Python and XML: An Introduction

Abstract

The Python programming language provides an increasing amount of support for XML technologies. This document attempts to introduce some basic XML processing concepts to readers who have not yet started to use Python with XML, and it takes the form of a tutorial. It is assumed that the reader knows the basic terminology of XML and is comfortable with "XML as text".

Prerequisites

Python 2.0

I believe that Python release 2.0 or greater is most appropriate for XML processing - this is partly due to the introduction of Unicode support and the reasonably high probability that, for a number of readers, some XML documents encountered will use character sets which are not so easily supported using traditional Python character strings. I cannot really imagine how Python 1.5.2, for example, is able to handle "non-Western-European" characters, but given the length of time since the introduction of Python 2.0, as well as the familiarity the Python community has with the newer features of the releases from that time until the present, Python 2.0 is a safe and not unreasonable requirement.

PyXML 0.6.6

Some releases of Python come with a fair amount of built-in XML support, and the extent of this support can be tested by starting Python interactively and testing for the presence of the `minidom` module:

```
import xml.dom.minidom
```

Should this module be imported without complaints from the interpreter, it would appear that your Python version is fairly recent and probably good enough for the purposes of this tutorial. Otherwise, you should download the PyXML package, choosing release 0.6.6 or greater.

Activities

All the activities in this tutorial require the import of the `minidom` module. Therefore, for all of the program fragments featured, this module must have been made available through an import statement similar to the following:

```
import xml.dom.minidom
```

We could import just the classes we need from the module, but we can leave our options open at this point. Entering the following statement gives us an idea of the contents of the module:

```
dir(xml.dom.minidom)
```

Namespaces

Before we start to experiment, here is a note about namespaces. It is very tempting not to use namespaces when starting out with XML documents and XML processing, but namespaces provide an interesting way of associating XML elements with certain meanings, applications and "domains". Rather than cause confusion later by introducing namespaces after the basic operations have been presented, I believe that they are easy enough to use at the start not to cause confusion at all.

Creating an XML Document

First, import the `minidom` module:

```
import xml.dom.minidom
```

To create a new XML document, just instantiate a new `Document` object:

```
def get_a_document():
    doc = xml.dom.minidom.Document()
```

This document is not really interesting without some contents, however, so we should add something to it.

Elements

XML documents have a single "root element" inside which all other elements and pieces of text are placed. We could create an XML document which describes departments in a company with a "root element" called "business" within the namespace "http://www.boddie.org.uk/paul/business" - we want to communicate the fact that the "business" element means something special to us, and that the use of our namespace indicates that it is our special "business" element rather than any old home-made "business" element; the following statement creates such an element:

```
    business_element = doc.createElementNS("http://www.boddie.org.uk/paul/business", "business")
```

At this point, the element exists but has not been placed in the document; we need to add it to the document at the "root level" as follows:

```
    doc.appendChild(business_element)
```

Let us return the created objects at the end of this function:

```
    return doc, business_element
```

Now, the "root element" has been added - we can investigate this by querying the document about the elements within it. At the Python prompt...

```
>>> import xhtmlhook # to read this document
>>> from XML_intro.Creating import *
>>> doc, business_element = get_a_document()
```

```
>>> doc.childNodes
```

```
[<DOM Element: business at 136860108>]
```

This shows a list of elements with only one element present within it. We can, of course, examine the list and the element more closely:

```
>>> doc.childNodes[0].namespaceURI
```

```
'http://www.boddie.org.uk/paul/business'
```

This was the namespace that we gave our element.

```
>>> doc.childNodes[0].localName
```

```
'business'
```

This was the element name that we used.

We can add elements within this one. For example, a "location" element might be interesting to describe a particular location of part of a company, and such an element could be created in a similar way to that used above:

```
def add_a_location(doc, business_element):
    location_element = doc.createElementNS("http://www.boddie.org.uk/paul/business", "location")
```

We add the element as a child node of the old element in the same fashion as before:

```
    business_element.appendChild(location_element)
```

Let us return the created object at the end of this function:

```
    return location_element
```

At this point, it is possible to navigate down from the "root" of the document to the newly added element:

```
>>> location_element = add_a_location(doc, business_element)
```

```
>>> doc.childNodes[0].childNodes
```

```
[<DOM Element: location at 136781996>]
```

```
>>> doc.childNodes[0].childNodes[0]
```

```
<DOM Element: location at 136781996>
```

```
>>> doc.childNodes[0].childNodes[0].namespaceURI
```

```
'http://www.boddie.org.uk/paul/business'
```

```
>>> doc.childNodes[0].childNodes[0].localName
```

```
'location'
```

Text

Text is a central feature of XML documents - inside elements, blocks of text may be stored and retrieved. We might have, in our example, an element which is called "surroundings", and this could be found within the "location" element as a means of describing the surroundings of a particular company location. Inside the "surroundings" element, there could be a block of text which forms such a description.

Here, we create the new "surroundings" element and add it within the "location" element:

```
def add_surroundings(doc, location_element):
    surroundings_element = doc.createElementNS("http://www.boddie.org.uk/paul/business", "surroundings")
    location_element.appendChild(surroundings_element)
```

And we specify the descriptive text within this new element by creating a new text node:

```
    description = doc.createTextNode("A quiet, scenic park with lots of wildlife.")
```

Of course, we need to add this to the document, and since the text is to be included within the "surroundings" element, it makes sense to add it to that element as a child node:

```
    surroundings_element.appendChild(description)
```

Let us return the created object from this function:

```
    return surroundings_element
```

We may now find our way from the document "root", if we want to:

```
>>> surroundings_element = add_surroundings(doc, location_element)
```

```
>>> doc.childNodes[0].childNodes[0].childNodes[0].childNodes[0]
```

```
<DOM Text node "A quiet, s...">
```

It is possible to see the entire contents of the text node using the `nodeValue` attribute of the node:

```
>>> doc.childNodes[0].childNodes[0].childNodes[0].childNodes[0].nodeValue
```

```
'A quiet, scenic park with lots of wildlife.'
```

We can, as with elements (as we shall see in a moment), add many text nodes within an element:

```
def add_more_surroundings(doc, surroundings_element):
    description = doc.createTextNode(" It's usually sunny here, too.")
    surroundings_element.appendChild(description)
```

Here is the "proof" that this worked:

```
>>> add_more_surroundings(doc, surroundings_element)
```

```
>>> surroundings_element.childNodes
```

```
[<DOM Text node "A quiet, s...">, <DOM Text node " It's usual...">]
```

We might want all our text within one node in future, however. Fortunately, a method exists to collect text nodes together (note the "-ize" spelling):

```
def fix_element(element):
    element.normalize()
```

The results can be investigated, too:

```
>>> fix_element(surroundings_element)
```

```
>>> surroundings_element.childNodes[0].nodeValue
```

```
"A quiet, scenic park with lots of wildlife. It's usually sunny here, too."
```

Attributes

Elements in XML documents may have attributes attached to them. For example, the "location" element could have another element within it (alongside the "surroundings" element) entitled "building", and this new element could have an attribute called "name":

```
def add_building(doc, location_element):
    building_element = doc.createElementNS("http://www.boddie.org.uk/paul/business", "building")
```

After adding this element...

```
    location_element.appendChild(building_element)
    return building_element
```

...it should be noticed that the new element appears after the "surroundings" element as a child element (or node) of "location". This can be seen at the Python prompt as follows:

```
>>> building_element = add_building(doc, location_element)
```

```
>>> location_element.childNodes
```

```
[<DOM Element: surroundings at 136727844>, <DOM Element: building at 136286548>]
```

Now, we may add an attribute directly to the new element like this:

```
def name_building(building_element):
    building_element.setAttributeNS("http://www.boddie.org.uk/paul/business", "business:name", 'Ivory To
```

After the namespace and the element name, the value is specified. This attribute does not need to be explicitly added to the element, although we could have used other means of creating and adding it. This can be tested as follows:

```
>>> name_building(building_element)
```

```
>>> building_element.getAttributeNS("http://www.boddie.org.uk/paul/business", "name")
```

```
'Ivory Tower'
```

One important thing to note is the use of the "qualified name" when setting the attribute and the "local name" when getting the attribute value.

Attribute Namespaces and Prefixes

One might expect that by setting an attribute with a particular namespace, there would not be any need to explicitly state a prefix to appear before the local name in the final, written XML document - after all, it should be possible to detect that a namespace has been employed and that a prefix is required in the full attribute name so that the attribute is recognised as being associated with that namespace. Unfortunately, we do not seem to have that luxury and must explicitly specify a prefix as part of the qualified name when setting such an attribute; in the above example, the qualified name consists of the prefix "business" and local name "name".

Writing an XML Document

All the above effort should not be wasted, and so we will attempt to write the document out. One of the easiest ways of doing this, whilst respecting the namespaces that we have carefully included, is to use another module (along with the `minidom`) module:

```
import xml.dom.ext
import xml.dom.minidom
```

Inside this module are a number of useful functions and classes. However, we shall use the `PrettyPrint` function to write the document to "standard output", ie. the screen:

```
def write_to_screen(doc):
    xml.dom.ext.PrettyPrint(doc)
```

We can then try this out:

```
>>> from XML_intro.Writing import *
>>> write_to_screen(doc)
```

```
<?xml version='1.0' encoding='UTF-8'?>
<business xmlns='http://www.boddie.org.uk/paul/business' xmlns:business='http://www.boddie.org.uk/paul/b
  <location>
    <surroundings>A quiet, scenic park with lots of wildlife.</surroundings>
    <building business:name='Ivory Tower'/>
  </location>
</business>
```

We could have used another, simpler printing function or class, but we are usually so accustomed to (or spoilt by) nicely formatted textual XML documents that anything less than `PrettyPrint` probably would not do! Especially since `PrettyPrint` allows us to write the document to a file:

```
def write_to_file(doc, name="/tmp/doc.xml"):
    file_object = open(name, "w")
    xml.dom.ext.PrettyPrint(doc, file_object)
    file_object.close()
```

Or even easier:

```
def write_to_file_easier(doc, name="/tmp/doc.xml"):
    xml.dom.ext.PrettyPrint(doc, open(name, "w"))
```

NOTE: Should we not use "wb" for portability reasons?

```
>>> write_to_file(doc)
```

Reading an XML Document

XML would not be very useful if we could not read it back later for subsequent processing. Fortunately, `minidom` makes this fairly easy to achieve:

```
import xml.dom.minidom
def get_a_document(name="/tmp/doc.xml"):
    return xml.dom.minidom.parse(name)
```

Or, if a file is already open for the purposes of reading...

```
def get_a_document_from_file(file_object):
```

```
    return xml.dom.minidom.parse(file_object)
```

Textual "Interference"

Unfortunately, if a file is written out in a prettyprinted fashion, it gets read in with all the extra padding, and this padding appears as text nodes between elements where we never inserted any kind of text nodes before. For example:

```
>>> from XML_intro.Reading import *
>>> doc2 = get_a_document()
```

```
>>> doc2.childNodes[0].childNodes[0]
```

```
<DOM Text node "\012">
```

What is this new text node? (We expected to see the "location" element's DOM object here instead.) Well, it appears to be a newline character which was inserted into our file to make the contents of that file look good when viewed in a text editor, for example. There are more of these things, too:

```
>>> doc2.childNodes[0].childNodes[1]
```

```
<DOM Text node "  ">
```

This text node is a piece of indentation - something which made the textual form of the "location" element appear slightly further right than the "left margin" used by the "business" element. If we keep looking, though, we can find the "location" element:

```
>>> doc2.childNodes[0].childNodes[2]
```

```
<DOM Element: location at 137096548>
```

So how do we avoid referencing the wrong nodes in the document?

1. Instead of assuming that any element is at a particular child node position, loop over the child nodes until the appropriate node is found.
2. Tell the parser (which reads the document from the file) not to include all the padding.
3. Use convenience functions to get down to the part of the document which interests us.
4. Use XPath querying to select the most interesting nodes.

Here is a brief description of each of the above options:

Node Iterators

What we could do in the above case is to loop over the child nodes and examine each node to see what type it is. If it is an element then we investigate it, starting with "business" just to be safe:

```
def find_business_element(doc):
    business_element = None
    for e in doc.childNodes:
        if e.nodeType == e.ELEMENT_NODE and e.localName == "business":
            business_element = e
            break
    return business_element
```

We know that if `business_element` is not `None`, then we found an element called "business". Naturally, we can change the name of the element to be found according to the situation. Note that we compare the value of the

nodeType attribute to a special attribute called ELEMENT_NODE. It is not obvious from this example, but special attributes such as ELEMENT_NODE and TEXT_NODE actually belong to the xml.dom.minidom.Node class, and are therefore available in (or shared by) all node objects.

Parser Option Changes

NOTE: This should be documented somewhere. I haven't investigated it yet.

Convenience Functions

It should possble to show the "surroundings" element's textual contents just by searching for the "surroundings" element, finding its child nodes, and then getting their values. Here, we use the getElementsByTagNameNS method on the document object to find all occurrences of the "surroundings" element in the document:

```
def get_surroundings_elements(doc):
    return doc.getElementsByTagNameNS("http://www.boddie.org.uk/paul/business", "surroundings")
```

The result can then be investigated:

```
>>> elements = get_surroundings_elements(doc2)
```

```
>>> elements
```

```
[<DOM Element: surroundings at 137101100>]
```

So, the contents of the element can indeed be discovered, too:

```
>>> elements[0]
```

```
<DOM Element: surroundings at 137101100>
```

```
>>> elements[0].childNodes
```

```
[<DOM Text node "A quiet, s...">]
```

```
>>> elements[0].childNodes[0].nodeValue
```

```
u"A quiet, scenic park with lots of wildlife. It's usually sunny here, too."
```

Note that the description that appears as the value of the text node is actually a Unicode string, but this should not concern you too much - Unicode strings are widely accepted by minidom and behave a lot like "traditional" Python strings.

Now, there are a number of problems with the above approach:

- What if there are many "location" elements? We only specified one, but a company might have many locations.
- What if more than one "surroundings" element is allowed in at a "location"? It would be a strange concept, but it would confuse the above approach, too.

The principal problem with using such convenience functions is that any structure or context that an element may have, which is taken into consideration by descending into the document yourself, is lost when the results of the function are returned: all "surroundings" elements are bundled together into one package and handed back. However, we can learn about the context of the elements by exploring various useful attributes of those elements.

Every "surroundings" element should have a "location" element as its parent. We can check this at the Python prompt:

```
>>> elements[0].parentNode
```

```
<DOM Element: location at 137096548>
```

This at least allows us to compare "location" nodes against each other using something like this:

```
def examine_descriptions(elements):
    if elements[0].parentNode is elements[1].parentNode:
        print "They both describe the same location."
```