if I'm a Python expert sometimes, but you should be familiar with Python syntax and programming concepts involving functions, so defining functions, arguments, calling functions, the usual classes, **so that's classes, objects, methods, defining new classes and using objects in Python. We do a lot of extending classes in almost every section and overriding the parent methods in a couple of sections, and other Pythonic things. So list comprehensions, dictionaries.** I love list comprehensions, really one of the best parts of Python, you should love them too. Nice to have regular expressions. So we do use them in a couple of cases, but feel free to learn them after the class too. You should also have some familiarity with the web. I'm going to put that in quotes. It's a little more nebulous, but listen, it's hard to automate something you don't know anything about, right? So HTML, you should know what HTML is. Know the parts of XML, tags, attributes, values, you know about IDs and classes in HTML. You should know the parts of web addresses, domains, paths, some notion of query parameters in HTTP. How a browser fetches files and loads webpages. So we're actually going to be discussing this one quite a bit and teaching it, understanding it as a huge part of web scraping. But if you're sitting there going, "What, web pages are made out of files?" Like you may want to hit the books a little bit and then circle back. Other than that, well, web scraping is a multidisciplinary field. We're going to be touching on a lot of things, learning a lot of things, you don't have to know or understand all of them right now. Whatever other skills you bring to the table, I'm sure you'll be fine.



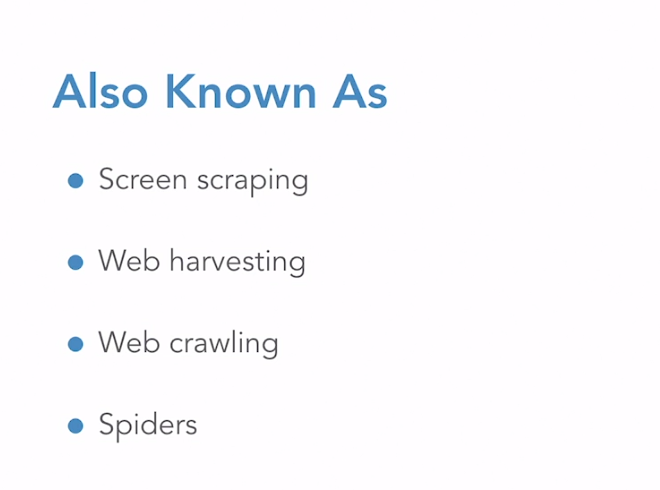


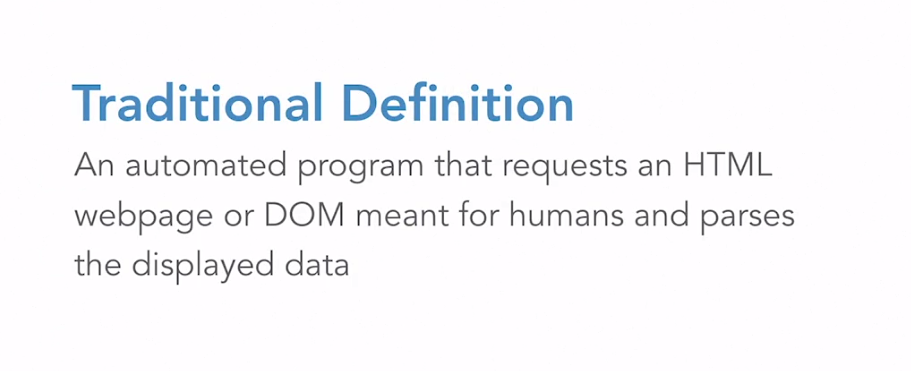


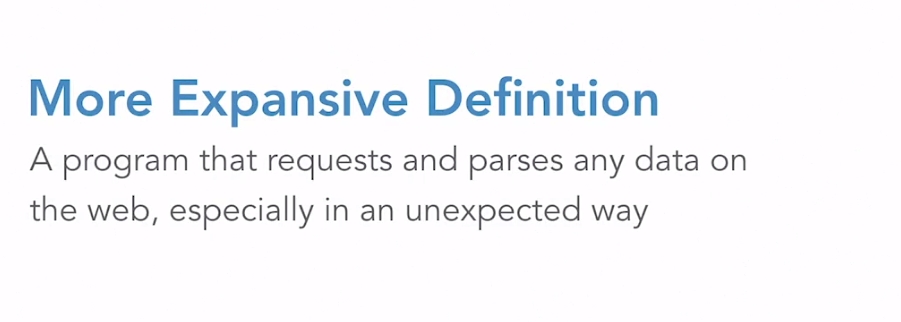
### **What is web scraping?**

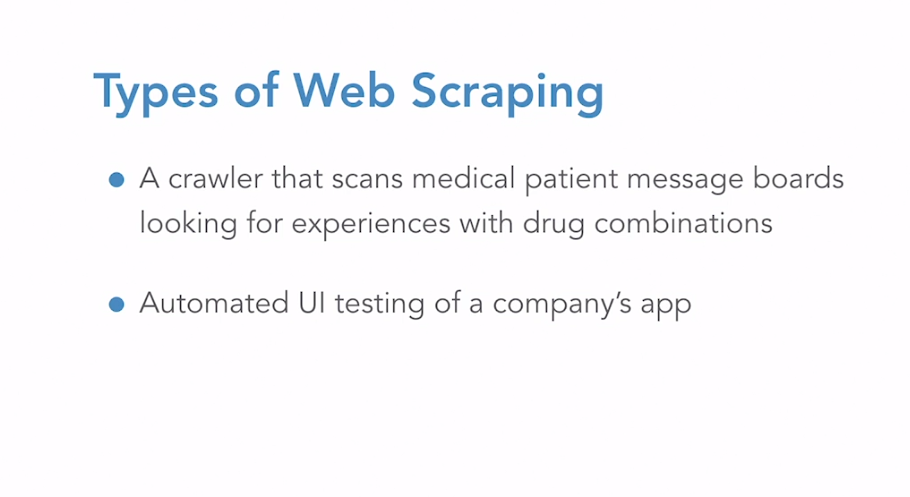
Selecting transcript lines in this section will navigate to timestamp in the video

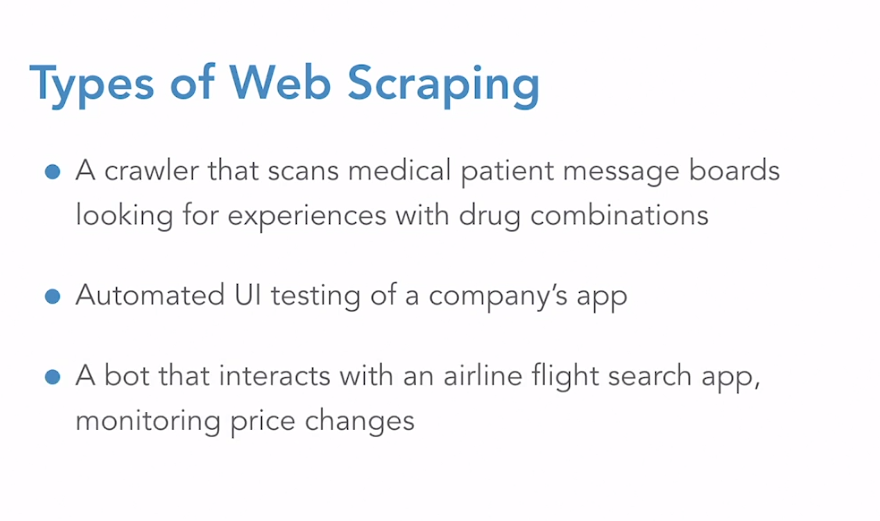
- [Instructor] So what is web scraping? Let's start this off by going over some web scraping terms. You may have heard web scraping being called many things, screen scraping and web harvesting are what I think of as older terms for web scraping, a little bit out of date. Web crawling or web spiders or spidering all refer to moving from one webpage to another via links. So, you can scrape data from a single page, or you can crawl across multiple pages, scraping data from each one. Of course, sometimes all these terms are used interchangeably, but in this course, we'll use the term scraping for getting data off of a single page and crawling specifically refers to the act of moving from page to page. But we'll get more into those details in the second chapter. So bots or web bots refer more to automated interaction with websites or web apps. So this interaction, may be with the goal of scraping data or with the goal of crawling from page to page, but it's the bot that performs this interaction. And the concept of interaction or automation of a web app is something that we'll explore more in chapter three. So you may be familiar with the traditional definition of web scraping that goes something like, an automated program that requests an HTML webpage or DOM meant for humans and then parses the displayed data. So I find this definition to be a little bit limiting. Not all data is HTML, right? Browsers are useful tools and their DOM can be a great interpretation of this web data that sometimes we don't really need or care about it. Is all data meant for humans? Sometimes it is sometimes it isn't. Sometimes it's somewhere in between, right? Let's look at a definition of web scraping that I like to use: a program that requests and parses any data on the web, especially in an unexpected way. So this doesn't necessarily have to be automated. Of course, many web scrapers are. This could be any data, text files, videos, could be HTML. I think the essence of web scraping, when most people talk about it, is this sort of unexpected or almost hacker-like use of the data. Maybe this data was even meant to be parsed by a computer program, just not your computer program. So web scraping is a lot of data detective work, really getting down and dirty, and understanding what's going on behind the scenes in order to repurpose the application for your own uses. Some uses of web scraping might be a crawler that scans medical patient boards looking for experiences with drug combinations, so lots of big data collection, natural language processing classification. You could have automated UI testing of a company's web app. Here, obviously you have permission of the people who own the app, but the app's code wasn't designed to be interacted with by a bot. So you're not scraping the data and parsing it in a traditional sense, but you're putting the data into a database. You are logging test failures and successes and that's a type of data. You can have a bot that interacts with an airline flight search app, monitoring price changes. So here, there's a similar app interaction. It's logging data that gets uncovered at the end of that interaction. You probably don't have the permission of the airline company. But what if you're collecting that exact same data, only this time, your bot just makes requests to the airline's public API. Is requesting JSON data from a URL somehow fundamentally different than requesting a web application page meant for humans? And the thing is, there's no real answer to that question. The definition of web scraping is totally arbitrary. It's this kind of nebulous field that encompasses many skills and practices, so application security, networking, data science natural language processing, law, data architecture. The bad news, this course will not teach all of these fields. But what I do want to teach is the sense of how to look at a web scraping problem and break it down correctly. You want to figure out the first step and the second, and if you do need to bring in outside help, say from a data scientist, you have a neat, nice contained little problem for them that slots in well with the rest of your web scraping project. And I do want to leave you with this sort of apocryphal, probably apocryphal quote, from Michelangelo about how he sculpted the statue of David. "It's easy. You just chip away the stone that doesn't look like David." So really what web scraping is, is being able to look at a website, a huge collection of links and media and ads and garbage and cut through all of that and see only your database. You want to chip away everything that isn't your data. And the first step to doing that, of course, is learning how the internet works over in the second video. So I hope you will come join me.

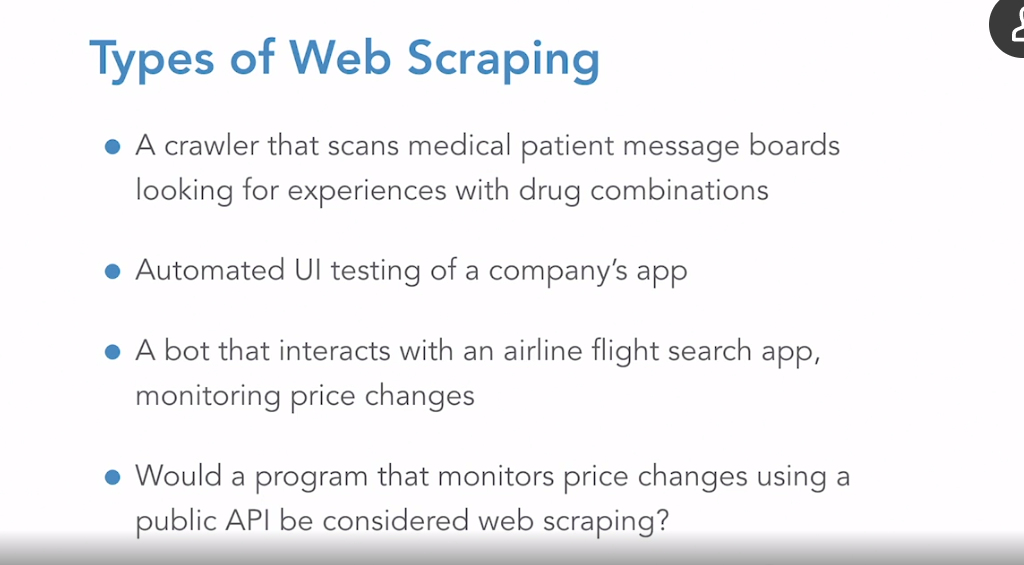


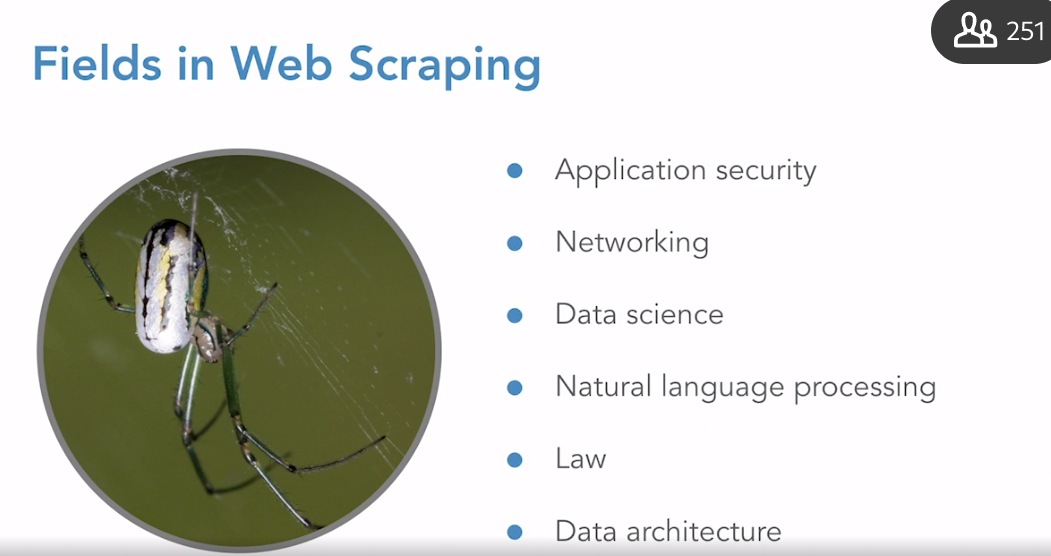


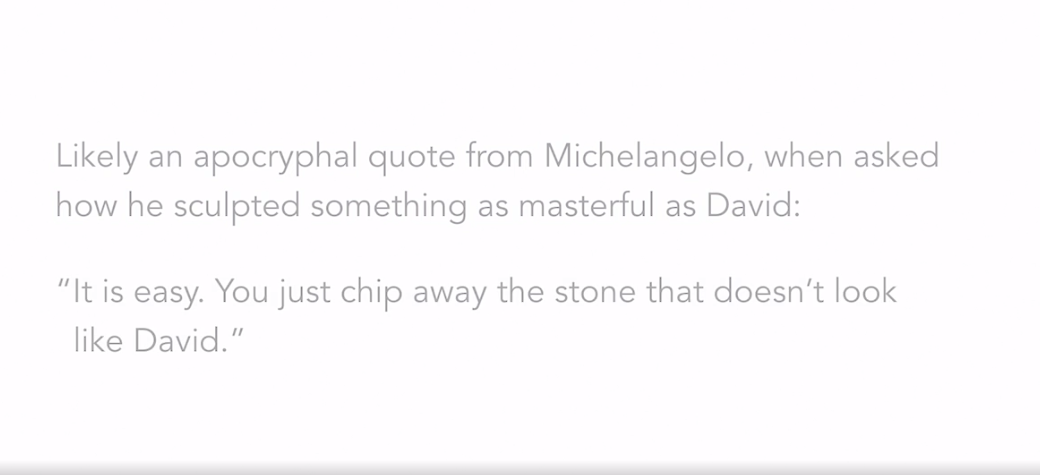








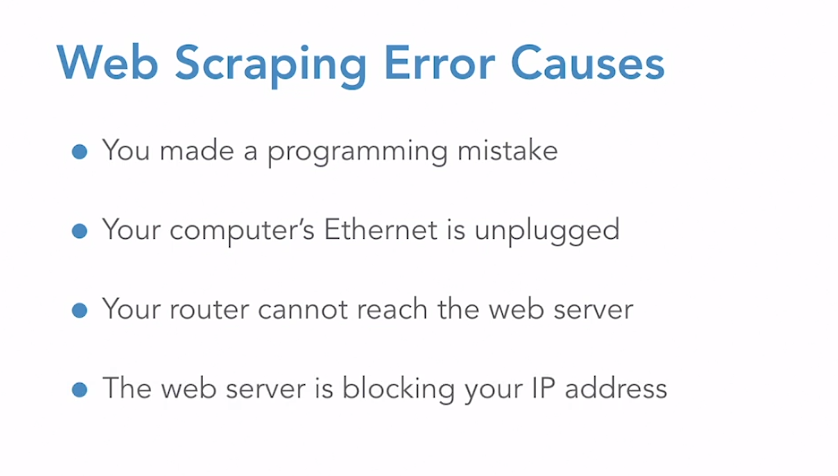


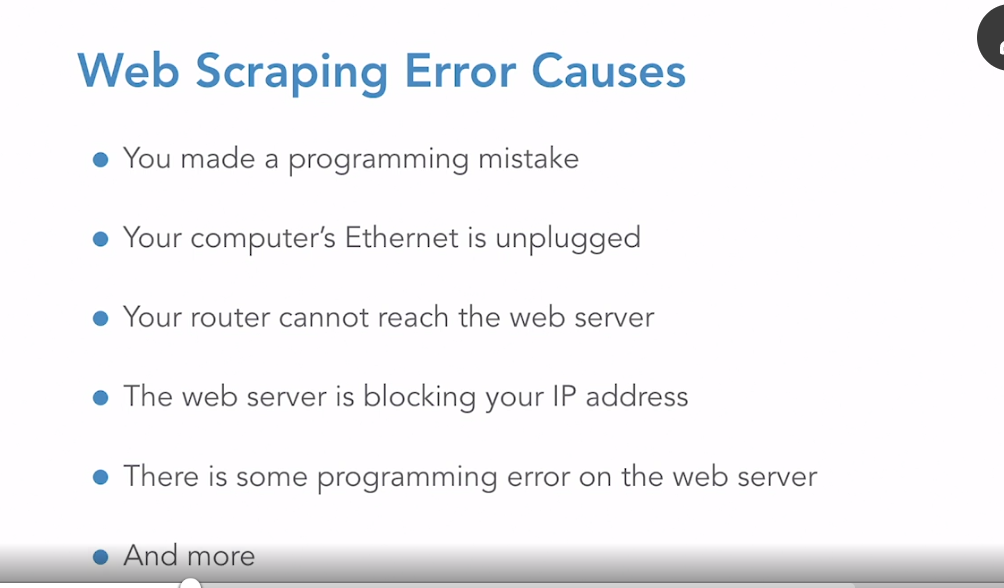


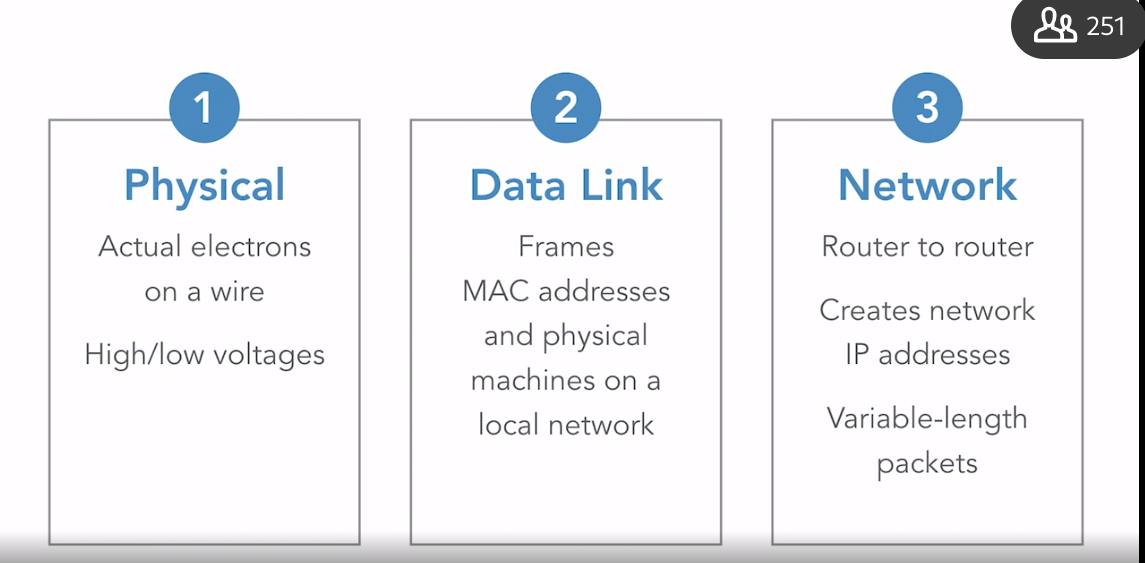
### **How the internet works: A brief summary**

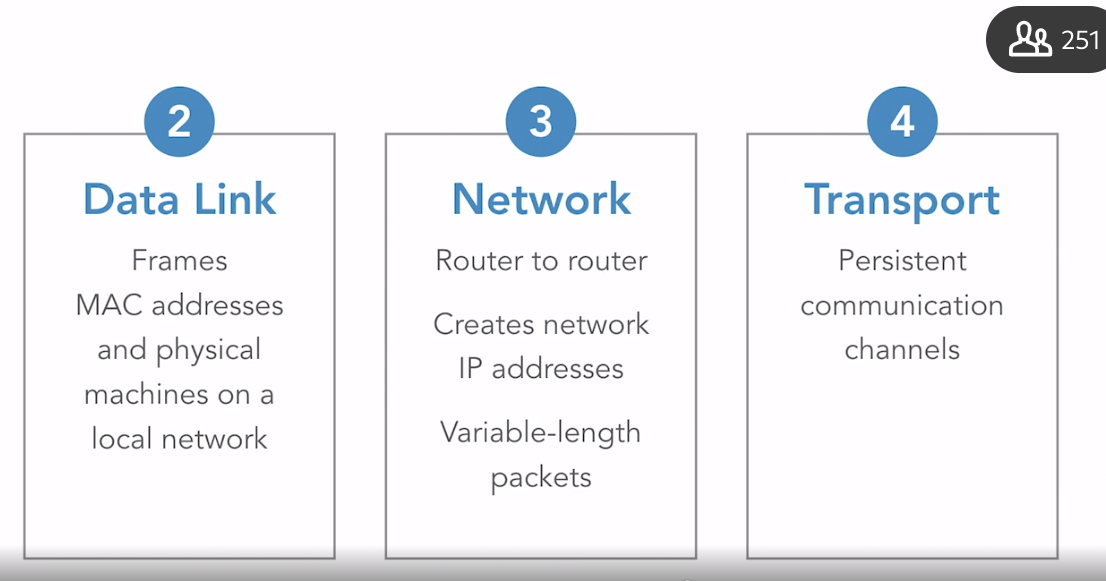
Selecting transcript lines in this section will navigate to timestamp in the video

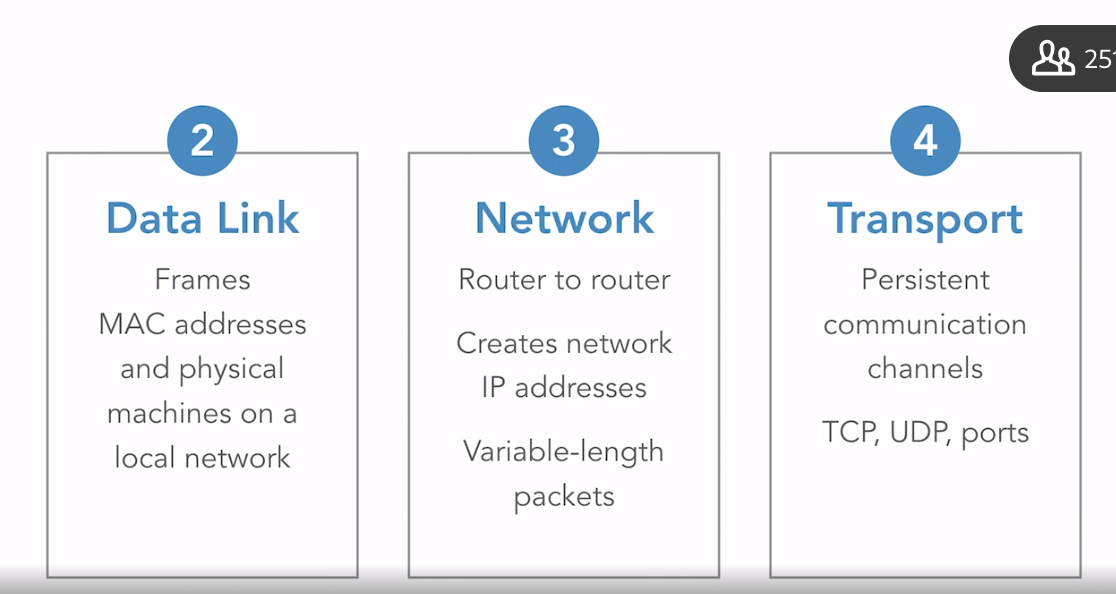
- [Instructor] How the Internet Works. So this is going to be a very brief summary, don't worry. So it's difficult to learn to drive a car if you've never experienced the concept of roads before. Right? Similarly, it's difficult to learn to scrape the web, if you don't have at least a basic understanding of how the internet works. There are lots and lots of things that can go wrong with web scrapers. And these things are influenced by the different layers of the internet that we're going to be talking about. For example, if you made a programming mistake, the web server might say, hey, that request doesn't make any sense to me, or your own client may not know how to direct the request, or maybe your computers ethernet was unplugged. So that's going to look like a very different error, right? Maybe your router can't reach the web server, or maybe the other web server is blocking your IP address. This can be very difficult to distinguish from other types of server errors. Maybe the other programmer made an error, and there's a programming error on the web server. But being able to quickly and efficiently diagnose problems depends almost entirely on having a solid mental model of the internet and how it works. So the mechanics of the internet worked because of a series of layers of addresses and information. This is similar to postal addresses. So when you send a letter, the post office is going to do a broad sorting by zip code, city, and state, right? The post office only looks at the last line, and then, they send it off to another regional post office. Then you have that regional post office look at the street address for delivery. Then, it might get to that street address, and an apartment manager will put it in the right mailbox based on the apartment number. Finally, maybe my husband picks up the mail that day. He sees that it's a letter for me, Ryan Mitchell, and physically hands it to me, like a router sending data to my local printer, right? I opened the letter and read it. There's a greeting inside with my name, a closing with the sender's name, and some content in the middle. Now, the whole point was to get me this content, but there are all these layers of addressing and envelopes, you have stamps involved. And so let's look at how this works on the internet. You have the physical layer. This is literally electrons on a wire. So high and low voltage is representing ones and zeros. This might be the ink on the envelope, in our letter example. Then you have the data link layer. This is the information that gets the data from your computer to the router. This is the layer with data frames and MAC addresses and that sort of thing. So, physical machines on the local network. Then you have the networking layer or the network layer. This connects routers together. This is also the first layer where we get variable length, discrete packets and IP addresses. So if you can talk to your local printer, but not reach the internet, that's the network layer problem. Then, we have the transport layer. This provides connection oriented communication between hosts. So before communication was all one-off discrete packets. Now, you have the concept of a flow of communication error checking and handling and whether or not communication has been disrupted. So this is where you find the TCP and UDP protocols. You also find ports that designate specific applications on your computer, that the data will get handed to. Then, you have layer five, the session layers. So this builds on top of the transport layer to open, close and manage the sessions, the connection sessions between applications. So if you've used AppleTalk or SCP, this is all in the session layer. Then there's the presentation layer. So this has string encoding, encryption, decryption, object serialization, files, what type of files, compression. And then, what we're going to be doing is spending most of our time in the application layer. So this is where you find HTTP, POST and GET requests, APIs. These all primarily concerned themselves with the application layer. So what are the big takeaways from this at the end of the day? How should we think about the internet? Remember, that each request you send has to go through many layers of wrapping and unwrapping and addressing and delivery to get to its destination and back. But at no point is a web browser required for this, right? Good news for web scrapers. A web browser is just a piece of software that sends requests and interprets responses. And we can do all of that ourselves with Python. Requests can be examined and replicated. Anything a browser can do, you can also do with a web scraper. I mean, by definition. So get comfortable with the internet, ditch your web browsers. I mean, unless you're watching this video on a web browser, obviously, and head over to the next section where we actually build our first web scraper.

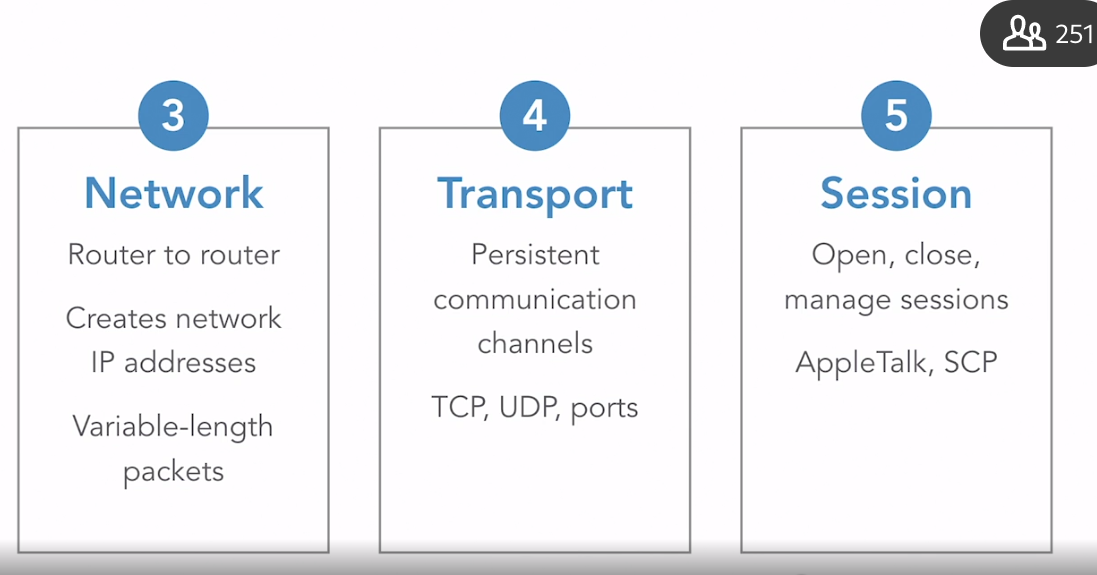


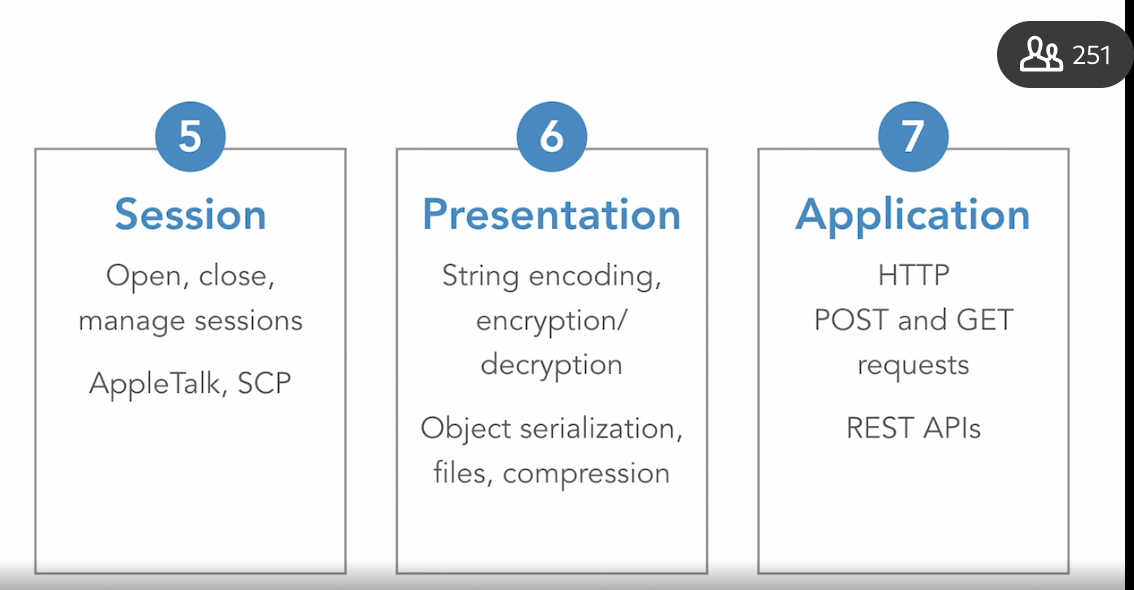


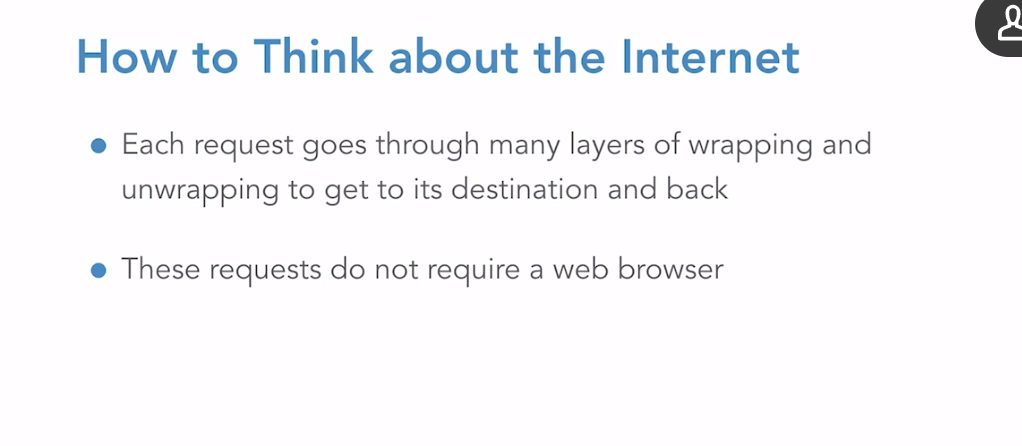


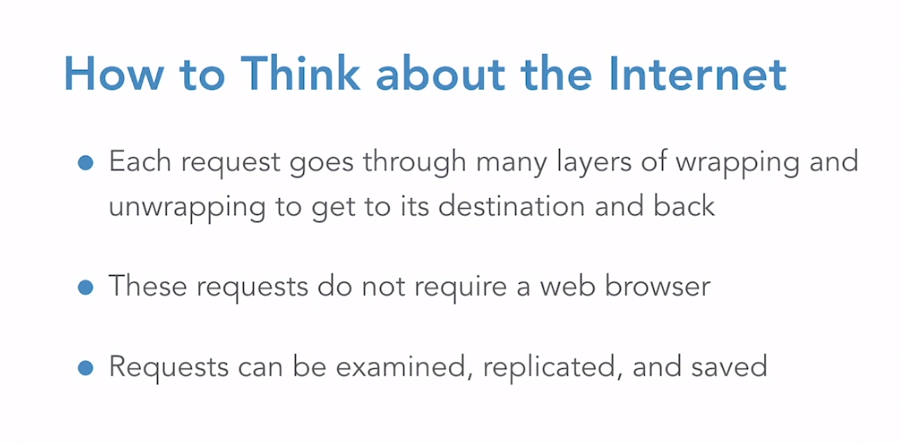












### **Hello world with Scrapy**

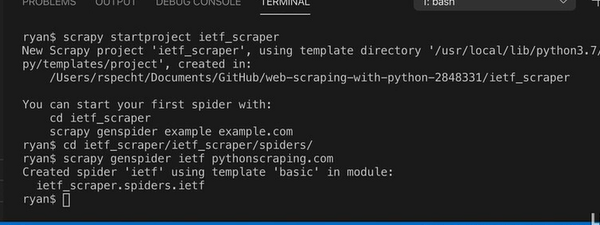
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Enough talk about web scraping theory. Let's build our first actual scraper. So you should already have Pip installed, which is the Python package installer. If you don't, go get it, review the Python package management, and then come back to this video. But Pip makes it really easy to install Scrapy with a one-liner. So pip install Scrapy, with a capital S. If you run into issues, check out the Scrapy installation guide, but after this is installed, you should have access to all the command line tools. So try Scrapy --help, just to make sure everything looks good. All right, so now we want to scrape a web page with Scrapy. So here's the page we're going to scrape, pythonscraping.com/linkedin/ietf.html. Nothing too fancy, I replicated a very famous IETF, that is, the Internet Engineering Task Force, paper, added some tags, cleaned it up a little bit, just so we can sort of use it as target practice for web scraping. And back to the terminal. The first real Scrapy command we're going to want to execute is startproject. So scrappy startproject ietf\_scraper. This will create a template for IETF scraper project. Scrapy creates a lot of different stub files that are somewhat optional. You don't need to fill all of them out in order to start scraping, but we will be discussing these files in the upcoming videos. Just as a quick overview, scrapy.CFG holds configuration information. Items.py defines the objects or the entities that we're scraping. Middleware contains various Scrapy hooks. Pipelines.py defined functions that create and filter items. Settings.py contains project settings. Then finally, the spiders directory. So this is what we're going to focus on right now. It's really sort of the powerhouse of your Scrapy project. In fact, a Scrapy project can be thought of as primarily a collection of spiders, but we need to let Scrapy know about each new spider we want to make with the new command. So let's navigate to our spiders directory. cd ietf spiders. All right. And tell it to make the new spider to go to the domain pythonscraping.com with Scrapy genspider IETF, the name of the scraper, pythonscraping.com. And you can see it creates this sort of template spider file here. The first thing we want to do is modify the start URL. That's going to be linkedinietf.html. There's also this parse function here that is passed a response object via Scrapy, and we want to fill this in with something that will return an object containing the data scraped from our site. Essentially, this turns the response object into a parsed Python dictionary of data. So we have this response subject to work with, and if you look at the page source, you'll see that this title is demarcated by this span class equals title tag. And there are two main ways to capture this title string with Scrapy, using XPath or using CSS. So if we want to use CSS, all we have to do is title response.css span.title, and we use the text pseudo selector and .get as a Scrapy function that gets the first tag retrieved that matches the CSS. If we wanted to get a list of tags, we'd use get all. And then, doing the same thing but with Xpath, span class equals title slash text, and then that same .get. All right. So now both of these methods will work just fine, but during this video series, we'll be using the XPath selectors. So let's stick with the second line using XPath, and I'll just comment that out. What we want to do is return a Python dictionary containing this title data. And there, that should be it. We can run the spider using Scrapy function, runspider ietf.py. And you're going to see a lot of output here, but if we search for avian... There you go, our spider works.

Pip install Scrapy

**Scrapy startproject ietf\_scrapper** >> create template for the projects

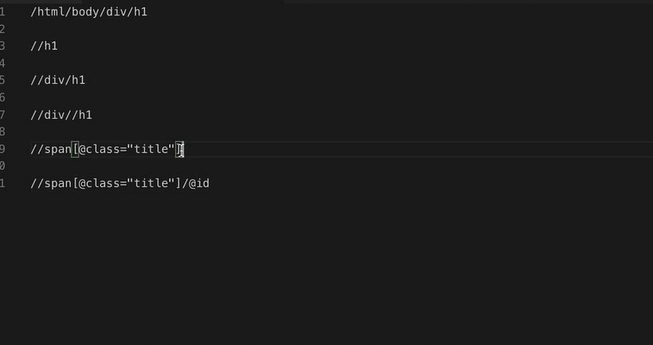
Cd ierf\_scrapper/ierf\_scrapper /spider



### **Challenge: Scraping all data on a page**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Instructor] In the last video, we very quickly touched on how to work with the scrapy response objects in a function and extract the data that we want from them. So remember, there are two main ways to do this, CSS selectors and XPath selectors. Although CSS notation may be tempting to use at first, especially if you have a lot of experience with front-end web development and are very used to using CSS, I would caution against it. CSS selectors are far less powerful than XPath. Although some pseudo classes have been added to work around some of the limitations, you'll likely run into long-term problems if you stick with CSS and completely ignore XPath. So we'll also be using XPath throughout these videos. The XPath syntax can seem a little convoluted at first, but it's actually extremely flexible and powerful, and you can do just about anything with it. It's like the regular expressions for XML or HTML. So let's take a look at a few examples. If you want to navigate to a particular tag in XPath, you have a couple of options. You can do /htmlbodydivh1, and this will navigate to say, a title on a webpage. But it's really annoying. You have to go from the top tag, passing through every intermediate tag to get to your H1. What if you want to go straight to the H1 tag without passing through every tag along the way? Just do //H1. The double slash allows you to select tags anywhere in the page, without starting from the top level of the tree hierarchy of HTML. You can also do something like this, and this will select only the H1 tags that are immediate children of the DIV tag. You can also do DIVH1. In contrast, this would select an H1 tag that falls anywhere under a DIV tag, regardless of whether or not it was an immediate child. And as seen in the example from the last video, elements are selected by attribute using this at symbol, @class=title. Let's bring this over into our examples. This will select any span tags that have the class title or the attribute the span tag is equal to the string title. You can also select the text from the attribute of the tags themselves. So if we wanted to select the ID attribute we do @ID, and this will select the value of the ID attribute where the class attribute is title. Of course you don't usually want the attribute text or the attribute value. What you want is the inner text from the tags. And as seen in the last video, you're going to be using the XPath tag selector a lot. So that's just slash text and that's it. About 99% of the XPath selectors you will ever want to write in web scraping will follow one of these patterns. So as a challenge, go back to the page we scraped the title from in the last video. Pythonscraping.com/linkedin/ietf.html. So create a scrapy project with a spider that scrapes, not just the title as in the last video. but all of the important content from the page in an organized way. So you might want to look at the author's name, the date, the subtitles, the text, et cetera. Look around as well. There may be multiple ways to retrieve the same information. After you have a solution you like check out the next video and we'll compare notes.



### **Solution: Scraping all data on a page**

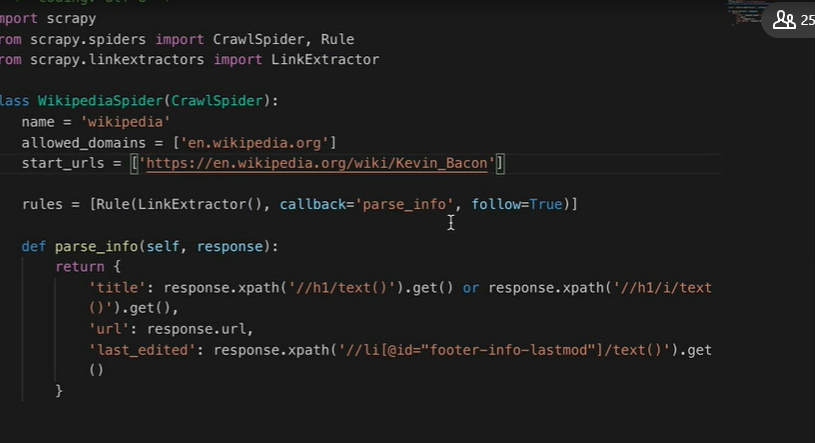
Selecting transcript lines in this section will navigate to timestamp in the video

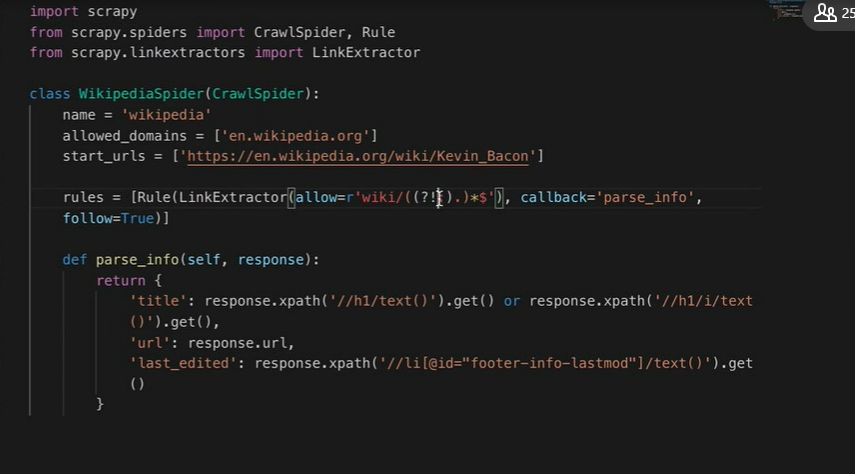
(upbeat music) - [Instructor] I hope you enjoyed scraping your first webpage. A huge skill in web scraping is just being able to stare at a page's HTML and not get lost. So if you manage this, you're halfway there. Some other skills you may have used: digging to find the cleaner sources of data, being a little creative, not being afraid to get dirty and do some data wrangling, looking for perhaps hidden data, if you saw any of that. So, the fields I ended up collecting, although you may have gotten more or fewer of them: the RFC number. This might be useful if you're collecting a database of these things. I had the title, obviously. The date, the description, the author's name, the author's company, the text of the document and the list of titles and headings. If you got the description field, congratulations. If you didn't, did you happen to notice the data in the header of the page? Page headers are often extremely valuable sources of data. They have metadata meant for search indexers like Google, you know, other web crawlers. Remember, to get the text content of the tag, you use text in your XPath. To get the value of an attribute in the tag, you use the at sign with the name of the attribute you want. So when we're getting the description, you're going to use @content, to get the value of the content attribute of the description tag. Sometimes the data and the metadata will be formatted better or at least differently. So you should check it out if you want options. For example, in the visible text of the document, you have a date formatted with the day. In the metadata, you have just the month and the year. Of course, you may want more information rather than less and say you prefer the date with the day field in it. But it is formatted differently so you do have options. This is also true with the author and the title where I have some alternative ways of getting the data shown here in the comments. Also, it looks like we have some internal codes on this page. Here, we have the version, here there's another case where we have the RFC number. If we were scraping a product page, we might find things like the UPC, the Universal Product Code in the metadata here. You might want to save unique identifiers like this. They may come in handy later for say, keeping things organized in a database or referencing it on the sources of website later on. So anyway, this is just my solution. It's not the solution. So I'd encourage you to run it, play around. Maybe you can do better.

### **Crawling a website**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] So in this chapter, we're going to switch gears a bit and focus on creating web crawlers that collect data from Wikipedia. Now, before you get any bright ideas, it is completely useless thing to build web crawlers for Wikipedia in real life. If you want data for Wikipedia, they have fantastic APIs you can use, right? So scraping them is really just a waste of your time and their servers. However, they do have a lot of links and articles and things. You can click on one link to the next, their HTML is really stable. It doesn't change very often. And for this reason, it's a really great site to use when you're learning web crawling or when you're teaching web crawling. So if you do feel inclined and you do want to crawl them as practice, I would highly encourage you to chip in towards the cost of their servers. You know, I give them 20 bucks every now and then, and they are a really great organization. Anyway, let's start a new web crawler. We're going to call this one article scraper rather than naming it Wikipedia scraper, because the type of the thing we want to scrape is an article. So scrapy startproject article\_scraper. All right. So we can name the individual spiders after the sites they scrape for the articles, but the project itself is going to be called article\_scraper. Oops, sorry about that. We definitely want to do start project and not star project. All right. So then we navigate to the article scraper directory and to the spiders directory, and then let's make our Wikipedia spiders. So scrapy genspider wikipedia, en.wikipedia.org is going to be our domain and that's the English-language domain. Right, so let's check it out, see what we have. Spiders, Wikipedia. Great. So the first thing we want to do is extend Scrapy's CrawlSpider class instead of scrapy.spider. And we need to make sure we import this. So from scrapy.spiders, import CrawlSpider. And the other thing we're going to import is Rule, which we'll get into in a second. And we also want to import LinkExtractor. So from scrapy.linkextractors, import LinkExtractor. And rules, LinkExtractor, and CrawlSpiders work really well together, as we're going to see. Needs to be HTTPS, en.wikipedia.org, and let's actually make our start URL /wiki/Kevin\_Bacon. You may have heard of Kevin Bacon the actor, also of course the namesake of the game Six Degrees of Kevin Bacon. So let's fill in a few things that we want to collect from each page in the parse function. Let's get the title response.xpath. That's going to be H1/text.get, or response.Xpath, h1/i/text.get. So for works of literature or movies or things like that, they're actually italicized. So we want to have the italicized version in there as well if we don't find the first version. Then we have the URL. That's just going to be response.url, pretty straightforward. And the last edited, this is the last edited date, at the bottom of Wikipedia pages. This page was last edited on the 27th of October, 2020. There's the timestamp there. We want to grab that too. So the last edited date is going to be a response.xpath li, where the ID is footer-info-lastmod, get the inner text, .get. Okay, great. So now we have something that looks just like the scraper we wrote in the last chapter, but what makes it a crawler and not just a scraper? We need to give it rules for the links to follow. So we can do this by using a Scrapy rule object imported at the top, and that takes as its arguments a Scrapy LinkExtractor object. So the first target is going to be a Scrapy LinkExtractor. The callback is going to be, let's call this the function parse\_info, and then just change this name. I like to use parse\_info for CrawlSpiders and parse for regular spiders. And then we want it to follow links. So follow equals true. This just means that every time it finds an internal URL it follows that, then it keeps following and following and finding new internal URLs and just basically goes ad infinitum. So let's take a look at that first LinkExtractor object. This is the thing that actually parses the HTML page and finds new Wikipedia links to visit. This takes in an allow argument. I actually have it in my clipboard right here. So unfortunately, regular expressions is outside the scope of this class, but what this regular expression means is that we want pages at /wiki and then /some text, sort of like this Kevin Bacon URL. And we do want to exclude any URLs that contain colons in them. So if you look at Wikipedia, pages that contain colons in the URL are things like the special pages, the discussion pages, talk pages, the random article page, right? So what we want are just those URLs that look like /wiki/some text. And now we can run this spider using the usual commands. So scrapy runspider wikipedia.py. All right, great. And you can see what it's doing is it keeps crawling to new Wikipedia pages, and the only way it'll stop, unless Wikipedia runs out of pages, which is probably unlikely, is with a Control + C. And this is multi-threaded, so you may have to press it a couple of times, but there you go. Your first Wikipedia crawler





### **Recording data**

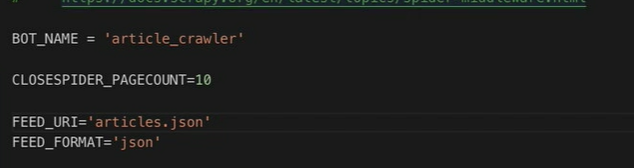
Selecting transcript lines in this section will navigate to timestamp in the video

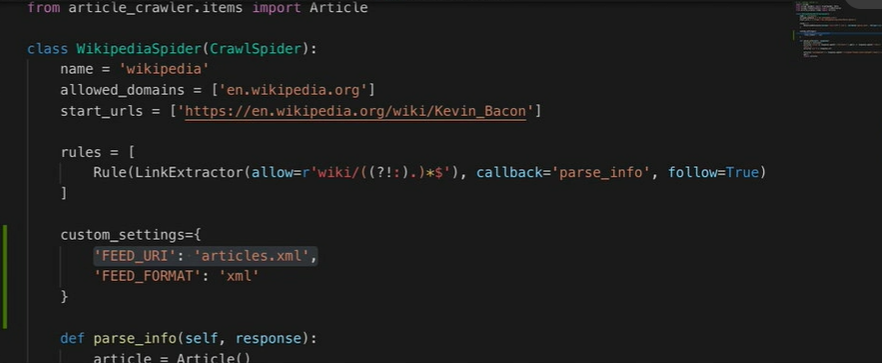
- In the last section, we built this web crawler that prints data to the terminal as it crawls from page to page. This is fun, but typically you don't want to just print data out, you want to collect it, if not in a database, at least some sort of a well-formatted file. Fortunately, this is really easy to do with Scrapy, and in the process of doing it, we're going to be adding a little bit of, you might say, enterprise infrastructure to our crawler that's going to serve as well in building robust, scalable web crawlers in any situation. I've mentioned before, some of these mysterious Scrapy files off to the side that get magically created when we make a new project, and now we're going **to be using one of them, items.py.** And this is where you define your items. Of course, an item is a type of content that you're scraping. In this case, we're scraping articles. So keep in mind these aren't Wikipedias or Wikipedia articles that we're scraping. Wikipedia is the source. It's one of our sources. You might imagine that we scrape articles from all over the place, many different websites, all with their own spiders, but they have this common article object, and we want to define the standardized fields for that. So we're going to change this sort of default Scrapy thing to article, create an article class, and it's going to have three fields. So we're going to have a **title field. It's going to be scrapy.field, a URL field, It's also scrapy.field, and a last updated field.** All right, this may seem a little magical at first, but bear with me. I promise it's all going to come together. Back in our spider, Wikipedia.py, We want to import this article class that we just defined. So from **articlecrawler.items, import article.** Great. We also need to modify our parsing function a bit. Instead of returning a dictionary of data, **it's going to populate an article object. So let's just rearrange this. Article equals article. Article title is going to be equal to that. Article URL is going to be our response.url, and then article last updated is going to be that.** Okay, now we're ready to use this article object to tell Scrapy what data we want written to file. The easiest way to do this is to pass a few extra command line arguments to our run spider command. **Scrapy run spider Wikepedia.py -o articles.csv and then the format T CSV. So you can also use dash T json or dash T XML. This says, write a CSV file and write it to articles.CSV.** There is one little caveat here though. We can't just run this. So in the past, we were using control + C to close our crawlers. Unfortunately, this can cause problems with how Scrapy batches data and writes to files if you just kill it prematurely before it finishes writing, and we can prevent this by giving it a**n explicit close, sorry, close spider page count with the dash S flag and then allowing it to finish by itself, say with a page count of 10.** So let's just say equals 10. And there you go. Let's check it out. Okay, it's crawling, it's crawling. There we go and it stopped, and we should have an articles.CSV file. If we cat it, great, there you go. We have 10 pages. So like I said before, you can also use this for the json and xml format, and it's a really great way to write data to files quickly.

### **Scrapy settings file**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] As is usually the case when it comes to programming, scraping provides the user with multiple ways to do things. SO in the last section, we looked at an example where Scrapy would write CSV data to file. We also limited the number of pages it scraped with this command line option. But the command line options are hard to remember. Sometimes you just want to load some presets. So remember, in the last section we used the command CLOSPIDER\_PAGECOUNT equals 10. The dash s here actually stands for settings. These are one time settings that you can pass in via the command line, but if you want to make these applicable for every command, without having to write them each time, take a look at settings.py. We can write whatever we want in here. There's a bunch of great stuff. I would recommend reading through the comments and getting an idea of things you can control from this file. And it's pretty well-documented by Scrapy. So let's try our close spider page count equals 10 right here in the settings file. And then what happens if we just get rid of this? It should still stop at just 10 pages. All right, great. So it's still picked up on that without us having to explicitly say it. So check out this command here, the dash o articles.csv dash t csv. Again, that means to write the output of this crawler to articles.csv in the format, CSV. There's another way you can write this with dash s. So that would be FEED\_URI at say articles.csv, and then dash s FEED\_FORMAT csv. And that should do the same exact thing. And you might be guessing where this is going. If we know we can always dump the output of our crawlers into a CSV file or a JSON file, we can just put this into settings. So go back to our settings file, FEED\_URI articles.csv, and actually make sure we use the single quotes there, FEED\_FORMAT csv. And let's actually mix this up a little bit, change it to JSON, just so we get something different. And we can get rid of all of this stuff entirely and just use the plain old scrapy runspider Wikipedia.py. It should pick up on these settings. Great, and yeah, cat articles.json, check it out. Another neat feature of these Scrapy settings is the ability to add settings to the spider itself, so this Wikipedia.py spider. These configurations will also override for just that spider, any configurations you have in the settings.py file. So we can test this out by adding to export data to, say an XML file. So custom\_settings, this is picked up by Scrapy. It's going to be a Python dictionary, we want FEED\_URI, articles.xml to the other one we haven't done yet, and FEED\_FORMAT, XML. Okay, so for just this spider, it should pick up on the command, only scrape 10 articles in settings.py. But it'll override the JSON FEED\_URI and FEED\_FORMAT, and we should get an XML file. So there you have it, three ways to extract Scrapy data to files in three different file formats.

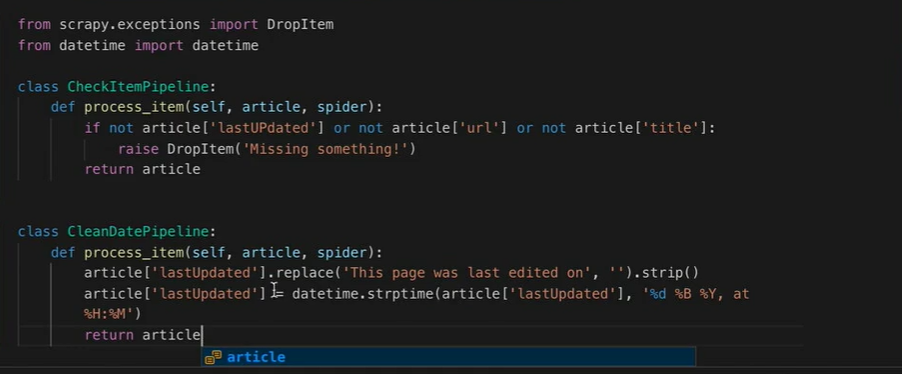


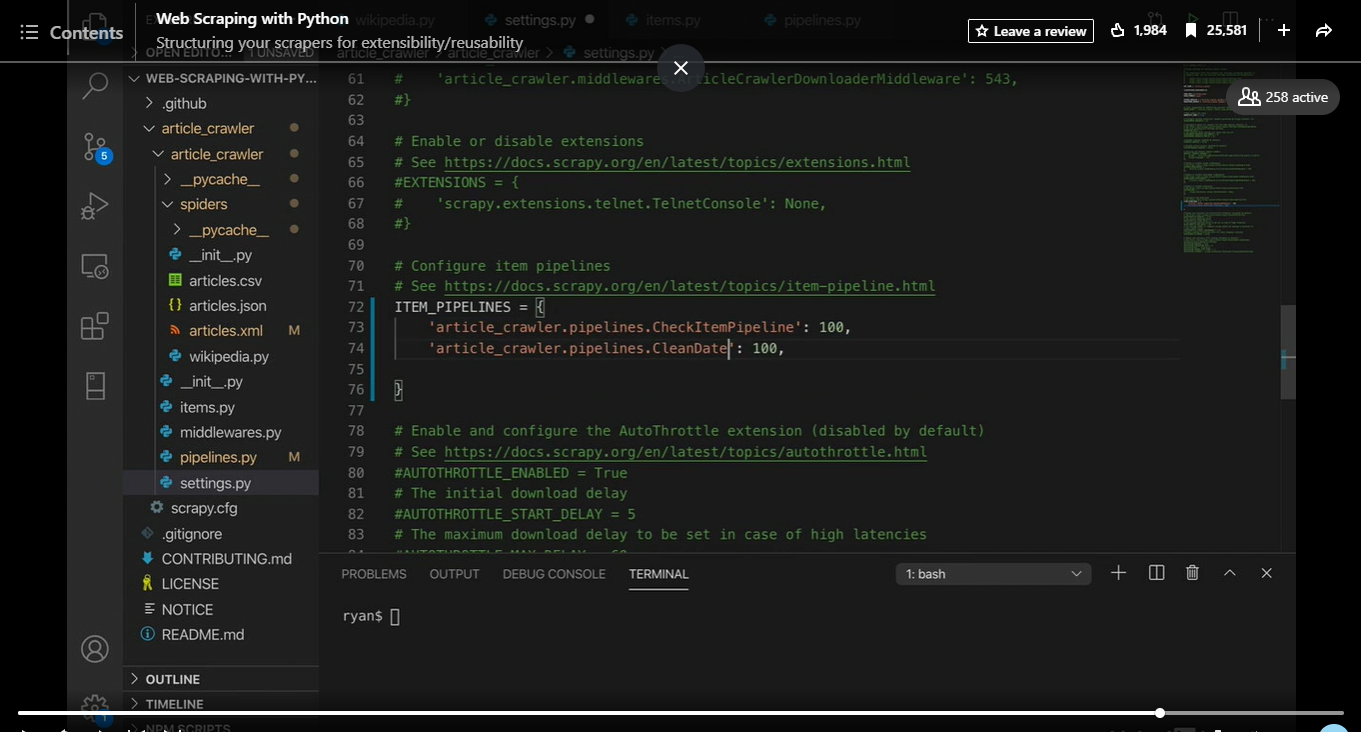


### **Structuring your scrapers for extensibility/reusability**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] So far, we've seen a few of scrapy's moving parts and how they come together to crawl websites. You have the scrapy settings which manages configure options for the entire project or for an individual spider. You have the spiders which encompass all of the websites specific volatility and are custom built to turn a messy website into a consistent type of item. Then you have the scrapy items, the item objects themselves, these items, at least in theory don't require a website-specific knowledge to deal with anymore because, all that was handled by the spider, their data isn't necessarily in the most clean or finalized state. We could write a bunch of cleaning code in the spider, of course. And by the time the data gets to the item object let's say it's absolutely pristine, but then you might end up duplicating a lot of cleaning code in each spider, if you have multiple spiders for multiple websites. You also might want to do other centralized tasks, like data validation, checking for duplicate records, storing data, sorting data, storing the data in different ways, depending on maybe some value of some particular attribute of the item. And these centralized tasks are accomplished in the scrapy Item-Pipeline. So each task in the pipeline, is represented by a class, in pipelines.py. So keep in mind, not all of the pipeline code needs to go here. You can certainly import classes and functions from elsewhere, but it's traditional to keep at least a reference to all of your pipeline code in this file. So let's read a couple of simple classes in pipeline.py. One, to turn that last updated text that we saw in Wikipedia into an actual Python date object, another to discard the item, if any of the fields are missing or invalid. So in pipelines.py, remember we need to make a class for each pipeline function that we want to write. And the first one is going to be, CheckItemPipeline. This function process item we want to keep, the item object passed in is actually our article. So let's just rename that to remind ourselves. And what we want to do, is we want to check each attribute in the article to make sure it exists. So if not, article lastUpdated, or not article URL, or not article title, let's raise a DropItem exception. This DropItem is from scrapy so we can import that, and let's return to the article. The other thing we want to import, is the datetime. So from date time, import date time and this is just the Python date time object. So let's make another pipeline function called CleanDatePipeline. That's going to have same process\_item, self articles, Spider. All right. This is going to take the article and take the lastUpdated. And then let's just replace, that this page was last edited on text. Just as a reminder, this is what the text looks like from the Wikipedia page. So this page was last edited on, blah, blah, blah. Let's just go and replace that. This page was last, edited on, replace that with an empty string, and then strip any white space around it. Article last updated, it's going to be equal to date time. This is the Python string to time function article last updated, and then we give it the date format. So that's going to be, d capital B, Y at H, M and then return the article. Okay. So now that we have these item pipelines written, how do we tell scrapy that they should be called? For this we need to go back to the settings.py file. So in settings.py, you're going to see some commented configurations called ITEM\_PIPELINES. Let's just uncomment this, and add our own pipelines in. So the first one is going to be CheckItemPipeline. Let's just give this a value of 100. And then let's add the second one CheckItemPipeline, and give that a value of 200. The number value here on the right determines the order of the pipelines are called in. So lower numbers are performed first, higher numbers are performed last. Although the numbers can be in any range that you want, they're traditionally zero through a thousand. So all right, let's check it out. Scrapy runspider wikipedia.py. Note that, although we won't be getting into databases in the series. If you do want to write your items to a database, the item pipelines are the preferred way to do that. If you just want to write your data to say JSON file or something, use the built-in feed exports. We looked at in the last section.

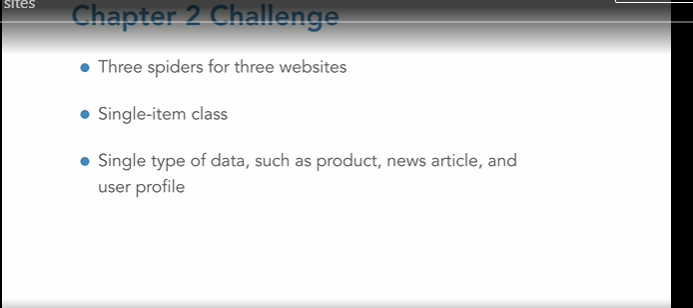




### **Challenge: Scraping news sites**

Selecting transcript lines in this section will navigate to timestamp in the video

(lively music) - [Instructor] Although I've talked a lot in previous videos about projects with multiple spiders, we've really only seen projects with a single spider so far, either the Wikipedia spider or the IETF scraper. So in this challenge, we're going to put theory to the test and build a full Scrapy project with at least stubs of all the components working together. So you're going to want an item class, some pipeline process or processes, some settings that maybe export JSON or some other file format using the settings file. And the goal is to scrape some standard type of data from three different websites. So I'm going to scrape news articles from the Associated Press, CNN and Yahoo News. If you don't like news articles, feel free to get a little creative with this. Products, profiles. A word of warning though, don't go too crazy. You want something common, present on a lot of different websites so you can pick a few sites that have all of the information you need readily accessible. So you can be a little bit creative with this, pick something you like or you can be cool like me and just use news articles. CNN, Yahoo and AP News are pretty good sites, just saying. But aim for a three scrapers, three sites, a single-item class and an item pipeline.



### **Solution: Scraping news sites**

Selecting transcript lines in this section will navigate to timestamp in the video

(upbeat music) - [Instructor] Like I said before, I decided to scrape news articles from Associated Press, CNN, and Yahoo News. To be honest, I got a little lucky with these sites. I did scope out a few different sources and picked ones that seemed moderately easy to scrape, but sometimes you really don't know what you're going to get until you actually build it. So everything went pretty smoothly, all things considered. I created a NewsArticle item, and that contains the title, description, date, author, full text, all that stuff. CNN was probably the most straightforward site to scrape. The only tricky thing I had to do was something that we already covered in chapter one, and that's use the metadata to get information. I was able to get really clean versions of the description and date published from the metadata in the header. The author's name was also there, but still required a little cleaning, which I did in the item pipeline. The text required a little bit of tricky finagling with xpath here. It's kind of a long selector, but I was able to extract it as a list of paragraphs using getall. Associated Press was a little bit trickier. I struggled to find a clean source of the author and date and such, but then I saw that they had all this information as JSON text in the script tag on the page in the header. So this kind of thing is actually pretty common. It's used as a source for JavaScript code on the page to access the data cleanly. So always keep an eye out for JSON blobs. They can be really handy when you're web scraping. So you can extract this blob of data as text, and then use Python's json.loads to turn it into a Python object. From there, you have all of the information in a beautiful, clean Python dictionary. Getting the body text data was a little bit harder here, but easier than CNN. So you know, a little navigation with xpath, a little elbow grease, and just get 'em all. Yahoo was pretty similar to Associated Press. Again, I found most of the data in a JSON blob on the page. All that was left was the article text, navigate to it with xpath, get all the paragraphs. Now I want to take a look real quick at the rules that were used and the LinkExtractor that I used. You have to do a little bit of detective work about what constitutes a news article or a product page or whatever it is when you're scraping versus what's a non-news article or maybe a main landing page. The regex took a little bit of trial and error, and I discovered some of the internal rules that these websites used. For instance, Associated Press starts with /article and always ends with a 32 character UUID. CNN always has the date, then they have two directories. Those directories can contain letters and dashes, didn't see any numbers, and always ends in index.html. So you really have to sort of use your powers of observation and see what's going on with the URLs when you're writing these. All right, so on to the item pipeline, there wasn't too much I needed to do here. I mentioned that the author names from CNN needed to be cleaned up, so I did that. I transformed the date strings into actual Python date objects, cleaned up the article body text a little bit. From there, I can put this into a database, write it to a file, do some analysis, or maybe just catch up on the news.

### **Submitting a form**

Selecting transcript lines in this section will navigate to timestamp in the video

- So our web crawlers have been making a lot of HTTP requests while they traverse the internet. These are the same HTTP requests that happen when you visit a website in your browser. And you can see all these requests pop up in your browser's developer tools as they're being made by the browser. And this can be a really handy starting point for scraping forms because all a website form is is a sort of creative application of HTTP requests. So we can see this form over here and we can see this request in action. If you don't understand what this form is asking, or if you're a little confused, by the way, I recommend that you watch more Monty Python. But my name is Ryan, my quest is to seek the grail of course and my favorite color is blue, nope, green. My favorite color is green. All right, let's do that. Let's see this form action.php down here and you can see all those parameters I put in there, and the page uses those parameters to display some content there. And we can take advantage of this with Scrapy. So I happen to have a freshly created spider here in a project called Form. And we're going to use the spider to traverse a list of names and quests and scrape data from the page that shows up, that submitted form page. So we're going to do this by taking advantage of start\_urls. Now usually we just put one URL in there and the caller takes care of the rest. But this time, we're going to be writing a function called generate\_start\_urls that provides all of the URLs we want our scraper to traverse, given some set of form values. So def generate\_start\_urls. Let's make some names. Alice, Bob, and Charles are always classics. Let's make some quests to seek the grail to learn python and to scrape the web. I mean, why else would you want to learn python? Okay, and let's return these URLs that we made. So pythonscraping.com,linkedin, and that's going to be formAction.php where the form was submitted to. And then the get parameters. Name and quest and color, let's just hard code that, we'll make it blue. Name quest for name in names for quest in quest using the python double list comprehension there. All right, so start URLs, that's going to be generate\_start\_urls. Then when we parse it, let's just return text so this is going to be the text content of the page, which is at the XPath. I looked it up so that you don't have to. Class is going to be wrapper. Text.get. Okay, great. So save that, run it with scrapy runspider get\_form.py. Okay, perfect. And you should see, great, we've scraped all those different permutations of text from the page. But what about other types of forms? If you check out form2.php, you'll see it's very similar, but instead of making a get request, it actually makes a post request so you don't see any of those get parameters there. No get parameters. But you do see this form data here. And we can actually use Scrapy to send form data as well. So I'm going to make a copy of this, and we're going to call it post\_form. So rename post form.py. And then instead of having this generate\_start\_urls out top there, we're going to actually take advantage of a built-in scrapy function called start\_requests. Perfect. And so this is actually going to be an attribute of the class. start\_requests, and Scrapy knows about this. It knows that, just like start URLs, if you call a function start request, Scrapy looks there and it looks for a list of FormRequest objects to be returned. So a list of FormRequest objects. We can import FormRequest. Sorry, from Scrapy.http import form request. Okay, great. And this FormRequest object takes a URL so that's going to be formAction2.php. The other argument it's going to take is form data. And that's just going to be a Python dictionary, so name is name, quest is quest, color, again, we'll just hard code that to be blue. And then a callback, that's going to be our self.parse function right there, and then get rid of that. For name in names, for quest in quests. That looks good. Oh, let's call this post\_form of course. All right, and let's run this. Great, and then you see all the different permutations of that text from postform.py so get request, post request. If you want to submit a form and scrape the content on the other side, there you go, you can do it with Scrapy.

### **Finding and using hidden APIs**

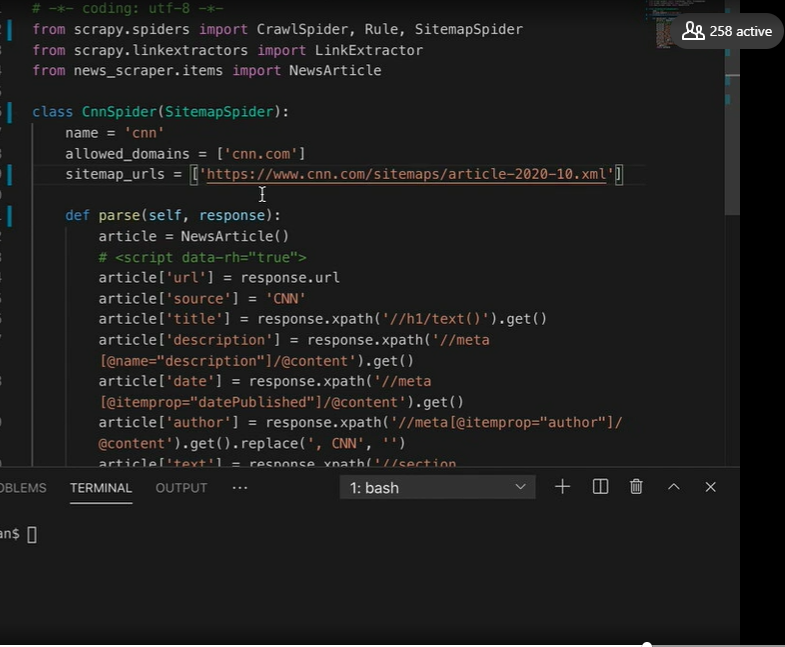
Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] One of the most important skills you can develop as a web scraper is your powers of investigation. So you can see the data right there, right on your screen. But also you need to know how it got there, right? Where it is, other sources of that data. You need to kind of understand how the website's working. So let's take an example that's kind of near and dear to my heart, Dunkin' Donuts. Now, something you never need to do where I live in the Boston area is search for a Dunkin' Donuts location. We typically just walk out of our front door and there's one right there, you know. But for the sake of example, let's check out the Dunkin' Donuts location finder. And I promise Dunkin' Donuts did not pay me for this, but you know, if they want to, I do accept free coffee. If I enter my zip code, 02155, I see 10 Dunkin' Donuts right there. And here's the one that's closest to my house for any fans seeking autographs. Let's see if this address, 850 Broadway, is located in the page's source. No, no, it is not. So what's going on here? Why do I see this on the page but I don't see it in the source. Well, what's happening is that all of this content is being loaded dynamically. All the HTML of the page contains is this search box form, and that form knows how to go out and make a request for some data at a totally different URL, and that data gets returned to some JavaScript code that sort of sneakily updates the DOM, or the contents of the page while I'm looking at it. And when I go to view the source, I don't see any of that content that got magically inserted, right? I don't see the current state of the page, all I see is this search box application that has the ability to go out and call JavaScript. So what to do? Well, we're looking in the wrong place. We don't want any of this HTML over here, even though that might look like where the content is. What we want is the URL that's being called by the page, which contains the actual location data. So let's try the search again, but while we're watching our network transactions very carefully. So, 02110, it's downtown Boston, and let's see if we see anything interesting. Oh, check that out. That looks like it might have some location data in it. Okay, so we can see, the search results. All right, great. So I'm not totally sure what all this stuff is. Suffolk County, Kosher, Monday hours, weekday hours. Yeah, it looks like a lot of stuff that's not even displayed on the page there, which is pretty cool. And we can actually take a look at this URL and we can copy this and we can see if we can make the same request ourselves. All right, great. Let's see what happens if we just start getting rid of stuff. So this callback thing there. Nice, that gives us better formatted of JSON, we get rid of that thing used by some JavaScript code up top. The key, do we need that? Yes, we do need the key, okay. So let's put that back in. This looks like our zip code, 0215, oops, that's not a real zip code. Let's put that back, 02155. That'll give me my zip code, max matches. What if we just changed that to say 300? Oh, and that radius, maybe we change that to 250. Whoa, that's a lot of data we're getting back. Okay, a lot more than 10 results. So that's pretty cool, just by doing a little bit of detective work and playing around, we go from this frustrating situation where we don't see the data on the page, it's just not there, what are we going to do? To, I mean, basically having our own personal Dunkin' Donuts database. So keep an eye out for things like this. Whenever it seems like data is missing from the page and it's not where you'd expect it to be, look through the network transactions. I promise that data is coming from somewhere. It may just take a little bit of elbow grease or like a really strong cup of coffee.

### **Sitemaps and robots.txt**

Selecting transcript lines in this section will navigate to timestamp in the video

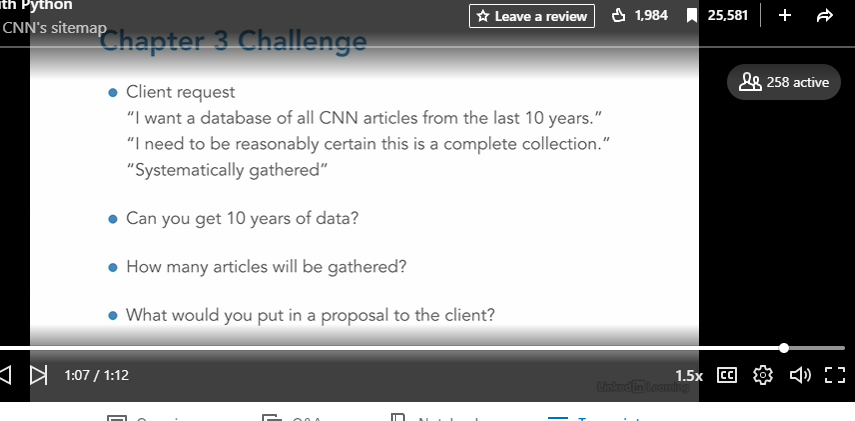
- You may have heard of robots.txt before. It's essentially a file at the root of most domains like Wikipedia/robots.txt. this gives instructions to any passing bots about what they should and shouldn't scrape. And the syntax of this file is determined by something called the robots exclusion standard or the robots exclusion protocol. The cool thing is, you don't need to really worry about any of that in order for your scrapers to follow robots.txt. Check out settings.py in the news article scraper I used as a solution for the challenge in chapter two. You have the obey robots.txt rules, and it's set to true here. So Scrapy will automatically go to the robots.txt for any domains you give it, check out what it can and can't scrape and then follow the rules there. And listen, obviously, if a site's robots.txt is a big problem for your scraper, you can always set robots.txt obey to, you know, false, I guess, but please don't do this for Wikipedia. As I've talked about before, they have a fantastic API out there, and there's no reason to disobey their robots.txt. It's really very generous, all things considered. Other sites, I guess go nuts, but Wikipedia is nice so let's be nice to it. Moving on, I don't want you to think that robots.txts is just a buzzkill for scrapers. Au contraire, here is a wealth of useful information that can be found in these files. In fact, when you have a large crawling project, investigating robots.txts can be a good first step. So check out cnn.com's robots.txt file for example. Like many robots.txt files, it lists some sitemaps up here. See those? And what these sitemaps are, are XML files designed to be read by search engine scrapers. So that search engines don't miss any pages or they can view the latest updates to pages. And if we follow this a bit, you know, we can see the site maps index, and we can actually choose any month we want, and check that out. It looks like the formatter is having a hard time with this one, but we can see just like a list of all the recent news articles. A lot of recent news articles, a lot of news lately that CNN's published. And this looks like October, 2020. So take a look again at this CNN scraper I made for the chapter two challenge. It goes to the start URL I give it, looks for internal links to articles looks for internal links to more articles from those and kind of goes on this recursive random walk throughout the website. The site map by contrast is an orderly listing so we can stop our scraper halfway through, know exactly where it left off, make sure we're not missing anything. Finding a site map, is an amazing thing for web scrapers and Scrapy can actually deal with them out of the box using the SitemapSpider. So the SitemapSpider, we want to import that from scrapy.spiders. Perfect. And it's pretty easy to change this spider here into a SitemapSpider. Obviously we get rid of start URLs. We don't need that anymore. We don't even need rules. It's literally just, going to take the site map and use that. So we do need to provide it site map URLs. we can give it a list, but here we'll just do them one. So October, 2020. We can use the same parsing function even. All right. I think that's it. So, let's run it. (typing) Perfect. Now, Wow! Now we have a spider that's crawling linearly through an organized list of articles provided by CNN itself. So, thank you robot overlords.



### **Challenge: Using CNN's sitemap**

Selecting transcript lines in this section will navigate to timestamp in the video

(bright music) - What if a client comes to you with a web scraping project and says, "I want a database of all CNN articles from the last 10 years, and I need to be reasonably certain that this is a complete collection, gathered in an ordered and systematic way." So what do you do? Remember a scrapy crawler, like the kind we worked with in Chapter 2 might reach all the pages eventually, but it might not, due to the random path it takes throughout the website. It's hard to crawl in an orderly way. So this type of solution isn't going to meet your client's needs. But is there a way where you can take five minutes, do a little exploration and go back to the client and say, "10 years of articles? I can give you 20." Or maybe it's, "Sorry, I can only do the last eight years." Is there a way you can estimate how many articles are available quickly, which can translate into how long it'll take to scrape? So take a few minutes, explore cnn.com. Think about how you'd build the scraper. Maybe put together some highlights for the client. Think about how you'd write an email back to them, for example, and then head over to the next section where we talk about this problem.



### **Logging in**

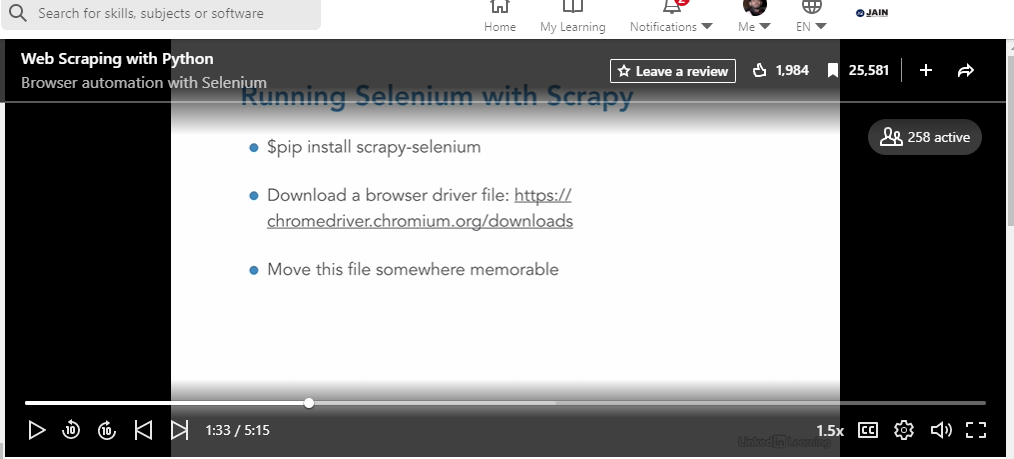
Selecting transcript lines in this section will navigate to timestamp in the video

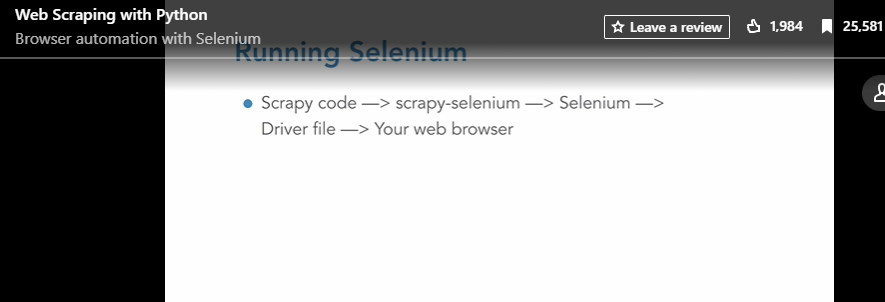
- [Instructor] So, so far all the data we've scraped has been public. You don't need a user account or any particular credentials. You don't even need to pretend that you're not a bot. The websites are really just happy to accept anyone, right? But in this chapter, we're going to start looking at becoming something we're not. So whether that's a credentialed human user of a website, an actual web browser executing code, or someone sitting in front of a computer screen pushing buttons. And in this particular section, we're looking at logins. So as usual, our web scraping adventures are going to begin, not with scrapy code but with just looking at the website. So that's really important. We're going to look at the website, we're going to see what's going on with it. And that website is going to be a login page, pythonscraping.com/linkedin/cookies/login.html And if you look at this login, there are some little hints about what you can use as credentials. So my username is going to be ryan and my password they helpfully suggest, let's try password. Okay, and let's open up the network tab to see what calls are being made when we hit the login button. You can see we've made a post request to welcome.php here and you can see that we've sent some form data along with it, there's our username and our password. And if you remember, we did the section on submitting forms and you might be tempted to replicate this with your Scrapy scraper, but you're not actually going to do anything just by replicating this. So the act of making the request to log in, making this HTTP request, isn't actually the important part. You can see that we're still logged in when we click from page to page on the site and the important part is actually the headers that get set the cookie that gets set. And if you look at the cookie that we're sending whenever we make a request to this website now, the username is Ryan and you have logged in equals one. Now in reality you're probably, hopefully not going to see a cookie like logged in equals one on a website and in reality, it's going to be some sort of long authentication token with a timeout and it gets renewed by the server and all that. But the principle is the same. You want to make sure that whatever cookies you have while authenticated on the website are being copied and sent by your Python web scraper while it's making requests. So let's look at how to do that. I have a new project called Profiles and it has a spider, pythonscraping.py and I've written a few things in here, really just start URL is going to be that profile.php page that says, Hey, welcome to the website. And then we're just parsing, the only parse function is just get the body text, so pretty straightforward. So we're missing something in the middle. How do we set these cookies? There are a few ways to do this, but the way I like to do it is to modify the scrapy request out to the website before that request object actually gets sent. And a good way to do this is to override the make\_requests\_from\_url function. So this is an internal sort of scrapy method that's used by scrapy.spider. And what we want to do is call the parent, so PythonscrapingSpider, self, make\_requests\_from\_url, URL. And so this is going to get our request here. This is sort of like intercepting it in the middle of Scrapy doing its thing. And all we want to do is add our cookie there. So cookies, username equals Ryan. Let's make this a little more exciting even, Ryan and then request.cookies, logged in, this is what lets the server know we're logged in equals one. Yes, I am logged in. Return requests, we need to make sure we return that. So we're getting this request from the parent, make\_requests\_from\_url function, we're sort of intercepting it, adding our cookies and then returning it. We can run this in the usual way, scrapy runspider pythonscraping.py and great. Hey, Hey, Ryan, looks like you're logged into the site.

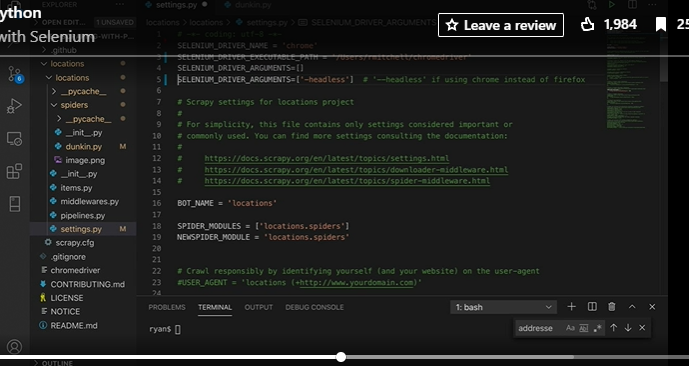
### **Browser automation with Selenium**

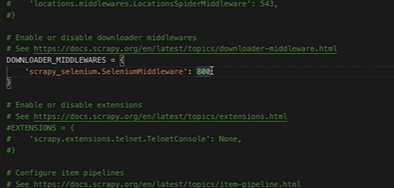
Selecting transcript lines in this section will navigate to timestamp in the video

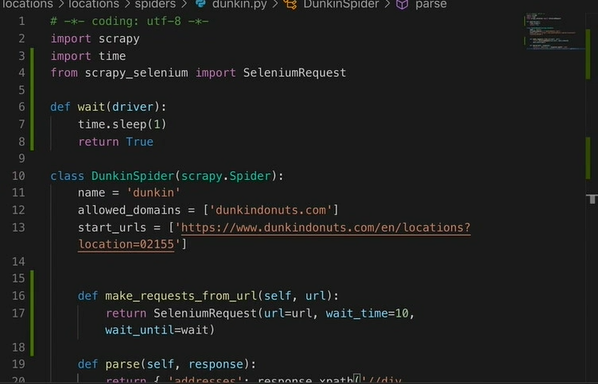
- [Instructor] Remember in chapter three, when we looked for Dunkin Donuts locations, but they were tricky to find because they were loaded dynamically on the page? Well, there are multiple approaches to this kind of a problem as it turns out. Personally, my favorite is just to look for the hidden API requests, just like we did in chapter three, find the original source of the data and then use that API request in your scraper. But now we're going to do something which is not quite as slick or elegant, but way more impressive looking. It's actually a really useful tool to use in a variety of situations. So that's right, we're going to bring out the big guns, the MOAB, that is the Mother Of All Bots, of course. And we're talking browser automation. To do this, we're going to need to grab a couple of things. So first, install a library called scrapy-selenium. This is a Python package that installs a browser automation tool called selenium. And this allows you to use it smoothly with scrapy. Next, you're going to need to grab a file called a browser driver. This driver file acts as an interface between the code in selenium and your actual web browser software, allowing selenium to drive the browser. Obviously, this Chrome driver file is only going to work with Google Chrome. If you don't have Google Chrome, I do recommend you try it. There's also a WebDriver for Firefox that you can get. The selenium approximately the same for both, but in my experience, Chrome driver is better maintained and less buggy than Firefox, especially in recent years. So once this file has been downloaded and unzipped, make sure to move it somewhere memorable because we're going to need to let our Scrapy project know where it is in the settings file. So the entire chain looks like this. Your scrapy code interacts with the scrapy-selenium library, which interacts with selenium, which knows where the driver file is. The driver file drives your web browser. And if this looks complicated, don't worry about it too much. It really just boils down to a few lines of configuration and a couple of lines of Scrapy. So let's go over it. Alright, I've conveniently made a new scrapy project called locations with the Dunkin' Donuts scraper in it. So, here are the three lines we'll need to add to settings.py, up at the top here. So there's the selenium driver name which is going to be Chrome. You want to let it know where it is. Absolute paths tend to work a little bit better. And then an empty array for the selenium driver arguments. If you want to use headless browser mode, which is where the browser doesn't actually physically pop up on your screen, but sort of runs in the background, just uncomment that line. You'll also want to add some middlewares. All right. So there's the scrapy-selenium middleware that we need to import there. Okay, so now for the actual spider. We're going to import the scrapy-selenium package from scrapy\_selenium, import SeleniumRequest. All right. Next, we're going to make use of that make\_requests\_from\_url function that we discussed in the last section. And what we're going to do is we're going to have it return a SeleniumRequest object. Alright, so we need to make sure this has self and URL in the arguments. And URL equals URL. We also need to provide a wait time, wait\_time. So, this is a maximum wait time in seconds that selenium will wait for the page to load before timing out. And this goes hand in hand with wait\_until. Now this whole selenium wait feature is really powerful. And honestly, we could do a whole chapter just on this, but for now, let's just wait a second. So let's just define a quick and dirty function up here. It's going to take in a selenium driver object, and it's literally just going to do a time.sleep one second and then return true. And this is kind of a nice little hack for situations like this when you don't want to do a lot of examination of the page, or maybe you don't know quite what you're waiting for yet, and you just want to wait for some set period of time. All right. Now the parse function is written just as if all those dynamic elements are on the page. It's none the wiser, it's going to get its selenium request object, and it doesn't really have to worry about anything. And we can actually just run this now with scrapy runspider dunkin.py. Before we do that, I'm going to minimize the page. And you can see the browser literally popped up. And if we look for the addresses, you can see them right there. The page loaded, the dynamic content loaded, and our scraper was able to actually pick up those addresses on the page.











### **Interacting with a page**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Okay, this section's a fun one. We're going to sort of tie everything all together in chapter four and show you a delightfully redundant way to log into websites without messing around with cookies and all of that. You just type your information in and press the login button, of course. Remember on this website the username can be anything and then the password has to be password. Great. So we're going to start this off with a brand new profiles project. And this project has a spider called PythonScraper.py. The start URL is going to be the same as the one we used in the first section of chapter four. So that's that login page that we just saw. We also want to make that same make\_requests\_from\_url that we used in the second section. And that's going to return a selenium request object. Also note that we want to bring over the same settings.py file we set up in the last section. So now let's go to the really fun part. Now let's write the parsing function. But this time it's not really a parsing function. There's actually a lot of sort of action stuff we have to do in here. What we're going to need to do is find this username field element, enter some text into it, then find the password field element, enter some text into it. Then find the button element and then click it. And then we'll have to find this profile link, click on that. And then we'll grab the text from the profile page. All right, so let's break it down. The first thing we need to do that is a Python object. This is a Python selenium object called a driver. This is not the same as the Chrome driver file that you downloaded. It's sort of the code representation of it in the python-selenium package. And this driver is going to be nested in our SeleniumRequest object. So that's going to be the response object that gets passed into this parse function. So we're going to be able to find that at response.request.meta driver. And yes, I did have to consult the documentation for this one. It's a little bit arbitrary but that's where that selenium driver object is. And this driver has some cool stuff I can do like driver.find\_element\_by\_xpath. Okay, so this is a selenium function, find\_element\_by\_xpath. And what it does is get an element from the DOM that we can use to interact with this element on the page. And if you look at the page the first element we want is that username field. So that's input name equals username. Okay, great. All right, so we're going to extract the username object into the variable username, and then we're going to want to use another selenium function called send\_keys. So username.send keys and we send the username that we want to log in with. Then we're going to do the same thing with the password field. So we can really just copy paste. Instead of the name being username it's going to be password, of course. And let's send the keys password. Then we want to find the submit button. So that submit driver.find\_element\_by\_xpath. And that would be yeah, input type equals submit. Great. And then we want to do submit.click and oops. Make sure those variables are unique. I think that would technically work but let's make this code a little bit clearer there. All right. And then we want to wait a second or two. Let's have a little fun with it. And then we want to find the profile link on the page. So we should be logged in now. We should have those cookies set in the browser. So let's find the profile link and that's at driver.find\_element\_by\_xpath, of course. That's going to be that a href equals profile.php. Great. And then profile link.click. Wait a couple of seconds for the next page to load, and now we should be on the profile page itself. So from there now we're back in Scrapy land. We just return text response.xpath. Note that we're using that Scrapy response again. And this is all back in Scrapy land text.get. Okay, so this is all selenium using the driver. This is Scrapy using that response object. And we can just run this in the usual way. Scrapy runs spider Python scraper.py. I'm going to tile the browser again. And the page opens. We enter the text, log in to the website, click the profile, and then great. Our scrapers should have grabbed that text. So that's just a quick sample of some of the cool interactions you can do with selenium and Scrapy.

### **Next steps**

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] So the thing that I recommend to everyone when pursuing any programming or computer science field, pick a project that interests you and learn via doing that. Granted, if you're watching this series because you already have a specific project for work, well great. Maybe you declare your interest to be making money and pursue that. But seriously, having a specific goal you're working towards is going to make the path to learning so much easier. More concretely, I think there are a few major directions that most people are going to want to go in from here. If you're interested in Selenium and automated testing with Selenium, check out "Python Automation and Testing" by Bhoomika Agarwal. This covers Selenium from a web application testing point of view, but the principles can also be applied to web scraping. So the only difference is you're not testing an app you own and web scraping, you're testing someone else's app, right? You might also be interested in data science, so, you know, doing stuff with your scraped data. Now, as it turns out, data science is not quite as glamorous as Wired magazine articles would have you believe. A lot of people struggle with just keeping their data organized and forming proper relational data structures, so, might I point you to MySQL. I love planning and setting up a solid database before any large scraping project, and Bill Weinman has a fantastic introductory course called "MySQL Essential Training". MySQL works really well with Python, it's well supported, free, really a go-to part of my web scraping stack. If you want to work more with in-memory data structure manipulation or data cleaning, sort of actively working with the data and get to know a few popular Python libraries while you're doing that, check out "Python Data Analysis" with Michele Vallisneri. It's a really great course for building your sort of intuitive sense of data and data structures, and that's going to come in really handy for building web scrapers. And of course, there's a ton of web scraping specific stuff that just couldn't be covered in a short course like this. For all the rest, I did write a book, so please check that out if you're at all interested. It's not focused on scraping in particular, but it does go into more detail about all the topics we've covered here, as well as branching out into some really cool topics that we just didn't have time to get to. And you are done with this course, my sincere congratulations. Please reach out to me on Twitter, and let me know what you thought. Tell me about any cool projects you're working on or just interesting websites your bots have found on their travels. Thanks.