**Parsing Using DOM**

The Document Object Model (DOM) allows you to model an XML document as a tree structure. In fact, the entire document is accessible as a series of objects, and by following the branches of the tree, you can traverse the entire document. Because we are representing the XML document in one piece, we can use DOM both to parse existing documents and to create new documents.

The only problem with using DOM is that it stores the entire document in memory. For the small documents we’re working with here, this won’t be a problem, but a 512KB document may require up to five times that amount when it’s stored internally as a DOM object. Of course, in Python we don’t have to worry about allocating the memory, but that also means that we run the risk of using large quantities of memory without realizing it.

Under Python the DOM interface is based on the IDL version of the specification released by W3C. The standard Python 2.x distribution comes with a basic DOM parsing system, called minidom, and a more complex pulldom system that extracts individual elements from a DOM tree without having to read the entire XML document into memory.

Because of Python’s flexible object system, it’s very easy to create an equivalent of the tree structure that an XML document mirrors within a Python object. Coupled with the easy object-handling features (especially when working with dictionaries and lists), we have a good platform for handling XML documents.

**Using *minidom***

To parse an existing XML document into a DOM object using minidom, you need to call either the parse() method, which accepts a filename or file object and processes the contents, or parseString(), which parses a bare string of information that you may have read separately from a file or network connection. In fact, it’s as easy as this:

from xml.dom.minidom import parse, parseString

# Parse a bare string as XML

stringdoc = parseString('<para>Some text</para>')

# Parse a file object

xmlfile1 = open('myfile.xml')

filedoc = parse(xmlfile1)

# Parse a file directly

filedoc = parse('myfile.xml')

Once you’ve converted the XML stream into a DOM object, you can then access the individual tags by name. For example, suppose that we’ve modeled a client’s bank accounts in XML, as shown in [Listing 11.5](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P302).

Listing 11.5: A Sample Account Record

<client>

<clientname>Martin Brown</clientname>

<account>

<accname>Checking</accname>

<provider>HSBC</provider>

<balance>$4567.00</balance>

<transaction>

<payee>Rent</payee>

<amount>$280.00</amount>

<freq>Monthly</freq>

</transaction>

<transaction>

<payee>Time Subscription</payee>

<amount>$26.00</amount>

<freq>Quarterly</freq>

</transaction>

</account>

<account>

<accname>VISA</accname>

<provider>Morgan Dean Stanley Witter</provider>

<balance>$-3485.00</balance>

<transaction>

<payee>Supermarket</payee>

<amount>$-450.00</amount>

</transaction>

<transaction>

<payee>Gas Station</payee>

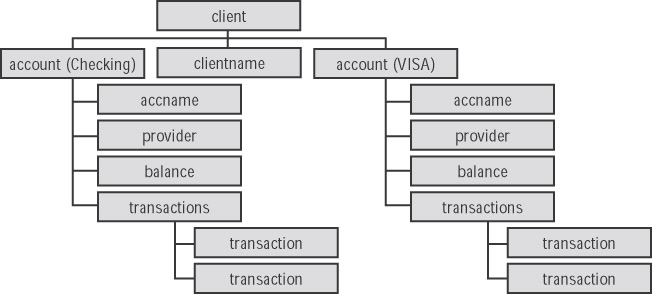
<amount>$-18.00</amount>

</transaction>

</account>

</client>

The document could be represented as a tree structure, as shown in [Figure 11.1](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P337). We’ll be using this diagram to help us understand how Python’s DOM implementation works.

  
Figure 11.1: An XML tree

We could get the name of the client who owns the account information using [Listing 11.6](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P339).

Listing 11.6: Extracting Content from an XML Document Using *minidom*

from xml.dom.minidom import parse

# Create a function to get the data between XML tags

# Information is held in nodes (discrete blocks)

# which we’ll need to concatenate together to get the

# full picture. We only need to add text nodes to the

# string

def getdata(nodes):

rc = ''

for node in nodes:

if node.nodeType == node.TEXT\_NODE:

rc = rc + node.data

return rc

# Parse the document

client = parse('client.xml')

# Get the first clientname tag from the document

clientname = client.getElementsByTagName("clientname")[0]

# Print out the data contained within the tags

# using getdata to extract the text from the nodes

# defined within the element

print 'Client name is', getdata(clientname.childNodes)

The getElementsByTagName() method returns a list of *all* the tag elements with the supplied name. The resulting objects contain the information about the tag, including any attributes if supplied, and a set of nodes that make up the data contained within the tags.

Note that the object returned by getElementsByTagName() is a branch (or leaf) of the tree structure shown in [Figure 11.1](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P337). The root of the tree is the first (root) tag within the document—so to access all the elements within the XML document, we’d have to access the client branch. From that base, we can then access the other elements. For example, to extract the data from the clientname branch, we must refer to the branch in reference to its parent, the client branch. Further branches and leaves are referenced in the same way, relative to their parent branches.

Had we used this:

accounts = client.getElementsByTagName("account")

The accounts object would now be a list containing the two account branches. Each element would refer to one of the account branches in our diagram. To get a list of the transactions within the checking account, we could have used this:

checking = accounts[0]

trans = checking.getElementsByTagName("transaction")

Now trans would contain the information in the two transactions in our account. Each element would be one of the transaction branches.

**DOM in Action**

To put all of this into practice, [Listing 11.7](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P374) is a script that uses DOM to generate a simple list of accounts and transactions for a given client. The script is actually a good example of a tree-based XML parser in Python. Because we’re not following the tree sequentially, we can be a little less restrictive about how we extract information: We don’t have to worry about recording states or determining whether the output format should change because we’ve reach a particular end tag.

Listing 11.7: Using *minidom* to Summarize an XML Document

from xml.dom.minidom import parse

def getdata(nodes):

rc = ''

for node in nodes:

if node.nodeType == node.TEXT\_NODE:

rc = rc + node.data

return rc

def handleclient(client):

clientname = client.getElementsByTagName("clientname")[0]

print 'Client:', getdata(clientname.childNodes)

accounts = client.getElementsByTagName("account")

handleaccounts(accounts)

def handleaccounts(accounts):

print 'Accounts:'

for account in accounts:

handleaccount(account)

def handleaccount(account):

accname = account.getElementsByTagName("accname")[0]

provider = account.getElementsByTagName("provider")[0]

print ' ' \* 4, '%s (%s)' % (getdata(accname.childNodes),

getdata(provider.childNodes))

print ' ' \* 4, 'Transactions:'

trans = account.getElementsByTagName("transaction")

for transaction in trans:

handletransaction(transaction)

balance = account.getElementsByTagName("balance")[0]

print ' ' \* 9, '%-40s %s' % ('', '======')

print ' ' \* 9, '%-40s %s' % ('', getdata(balance.childNodes))

print ''

def handletransaction(transaction):

payee = transaction.getElementsByTagName("payee")[0]

amount = transaction.getElementsByTagName("amount")[0]

print ' ' \* 9, '%-40s %s' % (getdata(payee.childNodes),

getdata(amount.childNodes))

client = parse('client.xml')

handleclient(client)

If we run this script on our client XML document, we get the following output:

$ python exdom2.py

Client: Martin Brown

Accounts:

Checking (HSBC)

Transactions:

Rent $280.00

Time Subscription $26.00

======

$4567.00

VISA (Morgan Dean Stanley Witter)

Transactions:

Supermarket $-450.00

Gas Station $-18.00

======

$-3485.00

We could have just as easily converted this document into HTML or XHTML or extracted the information easily for writing into the individual tables of a database.

**Building XML Documents with DOM**

You can write XML documents just by including the necessary print or similar statement in your script, but it relies on generating the tags in the correct order and structure and ensuring that they are matched up. Although this is not an impossible task, it does add extra levels of complexity to the process.

Simple serialization from one format into XML is easy if the information is in sequence and you convert directly into an XML document following the same structure. But what happens if you need to add new branches within the existing structure, or the definition of the XML document requires you to organize the information into a given structure that doesn’t match your source material?

The immediately obvious solution is either to separately model the incoming data into a more suitable format before translating it to XML or to cache information into one or more objects and dump them at appropriate times. Neither solution is infallible, and both are entirely reliant on getting the information correctly and in the order you expect in the first place.

A much better solution is at hand, though. The Document Object Model specification is really just a method for modeling XML documents within the confines of a programming language or other system. Up to now, we’ve used the system only to model an external XML document into an internal tree to extract information.

We can also use DOM to build an XML document by creating the branches and leaves of the document. Because DOM is not a sequential system such as SAX or Expat, we can add new branches and leaves to any part of the document without making modifications to the XML document in its raw text format.

The xml.dom.minidom module supports a very simple interface for adding new XML tags and data to an XML document. For a quick example, see [Listing 11.8](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=527929344#wbpch11P442), which adds both a text block and a tag to a previously parsed XML string.

Listing 11.8: Rebuilding an XML Document

from xml.dom.minidom import parseString

dom = parseString('<title></title>')

root = dom.documentElement

nelem = dom.createElement("separator")

root.insertBefore(nelem, None)

cdata = dom.createTextNode("The New Avengers")

root.insertBefore(cdata, nelem)

print root.toxml()

The start of the process is to create the equivalent of the blank root document as a DOM object by using the parseString() function to parse a string in XML format into an object.

Then we get the root of the document and create a new element, “separator.” The insertBefore() method in our document then inserts the element according to its reference location. In this case, we’re inserting the element with reference to None, which will insert the tag between the root title tags in our original XML string. The createTextNode() method creates a new block of text that we’ll insert before the element we just created.

Finally, the toxml() method returns the entire DOM structure as an XML string that we print out. Executing the script gives us a very simple document in return:

$ python dombuild.py

<title>The New Avengers<separator/></title>

Although this is simplistic, it demonstrates how easy it is to insert new tags and text data anywhere within a given DOM tree. The process of converting XML data to or from an alternative source will be a recurring topic. XML is not the ideal format for all situations, so we’ll be returning to the DOM system in Python in later chapters.