XML processing with Python, Part 1: SAX and DOM

STRUCTURAL ANALYSIS

XML and Python make a great team. With Python it is easy to control SAX as well as DOM parsers, which allow you to analyse structured documents. Andreas Jung explains how

Over the last few years XML has developed into a platform-independent standard exchange format for data and documents. Apart from the actual XML standard there are other standards, such as XSLT and XPATH, which relate to converting and accessing XML documents. Python and XML could therefore both be described as middleware – reason enough to have a closer look at the possibilities of XML processing with Python.

XML modules in Python

The standard distributions of Python 2.1 and 2.2 already contain the most important modules for XML processing. However, these do not cover all the functionalities that we require for the purposes of this article. The PyXML package provides a much greater functionality range. PyXML can be installed either as an rpm or directly from the sources using Distutils (*python setup.py install*). Binaries for Windows are also available for download.

Objectives

In the following example we will be using the XML file *pythonbooks.xml*. This file contains data for three Python books, which we will convert into a simple HTML table using various XML techniques. To simplify matters, a "book" (*<book>* tag) consists only of its title, author and publisher (Listing 1).

Listing 1: pythonbooks.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<pythonbooks>
    <book id="1">
      <title>Programming Python</title>
      <author>Mark Lutz</author>
      <publisher>O'Reilley</publisher>
    </book>
    <book id="2">
      <title>Python &amp; XML</title>
      <author>C. Jones &amp; F. Drake,
Jr.</author>
      <publisher>O'Reilley</publisher>
    </book>
    <book id="3">
      <title>Python Essential Reference
</title>
      <author>Guido van Rossum &amp; David
Beazley</author>
      <publisher>New Riders</publisher>
    </book>
</pythonbooks>
```

Keep it simple with SAX

SAX stands for "Simple API for XML". A SAX parser is essentially based on a callback API. This means a number of functions, which the application has registered for a certain event type, are called during the process of parsing an XML document. Such events typically include opening and closing XML tags, text and entities, but also parser errors, which are reported to the application as events. The most important component is the content handler, which implements the callback functions *startElement()*, *endElement()* and *characters()*. The content handler is registered with the SAX Parser via its *setContentHandler()* method. For example, for every opening

Table 1: SAX *ContentHandler* class methods

Method	Description
startDocument()	Call for starting parser
endDocument()	Call for terminating parser
startElement(name,attrs)	Call for opening tag <name>
endElement(name)	Call for closing tag </name>
characters(content)	Call for text

tag it calls *startElement()* with the tag's name and attribute list.

Listing 2 (*sax.py*) shows the implementation of the converter using the SAX parser. The actual application logic is contained within the individual if-then-else blocks. Table 1 shows the most important functions of the ContentHandler class.

DOM: A few sizes up

The Document Object Model (DOM) is defined by a number of standards set by the World Wide Web Consortium (W3C) and covers all aspects of XML processing. Unlike SAX, when the DOM parser parses an XML document it creates an internal hierarchical tree structure which the application can access using the DOM API. Figure 1 shows the internal structure

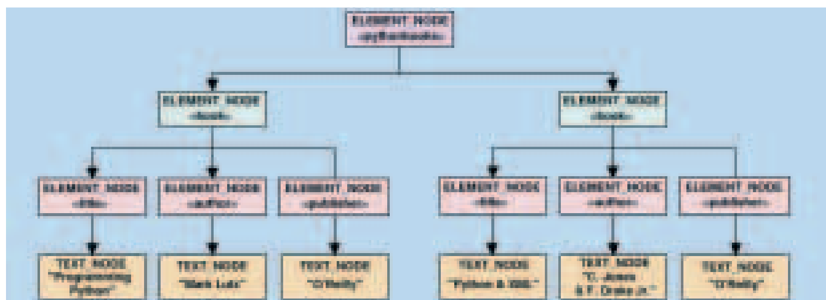


Figure 1: Internal XML structure for the example "Pythonbooks".

for the earlier XML example. There are various types of nodes on the tree (see Table 2) representing, for example, XML tags (ELEMENT_NODE) or text elements (TEXT_NODE) between XML tags. All nodes have a number of attributes that can be used to navigate a DOM tree (see Table 3).

The difference to SAX becomes obvious in the DOM implementation of our example (Listing 3). With DOM the application determines the processing sequence (with SAX the application reacts to the parser events). The simplest way of transferring an XML document to a DOM tree is via the *FromXmlStream()* method, which reads an XML document from an input stream, parses it and returns the top node of the DOM tree.

In our example we are first of all looking for all element nodes representing the <book> tag. The *getElementsByTagName()* method searches for all element nodes representing the tag in question. Once you have located these nodes you can retrieve the nodes for *<author>*, *<title>* and *<publisher>* in the same way and then extract the text contents of the corresponding tags. The

Listing 2: sax.py

```
from xml.sax import make_parser
from xml.sax.handler import ➥
ContentHandler

class BookHandler(ContentHandler):

    book = {}
    inside_tag = 0
    data = ""

    def startElement(self, el, ➥
attr):

        if el == "pythonbooks":
            print "<table>"
            print "<tr>"
            print ➥
"<th>Author(s)</th><th>Title</th➥
><th>Publisher</th>"
            print "</tr>"

        elif el == "book":  ➥
self.book = {}
        elif el in ➥
["author","publisher","title"]:
            self.inside_tag = 1

    def endElement(self, el):

        if el == "book":
            print "<tr>"
            print ➥
"<td>%s</td><td>%s</td><td>%s</td➥
>" % \

(self.book['author'], ➥
self.book['title'], ➥
self.book['publisher'])
                print "<tr>"

            elif el in ➥
["author","publisher","title"]:
                self.book[el] = ➥
self.data
                self.data = ''
                self.inside_tag = 0

        def characters(self, chars):
            if self.inside_tag:
                self.data+=chars

# Content handler
bh = BookHandler()

# Instantiate parser
parser = make_parser()

# Register content handler
parser.setContentHandler(bh)

# Parse XML file
fp = open('pythonbooks.xml','r')
parser.parse(fp)
```

Listing 3: dom.py

```
from xml.dom.ext.reader.Sax2 import ➥
FromXmlStream

def getText(nodelist):
    lst = []

    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            lst.append(node.data)

    return ''.join(lst)

def td(txt):
    print "<td>%s</td>" % txt,

fp = open('pythonbooks.xml','r')
dom = FromXmlStream(fp)

print "<table>"
print "<tr>"
print  "<th>Author(en)</th><th>Title</th><th>➥
Publisher</th>"
print "</tr>"

for book in dom.getElementsByTagName('book'):
    print "<tr>"

    for item in ['author','title','publisher']:

        node = ➥
book.getElementsByTagName(item)[0]
        td( getText(node.childNodes) )

    print "\n</tr>"

print "</table>"
```

Listing 4: dom1.py

```
from xml.dom.ext.reader.Sax2 import ➥
FromXmlStream
from xml.dom.ext import PrettyPrint

fp = open('pythonbooks.xml','r')
dom = FromXmlStream(fp)

# find 'pythonbooks' node
top_nodelist = ➥
dom.getElementsByTagName('pythonbooks')

# new 'book' node
new_book = dom.createElement('book')

# all child nodes for 'book'
new_author = dom.createElement('author')
new_author.appendChild( ➥
dom.createTextNode('Andreas Jung') )

new_title = dom.createElement('title')
new_title.appendChild( dom.createTextNode('XML➥
processing with Python') )

new_publisher = dom.createElement('publisher')
new_publisher.appendChild( ➥
dom.createTextNode('Linux Magazine') )

# link nodes
new_book.setAttribute('id', '4' )
new_book.appendChild(new_author)
new_book.appendChild(new_title)
new_book.appendChild(new_publisher)

# and attach new book to book DOM
top_nodelist[0].appendChild(new_book)

PrettyPrint(dom)
```

function *getText()* steps through every child node and tests whether it is a text node. If it is, the text is extracted.

Modifying a DOM tree

The great advantage of DOM is that its tree structure can be reorganised dynamically. *dom1.py* in Listing 4 shows how simple it is to add a new *<book>* element. The corresponding element nodes are created using *createElement(tagname)*, while text nodes are created with *createTextNode(text)*. The crucial point is the integration of the nodes into the tree structure. In our example the text nodes are appended to the element nodes using *appendChild()*. The element nodes for *title*, *author* and *publisher* are in turn appended as descendants of the newly created *book* node. In the last step the new "book" with all its child nodes is appended to the existing tree. We are using the *PrettyPrint()* utility to output the extended tree (see Listing 5 *pythonbook1.xml*).

Spoilt for choice

Whether you use SAX or DOM very much depends

Table 2: The most important DOM node types

Node type	Description
ELEMENT_NODE	element nodes (XML tag)
ATTRIBUTE_NODE	attribute nodes (XML tag attributes)
TEXT_NODE	text nodes (text within XML tags)
CDATA_SECTION_NODE	nodes for CDATA elements
ENTITY_NODE	XML entities (e.g. &)
ENTITY_REFERENCE_NODE	XML entity references (e. g. ®)
COMMENT_NODE	XML comments
DOCUMENT_NODE	document nodes
DOCUMENT_TYPE_NODE	document type definitions
DOCUMENT_FRAGMENT_NODE	document fragments
NOTATION_NODE	notation nodes

Table 3: Attributes and methods for all DOM nodes

Attribute/method	Description
attributes	node attributes
childNodes	lists all child nodes
firstChild	the first child node
lastChild	the last child node
nodeType	node type (see Table 2)
parentNode	the node directly above in the DOM tree
nextSibling/previousSibling	right/left sibling node
removeChild(childNode)	removes a child node
appendChild(newChild)	adds a new child node
insertBefore(newChild,refChild)	Inserts a new node before another child node

Table 4: ELEMENT_NODE API

Attribute/method	Description
tagName	name of the XML tag
getAttribute(name)	retrieves the value of an attribute for the node
getElementsByTagName(name)	retrieves a list of all descendant element nodes of the same name
setAttribute(attr, val)	adds a new attribute to the node
removeAttribute(attr)	removes an attribute from the node

Table 5: TEXT_NODE API

Attribute	Description
data	string representation of the text
length	length of the text

Table 6: SAX Parser vs. DOM Parser

SAX	DOM
+ fast	+ modification of XML documents possible
+ memory efficient	+ flexible navigation
– no modification of XML documents possible	– not suitable for very large XML documents
– no navigation possible within the XML document	– whole document in memory

Listing 5: pythonbook1.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE pythonbooks>
<pythonbooks>
  <book id='1'>
    <title>Programming Python</title>
    <author>Mark Lutz</author>
    <publisher>O'Reilley</publisher>
  </book>
   ..
   ..
  <book id='4'>
    <author>Andreas Jung</author>
    <title>XML processing with Python</title>
    <publisher>Linux Magazine</publisher>
  </book>
</pythonbooks>
```

on the specific requirements in each case. SAX impresses with its speed and simplicity. More complex applications would suggest the use of DOM if they involve multiple access to many parts of an XML document or changes to the document's structure. The most important advantages and disadvantages are compared in Table 5. In part 2 of this article we will take a closer look at the use of XPath and XSLT under Python.

The author

Andreas Jung lives near Washington D.C. and works for Zope Corporation as part of the Zope core team. Email: *andreas@andreas-jung.com*

Info

Python XML: *http://pyxml.sourceforge.net*
C. A. Jones and F. L. Drake, Jr, *Python & XML* (O'Reilly, 2002)

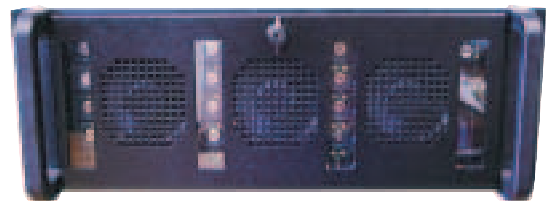
A NEW ERA IN NETWORK STORAGE

0.96TB = £3371   2.56TB = £9317

THE TERAVAULT STORAGE SERVERS from Digital Networks provide complete networked storage of up to 2622GB in size.

The Teravault 416S, pictured right, features hardware RAID storage with hot-swap capability, dual Intel Pentium III processors, up to 6.0GB of RAM and multiple Ethernet interfaces. Linux, UNIX, Windows and Apple clients are supported, and the system can be administered remotely with the included web based interface or by SSH.

From now on network attached storage needn't cost an arm and a leg. Multi-terabyte network storage from under £10,000. For full details, visit **www.dnuk.com.**



Digital Networks