Python XML Element Trees

Adrian P. Robson

22 October 2015

Here we describe solutions to some problems that might be encountered working with the `xml.etree.ElementTree` package.

1 Sorting

There is no completely generic solution for sorting XML trees because possible XML schema are too diverse.[1] However there are particular solutions, and as an example we will sort an XML address book that looks like this:

```
<book>
   <address name="John Smith" sort="smithjohn">
      <tel>2222</tel>
   </address>
   <address name="Joe Brown" sort="brownjoe">
      <tel>4444</tel>
   </address>
</book>
```

The following function shows how this XML data can be sorted by the `sort` attribute in its `address` element. Its single parameter is the root of the tree to be sorted; and the function returns a sorted copy of this tree. The original tree is unchanged.

```
1  import xml.etree.ElementTree as ET
2  ...
3  def sortXmlAddrs( book ):
4
5      def getkey(address):
```

[1]This is probably the reason why the element tree package does not provided a sort facility.

```
6            return address.get("sort").upper()
7
8        roottag = book.tag         # save root fields
9        roottext = book.text
10       roottail = book.tail
11       stree =  ET.Element(None)  # dummy tree
12       stree[:] = sorted(book,key=getkey)
13       stree.tag = roottag        # restore root fields
14       stree.text = roottext
15       stree.tail = roottail
16       return stree
```

The actual sort is performed at line 12. The assignment at line 11 creates a variable of the correct type, and the use of`[:]` in line 12 preserves this type after the assignment of the sorted structure.[2] A side-effect of the sort operation is the removal of the root element's `tag`, `text`, and `tail` fields. Lines 8-10 and 13-16 correct this problem.

Unfortunately, sorting the address book is not completely successful because the operation does not correctly manage the indentation used in the example. This is a general problem with sorting the contents of an XML element that is of mixed complex type. In this case, it can be fixed by applying the `indent` function given in §3 to the sorted tree.

The sort function given above might be used like this:

```
try:
    book = ET.parse("test.xml").getroot()
except Exception as x:
   print("XML parse failed because {}".format(x))
   return
sbook = sortXmlAddrs(book)
```

2 Copying XML Trees

Here we distinguish between *shallow* and *deep* copies of tree or list data structures. A shallow copy assigns a reference to the original data structure; and after the copy, modifying the the new variable also changes the original. A deep copy produces a distinct clone of the original that is completely independent from the original.

Examples of shallow cop[y are simple assignment and `ElementTree` creation: In Python an assignment creates a reference to the original data:

[2]This Python alchemy is needed because of the language's dynamic type system.

```
book = ET.parse("book.xml")
nbook = book.getroot()    # shallow copy!
```

XML trees are Python lists, and this means that the variables `book` and `nbook` are references to the same XML tree, and a modification to one will thus also change the other. Creating a new `ElementTree` object might imply that we are getting a completely new tree, but it actually produces a shallow copy:

```
nbook = ET.ElementTree(book.getroot())   # shallow copy!
```

Again `book` and `nbook` are references to the same XML tree.

A shallow copy is often inadequate, and we must have a deep copy of a XML tree. There are a couple of easy ways to do this: We can use strings, or the `copy` package. Converting the XML tree to a string and then back to a tree produces a deep copy:

```
nbook = ET.fromstring(ET.tostring(book.getroot()))
```

Now we get separate data structures, and a change to `nbook` will not affect `book`.

Perhaps the simplest way to get a deep copy is the use the `copy` package like this:

```
from copy import deepcopy
...
book = ET.parse("book.xml")
nbook=deepcopy(book.getroot())
```

3 Indenting

The following 'pretty print' function fills XML element `text` and `tail` fields with appropriate newlines and spaces. It is recursive:

```
def indent( elem, level=0 ):

    INDENT = "   "

    def sindent( elem, indents ):
        if not elem.text or not elem.text.strip():
            elem.text = "\n" + INDENT * indents

```

```
9       def eindent( elem, indents ):
10          if not elem.tail or not elem.tail.strip():
11              elem.tail =  "\n" + INDENT * indents
12
13      if len(elem):
14          sindent(elem,level+1)
15          eindent(elem,level)
16          for sube in elem:
17              indent(sube,level+1)
18          eindent(sube,level)
19      else:
20          eindent(elem,level)
```

The nested functions at lines 5-6 and 9-11 are responsible for inserting indention and line breaks in `text` and `tail` fields. Their if statements ensure that only white-space can be overwritten.

The test at line 13 determines if the current element is a branch node with sub-elements, or a leaf. Lines 16 and 17 recursively calls the `indent` function for each sub-element.

The function calls at lines 14 and 15 is apply indentation to branch nodes (head recursion); and line 20 does it to leaf nodes (recursion end). So all sub-elements get the the same text and tail indention. Unfortunately, this *applies the wrong value* to the last sub-element's `tail` field (!); but a correction is performed at line 18 when the indent function returns to its caller (tail recursion). When the for loop of lines 16 and 17 ends, the 'target' variable `sube` still exists[3], and is actually the last sub-element of the current branch node. So `sube` is used in line 18 to replace the erroneous `tail` field value.

The above function given is similar to code given at Stack Overflow [1], where it is stated that it is part of the Python 2.5 and later `ElementTree` library; and at effbot.org [2], where it is suggested that the function may [sic] be part of an `ElementLib` module. However, there is no evidence of such a function in Python 3.5 code (!).

4 Deleting Attributes

The standard Python documentation does not mention a process for deleting XML element attributes, but an element's attributes can be obtained as a Python dictionary. This dictionary can then be modified to remove a key, but

[3]This is standard python behaviour, although it it not allowed is some other programming languages.

it is not clearly stated in the documentation that such a dictionary update will result in an actual XML element tree change.

In practice (with Python 3.3) such a dictionary update works, and results in an element update as required. The following code gives an example of how this might be handled:

```
    if myelement.get("akey1"):
            del myelement.attrib["akey1"]
```

Python documentation advises that `get` and `set` should be be used in preference to direct dictionary operations for accessing, adding or modifying attributes.

References

[1] *Pretty printing XML in Python*, Stack Overflow, Answered by Ade, 2011. `http://stackoverflow.com/questions/749796/pretty-printing-xml-in-python`

[2] Fredrik Lundh, *Element Library Functions*, effbot.org, 2004. `http://effbot.org/zone/element-lib.htm#prettyprint`