ProgrammingParadigms-Lecture26

Instructor (Jerry Cain):Here we go. Hi, everyone. Welcome. I actually have, in spite of what it looks like, three handouts for you today, but we had some photocopier drama, and so I handed out the practice midterm already, and we have two more handouts. The solution to the practice set of problems, and also tomorrow's discussion section handout. I know a lot of you have 107 is falling off the radar because you know the assignment isn't too bad, and the Python's not on the final exam.

But this memorization which was discussed in the Assignment 8 handout is central to you getting the problem done nicely, and if you have any trouble with that then you're gonna wanna go to the discussion section tomorrow because we do an even more difficult example than the one you have to solve in that section handout. We also do a little XML processing and data processing using some of the built-in functionality from Python that I'm gonna go over today. I think it's a cool little section handout.

It's pretty lightweight, but nonetheless, go to the discussion section tomorrow if you have any more curiosity about Python. I'm really regretting not having more time for Python because I think it's such a fun little language. But we've already learned all the paradigms that I wanna go over, so we're just looking at Python as a representative of all three of them. As far as the final exam is concerned, remember it's a week from today. I can't say it enough, but there's no Python on the final, but everything up to the last lecture of Scheme through the entire quarter is fair game.

I certainly will emphasize material not covered in the midterm, but given that the midterm is worth what percentage that it is, and the final's worth a little bit more, I really try to make it so that all of the C material that was emphasized in the first half of the class really contributes about half as much, or I'm sorry, equal weight as all the post pure C stuff. So expect to see one very simple C question, a code gen question that focuses on C++ and references, the stuff I did not emphasize on the midterm, and then scheme, concurrency, etc. Okay. No etc., scheme and concurrency. Okay. Yep?

Student: That was [inaudible]

Instructor (Jerry Cain):Sorry, code generation. So yes. Certainly, yeah. I'm sorry, there's a morning offering, 8:30 it's in Dinkelspiel Auditorium. You're gonna have so much room there because only about 25 percent of you are going to come to that time, and then the afternoon one is in Hewlett 200 which is where I believe your midterm was. Okay. What I wanna do today is I wanna talk about the more modern practices in dealing with XML processing. I think XML is kind of neat. It's not exactly a new technology although it's not exactly young either.

All of you have some exposure to XML processing because you had to do at least a little bit of it, or understand a little bit of it, for Assignments 4 and 6, and when you got to Assignment 6 you really did real XML processing because you were forced to deal with the X-pack parser, this open source [inaudible] that happened to be installed on all our machines here, and you used the X-pack parser to get all the titles out of there, and all the documents, and all the URLs and things like that. What I wanna do today is I wanna show you two different XML parsing models as they're supported by Python.

One of the takeaway points is that as opposed to C and C++, Java and Python, because we're talking about Python, both have built-in functionality for dealing with Internet connectivity, and in particular, XML processing. C and C++ and their specifications were laid down to rest for the most part well before the Internet became all savvy, and well before XML really had a presence in the Internet community.

Java and Python are much younger languages, they're kind of equal age with XML, and so all of them kind of – XML and Python grew up together. There actually is built-in support for XML processing, and Python I wanna use XML processing as our first gesture to Internet programming to see why Internet programming is so much easier or at least more gracefully supported by the Python language. Okay.

So let me just put a dummy fragment of XML up on the board. Let me write it this way. I will write it as entry and I will say address, let me say number is 2210 number and then continuing, and emphasizing the fact that this is all one big stream of text. Street, Hope Lane, this is where I grew up, suburban New Jersey, and close tag, this right there, and then you get the gist, and I'll just end the address part of it, and then finally just open up another tag, and then let your imagination run wild as to what my phone number was.

The reason I'm drawing it this way, is because this is more or less how the text came to you for Assignment 4 and Assignment 6's purposes. Okay. And the reason I'm doing this is as opposed to drawing it as this hyperstructure text tree which I actually will draw it that way in a minute for a different reason, this emphasizes the fact that even though there is structure and an hierarchy to the XML, that the XML parser we're familiar with from Assignment 6 and Assignment 4 actually was a stream-based parser which means it doesn't store it in memory.

I'm sorry, it stores only a constant subset of the entire text stream in memory at any one moment, and as it gets enough text because it sees the end of a start tag or the end of an end tag or it sees some character data, either the end of a stream of it or up to some maximum of characters, you know that it fired little events that we installed into the XML parser. Okay.

So right there, and right there, and right there, all of those arrows point to the place in a character stream where you would have seen some kind of start element tag handler invoked. Okay. And right here, you would have seen some character data handler invoked on your behalf. Right here, you would have seen an end tag handler invoked. Okay. And the stream-based parser that you did or you saw done for you for Assignment 6 is the opposite of functional programming.

It relies on state and side effect to keep track of where you are, and you know that when you read something like this that you're probably inside an address in an entry tag just based on the context of the individual example, and as you leave a number you know that you probably very recently read in an actual number as a text element. Okay.

What Python does, because it's object oriented, and C from Assignments 4 and 6 was not, Python actually uses a more modern approach that I think is a lot more graceful where via two packages from – there's actually a package called urllib, it's old and compared to the other package that has more modern support for more URL functionality, the snazzy name for it is urllib 2, I'm gonna go ahead and import this function called urlopen.

It is basically the equivalent of f open, but you give it not the path to a file name on your hard disk, but you give it a URL, and you just pray, because I don't care about error checking at the moment, that it actually points to a real document somewhere. Okay. From xml.sax import, these two functions make parser, and then a class which I'm gonna talk about called content handler, and then I'm also gonna import sys, all of it because I want to be able to print to standard L. Urlopen is the equivalent of f open. Okay.

It just works beautifully. It's a much more graceful implementation of URL connection, the URL connection class that you're familiar with the two middle assignments in the course. Make parser right here is what's called a factory function. It just promises to hand you back an XML parser that responds to a certain number of methods.

This content handler is actually an interface for – or it's a base class that has to be subclassed, and the subclass has to implement certain methods that actually have functionality associated with them so that it knows what to do when it reads a start tag, and when it reads an end tag, and when it reads character data. Okay. Does that make sense? Okay.

So here is the gist of what I want my XML parser thing to do. I'm really gonna deal with the understanding that some XML document is gonna have a title tag somewhere near the top. This happens to be the title of the entire feed, and it was irrelevant for the purposes of Assignment 4 and Assignment 6. I was interested in the title of the actual articles. There's actually a tag called channel. Inside you see things called items, and inside that, you see titles something like that.

There is a link and there was a description. I don't care about those. And you would see the same thing several times. Okay. Ends channel, ends – that's interesting. What I want to do is I want to write enough of a program that actually can read a URL, assume it's a URL that leads you to one of the RSS feeds that you were dealing with in earlier assignments, and I want to go in an extract all of the titles and just print from the standard out.

So this as content, and the same things in other item tags are of interest to me. I want to use a stream-based parser to do it, and I want to do it in Python. Okay. Just assume that content handler knows how to detect whether or not it's inside an item tag or a title tag within an item tag or it's leaving one or it's actually dealing with character tags.

There'll be more this, and this is really the heart of the code right here, but what I really want to happen is I wanna define a function called list feed titles, and I'm just gonna give it a URL, or I'm just assuming the URL is in fact something like Washington Post, anything you dealt with from Assignment 6. We're gonna use blog RSS feeds because they're campier, but I will just use those as an example. What I want to do is I want to not bother with error checking, I wanna call urlopen.

That basically gives me the file star that knows how to extract text from some remote document at that URL. Parser is equal to make parser. Now there's a little bit of mystery as to what that's doing. You have to just assume that it's some XML parser, and that this make parser thing is just returning a generic XML parser that doesn't know what type of XML document it's gonna be processing. Okay. But you know enough about XML that they all kind of have the same idea.

They're textural, but they're hyperstructured. You don't know the tag names that are relevant to any one particular RSS feed, but we're gonna make this particular parser relevant to RSS news feeds. So it's gonna eventually be able to detect things like channel and item and title within item. Okay. There's one method called set content handler – I'm just gonna do this.

I haven't devised this class yet, but I can tell you at this moment that I'm just gonna say RSS handler is gonna be a subclass of this thing called content handler, and from a generic standpoint, all you know is that by installing an instance of this type of class here, that we're gonna implant functionality in this class that knows how to take a document of this structure right here, and just print out all the titles. Does that make sense? Okay.

So now the parser has gone from generic to RSS news feed specific. Then I'm gonna do this: parser.parse info. So the only two methods that you need to rely on being available to your parser is the set content handler. You don't have to call it up, but then your parser wouldn't do very much, and the parse method which is kind of the method you would expect to send to your parser object. Okay. If I don't call this, the thing that's installed by default is an instance of one of these things. Let me just show you what the implementation of content handler looks like.

Class, I haven't actually looked at the implementation behind the scenes, but it behaves more or less like this. There are five methods that are of interest, but I'm only gonna focus on three of them. There's an init method certainly, but I'm most concerned about this, start element takes self, it takes tag, and it takes attributes. There's another method called end element that takes self, it takes the tag that's being ended, and that's it.

It also has a method called characters that takes the self which is equivalent of this pointer, it also takes data, and that's the signature for that. There's also a start document and an end document method. I'm not worried about those because I don't have to do anything specific for those. There's also a default constructor. Okay.

When you deal with characters not surprisingly this gets invoked anytime the parser is parsing, and it's halfway through something that's clearly textual – not halfway through, either entirely the way through or the character data, the stream of text that is there is so long that it really just doesn't wanna build an arbitrarily large string, so it goes up to some max and hands the character data over to the first of what will be several character method indications. Does that make sense to people? Okay.

When it gets to something like that, it would invoke the start element method, passing address as tag, and attrs, in this case attributes would be an empty dictionary. This is supposed to collect all of the things that are passed as arguments to a tag. You normally see that more in HTML. You certainly see it in XML as well, but when you see a tag like this – hold on, I'm not doing very well. Let's just do that right there, and maybe you have a couple of other things. And you know what I'm structurally getting at there.

This happens to be an HML anchor tag where the tag name is A. Those three attributes, one clearly is the h ref that's associated with the Google URL. This, that, and that would be stored as three keys in a dictionary. That, that, and that would be the value that's attached to those keys, and they would be all bundled in a Python dictionary that passed right there. Does that make sense? Now we don't have to worry about that in an example, and we also don't have to worry about it in the way I'm gonna be pulling the RSS news feeds. Okay.

This does the same thing in response to something like that. It actually passes the tag name to it so it knows which particular tag is ending. Okay. You should be able to infer it by a clean XML document because if some tag is ending, it's supposed to be the one that most recently started that hasn't ended already, but nonetheless this is invoked on your behalf because you may want to do some special processing when you notice that you're really at the end. Okay.

Now if I didn't have this line right here, that would be the class that handles all of the XML as it's pull through in this stream-based manner. The implementation of all three of these things is the equivalent of this. Now that isn't real Python, but you know what I mean when I put that there. Okay.

Well, the [inaudible] isn't supposed to use curly braces because you're not supposed to use curly braces at all in Python. Okay. At least not as block delineators, so the keyword that's used in Python is the word pass which basically is a placeholder which means I am nothing, but I'm functioning entirely as a no-op implementation. Okay.

What I wanna do, I wanna keep that there, is I want to, just by protocol you normally subclass that there so that you get the full suite of methods that should be available to a content handler by default, but you displace the ones that aren't good enough, and certainly start element, and end element, and characters actually have to do some kind of bookkeeping, and in some cases some printing.

If I really want to print that, and that out, that's certainly not gonna do it. It really would in this context, if I didn't have this third of the four lines right here, it would really call those four methods of those three methods up there, but all of them would be like, okay, you called me, great, I'm just returning. Okay.

What I wanna do is I wanna print out the character data that's inside every single title tag that's inside every single item tag, and I say it that way because I do not want to print out this title up here. Okay. Make sense? So what I wanna do is I wanna implement this thing called RSS handler. I would name it a little bit better if I had more space on the board, but I didn't so I used standard RSS handler.

And I want to design it as a class where it keeps just enough information to know whether or not it's inside an item tag, whether it's inside an item tag and inside a title tag, whether it's inside a title tag but not inside a title tag. Things like that. Okay. Because we're gonna use some Booleans that are tracked by the content handler class to figure out whether or not the character data that's actually being passed every single time to the character's method should incidentally be printed. Okay.

There's no way to control when characters gets called. It always gets called. It's gonna get called right there and right there. It would get called right there, and for all of the other things that are textual, but I only wanna print out these right there. Does that make sense? Yes, no? Okay. So the way you subclass in Python – actually I wish I had like three weeks to talk about inheritance in all the different languages because it's really, really interesting stuff, but I'll try to compress all the parts from three weeks of lectures into like two or three minutes.

To build a subclass in Python, you just define a class – you know what – class RSS handler, and you do this, you just say – and this right here is the equivalent of extends from Java. Okay. In the new version of Python, it's actually the 2.6 version of Python, I think it's 2.5 or 2.6, I forget, everything subclasses that object type I showed you on Friday by default in the same way that most classes by default – I'm sorry, all classes by default subclass java.lang.object in Java unless you specify which class it's subclassing.

I have to do the same thing here. If I didn't provide this it would just subclass the built-in object object, but I want it to subclass that so it has implementations of those three methods already. Okay. Does that make sense? I have to do a few things. I do have to define a constructor. The reason I do will become clear in a second. What I have to do here is I have to call content handler.

This is how you basically do the equivalent of a super call in Java. You have to say since I'm subclassing content handler, I have no idea what's involved in the construction of the pure content handler portion of it, but I should just make sure I call it in case there is something relevant there. But then what I want to do is I want to introduce to Booleans, just on the fly, it's just the Python way, in title is equal to false, and then all the sudden it's fairly clear I think what those Booleans are gonna be doing.

The double underscore at the front is a gesture to the Python environment that these are supposed to be treated as private variables. It's not formally enforced. All it really does is it actually mangles the names of those private fields to be something else. We can internally refer to these things right here, but it actually changes the variable names in the memory model so that if you try to reference these things from outside the class you have a harder time doing it.

It's not encapsulation, it's actually more of a hack than that. But nonetheless this is just convention for dealing with privacy in Python classes. As far as the characters data, the characters method is the easiest of the three to implement because there's no tricks. Character where I pass in self and I pass in data, normally I would just do this, sys.std.out.write, is the equivalent of print f. I would just print the data.

But do you recognize that that would print all of the character data in the order that it comes in the document, but nonetheless I don't want that. Okay. So what you need to do is you have to assume that in title will be set to true when we really are inside a title tag because then if the character method is being invoked, it's title text data as opposed to arbitrary text data. Okay. Does that make sense?

So the start element, and the end element, that's more about maintaining these two Booleans than anything else. Does that make sense? Okay. So I have two different scenarios that I want to dispatch against inside start element. I have self, I have a tag, which is a string that's going to be like title, or link or channel, or title or whatever, and then I have attributes which are being ignored here.

And I have two different if tests that I wanna worry about. If it is the case that tag double equals item – you can compare strings, you can double equals like that – that means with a normal RSS feed, you're actually entering an item subdocument where you expect to find a title, and a link, and a guide and a start data and a description and all that stuff that you're familiar with from Assignment 4 and 6.

If that is the case, all I wanna do is I wanna set self in item to be equal to true. Okay. You don't have to go on the next line if you only have a one-line id block. Otherwise, if tag is equal to title – the way I'm doing this is actually incorrect, but let me do this much – and I'm assuming that in title is being set equal to true because I've very recently stepped into not only an item tag, but also a title tag. Okay. However, this is not quite the right test.

You are guaranteed to only have one level of items tags in an RSS news feed, but title actually comes at two different levels. Now you could argue that that was silly, they could have used any one of a million different strings so they didn't have to overload the meaning of title in RSS, but I didn't design it, so I have to deal with their design. So if the tag is equal to title, and incidentally end is really the key word, and it's also the case then I'm in an item, then go ahead and catalog the underscore underscore in title to be true. Okay.

So do you understand how start element – I don't wanna say it's sophisticated, but it's not trivial either because you really have all these little side effects associated with start element calls where the two Booleans inside are constantly tickering and toggling betweens trues and falses, and technically you need both of them to be true, although we're only gonna check in title to be true. Technically both have to be true if you're really gonna be printing out title data. Okay.

As far as the end element is concerned, self-tag, if it is the case that tag double equals title, and self in title, then I know that I very recently responded to some character methods that printed some stuff out. Does that all make sense to people? This right here does not print a new line, so if I really want to publish the title on its own line, then kind of is a little last gesture to the title that just got recently printed, I will do this. Okay.

And I will also – I will set self in title to be equal to false. If it's the case the tag double equals item, then I will go ahead and not print anything, and do that right there. Okay. So the start and element tags are all about taking falses to trues and trues to falses, and the character method is incidentally printing character data when both of the Booleans are true. Okay. Do you understand what I mean when I say that's the opposite of functional programming? It's all about side effects. Okay.

So just so that you know that this is not really theory, and that it really works, let me show you three ways to examine an RSS feed. I'm not saying that you should be reading this blog, and I'm not endorsing it at all, but hopefully it'll – bring this up – somehow it's the case that when you do www.gizmodo.com you get a blog about cool and trendy gadgets about people who are often very bitter. Okay. But this is the presentation, and you can see just be scanning through all this stuff that there's titles, there's content, there's probably stuff that's pulled from URLs.

I can't really show the entire web page here, but when you visit gizmodo.com – I don't know how it constructs the Web page, I would not be surprised if it constructs the Web page from its very own RSS feeds. Okay. It might not do that because the Web page structure doesn't change all that often because the RSS feeds don't change all that often.

So in theory they could build them from the RSS news feeds, but they might just build the HTML document once, and then use the same static HTML document until either a new article comes along or the comments are updated or something like that. Okay. What you may not know is that you can just look at the raw RSS document, and this really is Firefox 3 beta's way of displaying an RSS feed as if it was an HTML page. It has a slightly different presentation.

This is the type of presentation that Apple gives to its – I'm sorry, Firefox gives to all RSS news feeds, and it looks more apple.com ish in its default presentation. So there's that. I actually identified the URL of the RSS news feed to the Web browser, and those are the types of things you dealt with within Assignments 4 and 6. Here's another way to actually look at the content.

This is the least – I think it's the most informative from a networking standpoint. You're not gonna be able to really digest what just happens here, but you understand in both cases when I actually viewed the full gizmodo Website, and also when I viewed the RSS news feed that my computer is talking to some other server. I know you know that much. The way they communicate is they actually pass text back and forth to one another, and the text somehow encodes what document is requested, and what the document really is.

It's not a very long conversation, in fact, my computer speaks once to the gizmodo server, gizmodo hears what I say, and in response actually publishes a much longer stream of text back to my computer which is either gonna be the RSS or the HTML. So what I wanna do here is in two phases – is this everything? No it is not. So now I'm realizing that you didn't see the whole Web page did you? There's the URL for the RSS news feed. Okay. Better.

Okay, so what I wanna do here is I wanna bring this back. I want to telnet to – let me see what the – feeds.gawker.com – before I press enter, let's copy this thing. This is gonna look like really random stuff. Fortunately the response is not gonna be very long. Let me scroll back up to show you what I type in here. Right there. You see all of that? Okay.

This line right here, basically telnet is an instruction to pick up the phone, feed.gawker.com tells you which house you're calling, and 80 tells you which of the 2 to the 16th different phone numbers you're calling. Okay. And the HTTP server that's running at feeds.gawker.com is listening on a port called 80 for HTTP requests. My HTTP request happens to be structured this way right here. Okay. You probably heard of things like get and post just as verbs that come up in programming. Okay.

This is just an instruction, when I'm speaking to the phone after they answer, and they answer right here. It says so. It connected. I'm interested in getting this document, and this is the actual protocol. The more modern protocol is 1.1, but I didn't know for a fact that it would actually be implementing that, so I actually went with the one that's been around for several years, and in response when it hears that, and basically the equivalent of a period is two carriage returns, it comes back with that, and a little bit more.

You know enough about – I know you have digested HTTP response code of 302 from Assignment 6 where that involved the redirection. Okay. So what this is saying is that thank you very much I have it, but I want you to actually consult this URL to get the RSS feed instead. So what I'm gonna do is I'm gonna say, okay, that's fine, I'll call these people, feedproxy.feedburner.com, and I will ask for this document instead, and be very fast about it because otherwise it hangs up the phone, thinks it's a prank call, and that should be enough.

It takes a few seconds unless I messed up. Actually this would not be good. Well, I don't know what's up. I didn't type it incorrectly did I? Oh that's disappointing. Let's try it again. It's really stuck. Whew, that's not good. I don't know what's going on because it won't hang up the phone. Okay, well, anyway, I guess I lose. Here is the response. Okay. It comes back with this, and it's hard to get a lot out of that, but I can tell you right now that the thing I'm interested in, let me see if I can find it, there's the title that I'm trying to ignore up there.

There should be an item somewhere. There's just you see all these things that you're familiar with. I don't see an item. Hold on a second. Oh, there's one. That's good. Okay. You can see that right there. Okay. So that's the one tag, and it actually spits back to the computer this stream of text that happens to have new lines in it so it looks like a human-readable document. Okay. It would have worked out just fine if it didn't have all these new lines inside of it. And that's the text that comes back to me in response to say a URL open call.

And I actually say thanks for that text, I'm gonna hook this in file URL open result to the parser, and I'm gonna call parse against it, and it's gonna pull all this text in, start element, start element, start element, characters, end element, cart, that kind of thing is gonna happen. Okay. Based on the text that's been read in. Okay. Does that make sense to people? Okay. So that worked.

The advantage of the type of parsing I['m discussing right here is that some XML documents, and not this one, it's not really that large, but in principal XML documents can be as large as they wanna be. We're probably dealing with a few kilobytes of data here, but there's nothing to say that you can't have an XML representation of like the English dictionary or like all world languages or something like that. Okay.

There might be an XML document that stores all of like the books written between 1700 and 1800. That's gonna be more than a few kilobytes, but the advantage of this type of parser, and the one you're most familiar with now, is that it doesn't try to store the entire document in memory.

It reads in jut enough to make some progress and as long as you can deal with the XML document in it's read-only capacity, and you only need to make one pass over it, you can actually digest the entire thing in small pieces, and discard what you've read in, after you've actually processed it to the degree you need to. We happen to be using the content handler subclass in Python to get the job done. Does that make sense? Okay.

That isn't the only way you can process XML documents, and the way that was not emphasized in any of the assignments and not in this example, is that there are some XML enthusiasts who for their reasons, actually like to put the entire XML document in memory. Now there's an advantage to that because if you can come up with an in-memory model of an entire XML document, you can do more than just print it.

You can reverse it or you can snip parts out, and you can sanitize parts of it, or you can restructure it, or add new items dynamically into the tree in ways you can't do if you're using the stream based read-only one pass approach. Does that make sense to everybody? Okay. So there was a second model of XML processing. This SAX model – SAX stands for Simple API for XML. Very popular low memory imprint. It's a great way – like I have an opinion – it's actually a common way to process XML documents.

Now what I'm gonna do is I'm gonna punt on this. I'm not gonna be able to get this entire example – I don't think I'm going to. Go ahead, I'm sorry. I'm gonna draw the XML document a little differently. Okay. This is an important way of digesting, and understanding how XML can be processed because this is really how web browsers digest, and manage, and render HTML. Okay.

If Web pages used a stream based approach to building the Web page after running the HTML, and it discarded everything, there would be nothing dynamic about Web pages at all, except for the fact that it could be constructed dynamically, but there'd be no interactivity at all.

You know how like when you press a button, like you're shopping at Amazon or something like that, and then the page updates and your shopping cart gets one more item in it, that's because the in memory model of the HTML document is actually updated by that button click. So you actually update the HTML document, not the original one, but the one that's being managed by your browser. Okay. You can do something similar with XML.

My first example, what I'm motivated toward isn't really a dynamic example in that it's gonna change the tree, but I certainly could if I wanted to. Let me just draw an item, subportion item, you gotta be careful about this – there's title, hello there, end item, there is a link ACTP, blah, blah, blah, and then there's a description tag. I'll abbreviate it that way.

It's what it was, and then there's the end of the item. Okay. That's a subtree of the overall XML tree that is of interest to me for this current example, and you can imagine that if there are a stream of these things, one after the other, effectively a stream representation of an array of item objects. Okay. They're surrounded by this thing I'll call channel. That gives this very left to right view of an XML document. Okay.

The overall thing has one tag up here called RSS, the end tag is RSS right here, but the root of the entire document which is managed by this node right here, is on the left of the drawing as opposed to the top of the drawing. But you've heard about XML documents. You may not have had too much practice with them, but I know you've at least seen XML documents before.

What some XML processing models go with is a tree representation of the XML document. Here's an actual node that has the tag RSS inside of it. Okay. It has several children which themselves have several children, and this may actually have as one of its children a channel node. Okay. So do you understand how I've basically taken this as a document? I've made sure that and that are associated.

It's like these are like two ropes at the end of a hammock and I've brought them together, and of course the hammock's lying on the ground unless I actually rotate up like I am over here. Okay. This channel would have several item nodes. The items themselves would have title, and link, and description, and even link actually is the same type of node. It actually has a little node down there.

I'm drawing it differently because it's actually a text node that has associated with it the actual text that's inside of it. Does that make sense? Now that isn't a very sophisticated drawing, it looks a little scary, like the worst mobile ever, but you can imagine how that will translate to an end area tree in memory. Okay. XML, as a standard, every single start tag is supposed to be closed by an end tag. There's much more structure with an XML document than there is even with HTML.

It turns out parsing HTML is a more difficult problem than parsing XML because XML has much more required structure to it. You can imagine something like this existing in memory where this and this and this and this are – and actually all five of this rectangles are categorized by a struct or a class that's just called node.

These four right here are specific types of nodes and C, C++ or Python, they would be modeled as subclasses of nodes that I would call element, that's what Python calls them anyway. Elements emphasize the fact that they correspond to things that are delimited by these greater than or less than signs. This would be a peer subclass of element called text node or called text in Python that actually carries whatever text is associated with something like this or that or that. Okay. Does that make sense?

There is another package in Python that's dedicated to the DOM model of XML processing. DOM stands for Document Object Model. Object is clearly there for, I think, obvious reasons, but it has this object-oriented approach of modeling an entire XML document, that's why the D is in DOM. Okay. And then if you build something like this in memory, you have an entire in-memory tree that represents the XML document, you can look for all of the nodes that have item inside them. Okay.

Then for all of those you can look underneath all of those subtrees for the item, for the node that have title inside of them. I know that all of those title nodes actually have one child that are text nodes that have text associated with them. Does that make sense? Okay. Conceptually, I think it's easy to digest.

It is a little bit of work to understand what all the methods in Python or Java or Ruby or any modern language that has Web capability built into it can do, but if I wanted to construct an RSS news feed, what I could do is I could collect all of my titles, and maybe I have normal C structs with title fields, and link fields and things like that. I could actually dynamically insert and construct all of these things. Okay.

And once I construct this, I can serialize it, traverse it in this in-order matter like you're very familiar with doing just regular binary search trees that way. You can do it in reversal of this, know how to print on behalf of this you print a start tag, you may print any attributes that are associated with it.

You fully marshal all of this in sequence then when you come back to here, you print the end tag associated with that, and if you follow that recursive formula throughout the entire thing, you can construct an in-memory model of your RSS news feed and then serialize it, do some kind of to print function or something like that. Okay.

That type of functionality is certainly built in to Java, which you probably do not hit on in 106A, but you certainly will hit on it in 108. It also exists in Python. Okay. Does that make sense? Now there's one example in the handout. I will try to get to it in the first few minutes of Wednesday's lecture. As far as Wednesday goes, I'm not sure. I do wanna bring you in because I do wanna talk about some other languages.

I'm trying to coax a very smart, but very young, and very nervous colleague of mine who is a little worried about speaking in front of people his age as an authority in the language, but he's super smart, smarter than I ever will be, and certainly was when I was 22 years old. He knows this language called Haskell really, really well.

He also knows the language that I kind of know pretty well called ML which is like Scheme, but with data typing. Okay. I certainly wanna talk about those, so what is going to be presented to you on Wednesday is really a mystery, but there will be something. So by all means, come on Wednesday. Remember the section tomorrow, and you have an assignment due Thursday night. Okay.

[End of Audio]

Duration: 50 minutes