Extracted from:

Data Crunching

Solve Everyday Problems Using Java, Python, and More

Chapter 4

XML

Unless you've been living under a rock for the last decade, you'll know that flat text is passé. These days, almost everything is represented using some kind of markup: HTML for web pages and its cousin XML for spreadsheets, web server configuration files, and books like this one. If you're crunching data, the odds are pretty good that your input, your output, or both will use some kind of marked-up format.

Luckily, there's a whole bunch of tools for dealing with marked-up data. After a quick introduction to HTML and XML, this chapter describes DOM and SAX, the two most popular of those tools, which have been implemented in many languages. After that, we take a look at XSLT, a language designed expressly to inspect and transform XML.

One thing we *won't* discuss in this chapter is WYSIWYG tools for editing and transforming XML. Like programmable editors (Section 2.5, *Including One File in Another*, on page 27), these are often the quickest way to solve small problems in isolation, but it's difficult to connect them with other tools to make a data crunching pipeline. They're still handy to have on your desktop, though; two I'd particularly recommend are Altova's XMLSpy,[1] a high-end commercial product, and the XMLBuddy [2] plugin for Eclipse.

4.1 A Quick Introduction

If you've ever written a web page, you probably already know most of what's in this section. Feel free to skip over it or to watch *The Simpsons* with one eye while reading it.

[1] `http://www.altova.com/products_ide.html`
[2] `http://xmlbuddy.com`

History

In the beginning, there was GML, the Generalized Markup Language, and its successor SGML, the Standard Generalized Markup Language. These were developed between 1969 and 1986 by Charles Goldfarb and others at IBM as a way of adding information to medical and legal documents so that computers could process them. For example, instead of just having a person's name, like Alfred E. Neumann, an SGML document would have:

```
<person role="litigant">Alfred E. Neumann</person>
```

or even:

```
<person role="litigant">
  <given-name>Alfred E.</given-name>
  <surname>Neumann</surname>
</person>
```

so that people could search databases to find cases in which Mr. Neumann had sued, as opposed to ones in which he had been sued.

SGML was a great invention but had one major drawback: its size. By the mid-1980s, the full SGML specification was more than 500 pages long. Writing software that followed all those rules was so daunting that only a few companies ever tried.

In 1989, though, when Tim Berners-Lee needed a way to describe titles, underlining, and cross-references for something he called the World Wide Web, he drew inspiration from SGML. His format, called the HyperText Markup Language, or HTML, had only a small, fixed set of tags. Some of these were semantic, such as "this is a level two heading", while others had a purely visual role, like the ones that *italicized* text. SGML purists sneered at HTML's simplicity, but that simplicity meant that anyone could write it. Within just a few years, millions of people around the world did exactly that.

Almost as soon as HTML appeared, programmers began adding their own extensions to it. This soon led to XML, the eXtensible Markup Language, the first version of which was approved in 1998. XML isn't actually a markup language, like HTML. Instead, it is a set of rules that particular markup languages must obey. While it is much more complex than HTML, it is still simpler than SGML, and has taken the world by storm.

As a result, three kinds of marked-up documents are common:

- Legacy HTML, which doesn't obey XML's rules. For example, web designers who type tags in by hand often don't bother to mark the end of a paragraph; instead, they just start a new one, trusting the browser to know what they mean. This makes automatic processing painful and error-prone. If you

have to deal with legacy HTML, the best approach is to convert it to proper XHTML (using Perl's HTML::Tidy module, for example) and use the tools and techniques discussed in this chapter.

- XHTML, which is classic HTML that conforms to XML's rules. Most WYSIWYG editors and software libraries now generate this, but it's worth checking a few samples before you start crunching: much of the XHTML in the world is still typed in by hand, which means that errors are common.
- Everything else. XML-conformant markup is used for everything from images (SVG) to music (MusiXML), chemical formulae (ChemML), and news headlines (RSS). Almost all of this data is created by machines, so it is both more likely to be correctly formatted and more verbose than handwritten HTML or XHTML.

Formatting Rules

An XML document is a tree containing *elements* and *text*. Elements are shown using *tags*; the text is the stuff between the tags.

elements
text
tags

At their simplest, tags are keywords enclosed in angle brackets, like <*this*> and <*that*>. Every opening tag of the form <*example*> must be matched by the corresponding closing tag </*example*>. (Note the slash before the tag name.) Everything between the two is the element's content. If there's no content—in other words, if the closing tag comes immediately after the opening tag, as in <*blah*></*blah*>—the shorthand notation <*blah*/> (with the slash at the *end* of the tag) can be used.

Since XML documents are trees, elements must be closed in the reverse order that they were opened. This means that <*a*><*b*><*c*></*c*></*b*></*a*> is legal, but <*a*><*b*><*c*></*a*></*b*></*c*> isn't. XML documents must also have a single *root element*; in other words, each document must be a single tree, not a forest.

root element

Let's look at an HTML page for example. Some common, basic tags used in HTML documents include:

<*html*>	HTML page (must be the document root)
<*body*>	Body of page (visible content)
<*h1*>	Level one heading
<*h2*>, <*h3*>, <*h4*>	Subheading, subsubheading, etc.
<*p*>	Paragraph
<*em*>	Emphasized (italics)
<*strong*>	Emphasized (bold)

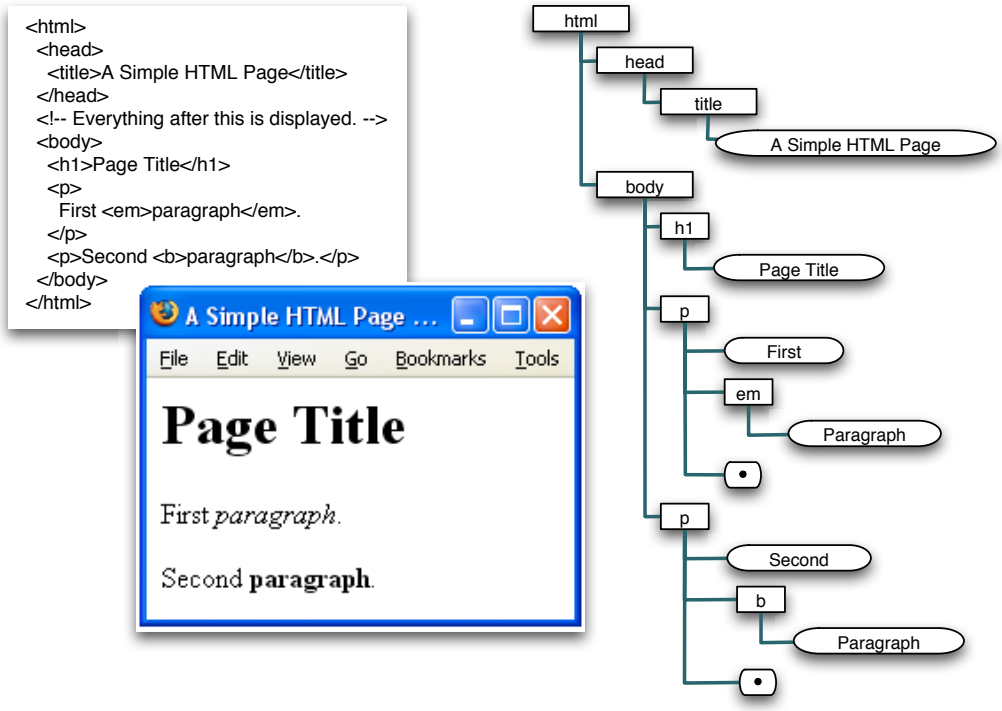

Figure 4.1: HTML DOCUMENT TREE

Now take a look at Figure 4.1 The source to the HTML page, which is stored as a text file, is shown at the upper-left. When this file is handed to Firefox, it translates the content into a tree in memory (shown on the right) and then displays that tree (you can probably guess by now).

Every page has exactly one <*html*> element, which must be the root of the document. This may contain one <*head*> element and *must* contain one <*body*> element. The head stores information like the page's author, version number, and so on; this information isn't displayed but is used by search engines and other tools. The body contains all the text that's to be shown to the user, along with references to images, sound files, and other content discussed in a moment. Finally, everything between <!-- and --> is a comment.

Markup gets more interesting when you start adding attributes to the elements. An *attribute* is just a name/value pair associated with an element. Attributes are written inside the opening tag of an element as name="value". An element may

Joe Asks...

When Should I Use Attributes Rather Than Elements?

Strictly speaking, attributes are redundant, since

```
<a b="c">
  <d e="f"/>
</a>
```

could always be written as

```
<a>
  <a-b>c</a-b>
  <d><d-e>f</d-e></d>
</a>
```

Attributes are a lot easier for people to type in, though, so the question arises: when should you use attributes *vs.* nested elements?

The answer is that you should use attributes only when

- each value can occur at most once for any element,
- the order of the values doesn't matter,
- those values have no internal structure, and
- the values themselves are short.

The first two rules come from XML's insistence that any attribute appear only once for a given tag and that attributes can't contain elements. The third is mostly a question of readability: if you try to put too much data in an attribute, it becomes impossible to see the document's structure.

have any number of attributes, but any attribute can be used only once with any particular element. Every attribute must have a value, and every value must be quoted. (Most browsers still accept legacy HTML such as <*tag color=blue*> and <*tag invisible*>, but neither is strictly legal: the first because the value isn't quoted, the second because the value is missing entirely.)

If you want to put a quotation mark inside an attribute value, or an angle bracket in text, you have to escape it. Escape sequences start with & and end with ;. In between, you can put the character's numeric code, as in *£* (which should display £). Alternatively, if the character you want is commonly used, it may have a descriptive name. The most common are *<*, *>*, *&*, and *"*, for <, >, &, and " respectively. You can find a complete list online.[3]

[3] http://www.w3.org/TR/REC-html40/sgml/entities.html

Links

HTML really earns its H when you start adding links between pages. Links are specified using the <a> element with an href= attribute, whose value specifies what the link points at. The text inside the element gives the viewer something to click on to follow the link. For example, the link

```
<a href="http://www.pragprog.com">Pragmatic!</a>
```

is displayed as—oh, you know how links work.

Confusingly, <a> can also be used to create an *anchor* inside a document. An anchor is just a point in the document that some other document can point to. To specify an anchor, use an <a> element with a name= attribute instead of an href= attribute. The name= attribute's value can be pretty much any string you want, although almost everyone uses descriptive labels like *first_citation* or *chapter:testing*. Another document can then point to that anchor by specifying its name after a # in an href=, as in *http://com.com/index.html#mid*.

4.2 SAX

SAX, the Simple API for XML, turns an XML document into a stream of events; users then write code to handle those events. This "inside out" model will be familiar to anyone who has done any GUI programming: instead of writing an entire program top-down, you hand control over to a framework, which calls your code when it needs to do so.

Like everything else, SAX has strengths and weaknesses. Its main strengths are that simple things are simple to write and that it can handle very large documents, since it stores only a small fragment of the document in memory at any time.

That feature is also SAX's greatest weakness. If you want to modify the document's structure, or if you need a broader context in order to decide what to do at some point, you have to keep track of where you are manually. It's rather like sorting or reversing the lines in a file when all you have is a method to read one line at a time: you have to create the necessary data structures yourself.

Getting Started

Python's implementation of SAX lives in the xml.sax module. This module contains a method called parse(), which does what its name suggests: parses an XML document. It takes an object as an argument; each time it encounters something interesting, like a tag or a piece of text, it calls one of that object's methods. (Java's org.xml.sax and Perl's XML::Parser::PerlSax work in more or less the same way.)

The object you pass to parse() must be derived from a class called ContentHandler, which is also in the xml.sax module. By default, ContentHandler's methods do nothing; in order to actually process the XML that's being parsed, you have to override one or more of the methods that handle

- the start and end of the entire document,
- the start and end of individual elements,
- text, and
- ignorable whitespace (which we will ignore for now).

If you've ever done any GUI programming, this should seem eerily familiar. Overriding a method to handle an opening tag event is exactly like overriding the method on a button that handles mouse-down or the method on a text widget that handles character insertion. In each case, the framework (SAX, or your GUI toolkit) takes care of the routine stuff. All you provide is the little bit of logic that's specific to your application.

Let's look at an example. Suppose we want to print out all the opening tags in an XML document. The first step is to derive a class, which we'll call ShowTags, and override its startElement() method. This method takes two parameters: the name of the element and a dictionary of its attributes. We then create an instance of this class, and pass it to parse(), along with the stream the parser is to read. The whole thing is as simple as

```
import sys
from xml.sax import parse, ContentHandler

class ShowTags(ContentHandler):
    def startElement(self, name, attrs):
        print name

handler = ShowTags()
parse(sys.stdin, handler)
```

If we hand this little program the following document

```
<html>
<head>
<title>A Simple Document</title>
<meta name="author" content="Greg Wilson"/>
</head>
<body>
<h1>A Simple Document</h1>
<p>This document shows how SAX (the
   <b>S</b>imple <b>A</b>PI for <b>X</b>ML)
   works.
</p>
<hr/> <!-- put a line between paragraphs -->
<p align="center">And that's all.</p>
</body>
</html>
```

it prints

```
html
head
title
meta
body
h1
p
b
b
b
hr
p
```

Let's make the output a little more readable by indenting tags to show how elements are nested. To do this, we need to count how many opening and closing tags we've seen. The IndentTags class shown below does this by overriding endElement() as well as startElement():

```
import sys
from xml.sax import parse, ContentHandler
class IndentTags(ContentHandler):
    def __init__(self):
        self.indent = 0
    def startElement(self, name, attrs):
        print ' ' * self.indent + name
        self.indent += 1
    def endElement(self, name):
        self.indent -= 1
handler = IndentTags()
parse(sys.stdin, handler)
```

Its output for the same document is

```
html
 head
  title
  meta
 body
  h1
  p
   b
   b
   b
  hr
  p
```

Example: Creating an Attribute Inventory

You have lots of ways to specify how the elements in an XML document can be nested and what attributes they're allowed to have (see the sidebar on page 86). In practice, though, you often have nothing to work with except the document itself. In these cases, you often have to reverse engineer the document's format before you can start crunching it.