Processing XML in Python with ElementTree

March 15, 2012 at 06:14 **Tags** Articles , Python

When it comes to parsing and manipulating XML, Python lives true to its "batteries included" motto. Looking at the sheer amount of modules and tools it makes available out of the box in the standard library can be somewhat overwhelming for programmers new to Python and/or XML.

A few months ago an interesting discussion took place amongst the Python core developers regarding the relative merits of the XML tools Python makes available, and how to best present them to users. This article (and hopefully a couple of others that will follow) is my humble contribution, in which I plan to provide my view on which tool should be preferred and why, as well as present a friendly tutorial on how to use it.

The code in this article is demonstrated using Python 2.7; it can be adapted for Python 3.x with very few modifications.

Which XML library to use?

Python has quite a few tools available in the standard library to handle XML. In this section I want to give a quick overview of the packages Python offers and explain why `ElementTree` is almost certainly the one you want to use.

`xml.dom.*` modules - implement the W3C DOM API. If you're used to working with the DOM API or have some requirement to do so, this package can help you. Note that there are several modules in the `xml.dom` package, representing different tradeoffs between performance and expressivity.

`xml.sax.*` modules - implement the SAX API, which trades convenience for speed and memory consumption. SAX is an event-based API meant to parse huge documents "on the fly" without loading them wholly into memory [1].

`xml.parser.expat` - a direct, low level API to the C-based `expat` parser [2]. The `expat` interface is based on event callbacks, similarly to SAX. But unlike SAX, the interface is non-standard and specific to the `expat` library.

Finally, there's `xml.etree.ElementTree` (from now on, ET in short). It provides a lightweight Pythonic API, backed by an efficient C implementation, for parsing and creating XML. Compared to DOM, ET is much faster [3] and has a more pleasant API to work with. Compared to SAX, there is `ET.iterparse` which also provides "on the fly" parsing without loading the whole document into memory. The performance is on par with SAX, but the API is higher level and much more convenient to use; it will be demonstrated later in the article.

with SAX, but the API is higher level and much more convenient to use; it will be demonstrated later in the article.

My recommendation is to always use ET for XML processing in Python, unless you have very specific needs that may call for the other solutions.

ElementTree - one API, two implementations

`ElementTree` is an API for manipulating XML, and it has two implementations in the Python standard library. One is a pure Python implementation in `xml.etree.ElementTree`, and the other is an accelerated C implementation in `xml.etree.cElementTree`. It's important to remember to *always* use the C implementation, since it is much, much faster and consumes significantly less memory. If your code can run on platforms that might not have the `_elementtree` extension module available [4], the import incantation you need is:

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
```

This is a common practice in Python to choose from several implementations of the same API. Although chances are that you'll be able to get away with just importing the first module, your code *may* end up running on some strange platform where this will fail, so you better prepare for the possibility. Note that starting with Python 3.3, this will no longer be needed, since the `ElementTree` module will look for the C accelerator itself and fall back on the Python implementation if that's not available. So it will be sufficient to just `import xml.etree.ElementTree`. But until 3.3 is out and your code runs on it, just use the two-stage import presented above.

Anyway, wherever this article mentions the `ElementTree` module (ET), I mean the C implementation of the `ElementTree` API.

Parsing XML into a tree

Let's start with the basics. XML is an inherently hierarchical data format, and the most natural way to represent it is with a tree. ET has two objects for this purpose - `ElementTree` represents the whole XML document as a tree, and `Element` represents a single node in this tree. Interactions with the whole document (reading, writing, finding interesting elements) are usually done on the `ElementTree` level. Interactions with a single XML element and its sub-elements is done on the `Element` level. The following examples will demonstrate the main uses [5].

We're going to use the following XML document for the sample code:

```
<?xml version="1.0"?>
<doc>
    <branch name="testing" hash="1cdf045c">
        text,source
    </branch>
    <branch name="release01" hash="f200013e">
        <sub-branch name="subrelease01">
            xml,sgml
        </sub-branch>
    </branch>
    <branch name="invalid">
    </branch>
    </branch>
    <branch name="invalid">
    </branch>
</doc>
```

Let's load and parse the document:

```
>>> import xml.etree.cElementTree as ET
>>> tree = ET.ElementTree(file='doc1.xml')
```

Now let's fetch the root element:

```
>>> tree.getroot()
<Element 'doc' at 0x11eb780>
```

As expected, the root is an `Element` object. We can examine some of its attributes:

```
>>> root = tree.getroot()
>>> root.tag, root.attrib
('doc', {})
```

True, the root element has no attributes [6]. As any `Element`, it presents an iterable interface for going over its direct children:

```
>>> for child_of_root in root:
...   print child_of_root.tag, child_of_root.attrib
...
branch {'hash': '1cdf045c', 'name': 'testing'}
branch {'hash': 'f200013e', 'name': 'release01'}
branch {'name': 'invalid'}
```

We can also access a specific child, by index:

```
>>> root[0].tag, root[0].text
('branch', '\n        text,source\n    ')
```

Finding interesting elements

From the examples above it's obvious how we can reach all the elements in the XML document and query them, with a simple recursive procedure (for each element, recursively visit all its children). However, since this can be a common task, ET presents some useful tools for simplifying it.

The `Element` object has an `iter` method that provices depth-first iteration (DFS) over all the sub-elements below it. The `ElementTree` object also has the `iter` method as a convenience, calling the root's `iter`. Here's the simplest way to find all the elements in the document:

```
>>> for elem in tree.iter():
...   print elem.tag, elem.attrib
...
doc {}
branch {'hash': '1cdf045c', 'name': 'testing'}
branch {'hash': 'f200013e', 'name': 'release01'}
sub-branch {'name': 'subrelease01'}
branch {'name': 'invalid'}
```

This could naturally serve as a basis for arbitrary iteration of the tree - go over all elements, examine

This could naturally serve as a basis for arbitrary iteration of the tree - go over all elements, examine those with interesting properties. ET can make this task more convenient and efficient, however. For this purpose, the `iter` method accepts a tag name, and iterates only over those elements that have the required tag:

```
>>> for elem in tree.iter(tag='branch'):
...   print elem.tag, elem.attrib
...
branch {'hash': '1cdf045c', 'name': 'testing'}
branch {'hash': 'f200013e', 'name': 'release01'}
branch {'name': 'invalid'}
```

XPath support for finding elements

A much more powerful way for finding interesting elements with ET is by using its XPath support. `Element` has some "find" methods that can accept an XPath path as an argument. `find` returns the first matching sub-element, `findall` all the matching sub-elements in a list and `iterfind` provides an iterator for all the matching elements. These methods also exist on `ElementTree`, beginning the search on the root element.

Here's an example for our document:

```
>>> for elem in tree.iterfind('branch/sub-branch'):
...   print elem.tag, elem.attrib
...
sub-branch {'name': 'subrelease01'}
```

It found all the elements in the tree tagged `sub-branch` that are below an element called `branch`. And here's how to find all `branch` elements with a specific `name` attribute:

```
>>> for elem in tree.iterfind('branch[@name="release01"]'):
...   print elem.tag, elem.attrib
...
branch {'hash': 'f200013e', 'name': 'release01'}
```

To study the XPath syntax ET supports, see this page.

Building XML documents

ET provides a simple way to build XML documents and write them to files. The `ElementTree` object has the `write` method for this purpose.

Now, there are probably two main use scenarios for writing XML documents. You either read one, modify it, and write it back, or create a new document from scratch.

Modifying documents can be done by means of manipulating `Element` objects. Here's a simple example:

```
>>> root = tree.getroot()
>>> del root[2]
>>> root[0].set('foo', 'bar')
>>> for subelem in root:
...   print subelem.tag, subelem.attrib
...
branch {'foo': 'bar', 'hash': '1cdf045c', 'name': 'testing'}
```

```
...
branch {'foo': 'bar', 'hash': '1cdf045c', 'name': 'testing'}
branch {'hash': 'f200013e', 'name': 'release01'}
```

What we did here is delete the 3rd child of the root element, and add a new attribute to the first child. The tree can then be written back into a file. Here's how the result would look:

```
>>> import sys
>>> tree.write(sys.stdout)   # ET.dump can also serve this purpose
<doc>
    <branch foo="bar" hash="1cdf045c" name="testing">
        text,source
    </branch>
    <branch hash="f200013e" name="release01">
        <sub-branch name="subrelease01">
            xml,sgml
        </sub-branch>
    </branch>
    </doc>
```

Note that the order of the attributes is different than in the original document. This is because ET keeps attributes in a dictionary, which is an unordered collection. Semantically, XML doesn't care about the order of attributes.

Building whole new elements is easy too. The ET module provides the `SubElement` factory function to simplify the process:

```
>>> a = ET.Element('elem')
>>> c = ET.SubElement(a, 'child1')
>>> c.text = "some text"
>>> d = ET.SubElement(a, 'child2')
>>> b = ET.Element('elem_b')
>>> root = ET.Element('root')
>>> root.extend((a, b))
>>> tree = ET.ElementTree(root)
>>> tree.write(sys.stdout)
<root><elem><child1>some text</child1><child2 /></elem><elem_b /></root>
```

XML stream parsing with iterparse

As I mentioned in the beginning of this article, XML documents tend to get huge and libraries that read them wholly into memory may have a problem when parsing such documents is required. This is one of the reasons to use the SAX API as an alternative to DOM.

We've just learned how to use ET to easily read XML into a in-memory tree and manipulate it. But doesn't it suffer from the same memory hogging problem as DOM when parsing huge documents? Yes, it does. This is why the package provides a special tool for SAX-like, on the fly parsing of XML. This tool is `iterparse`.

I will now use a complete example to demonstrate both how `iterparse` may be used, and also measure how it fares against standard tree parsing. I'm auto-generating an XML document to work with. Here's a tiny portion from its beginning:

```
<?xml version="1.0" standalone="yes"?>
<site>
  <regions>
```

```
<?xml version="1.0" standalone="yes"?>
<site>
  <regions>
    <africa>
      <item id="item0">
        <location>United States</location>    <!-- Counting locations -->
        <quantity>1</quantity>
        <name>duteous nine eighteen </name>
        <payment>Creditcard</payment>
        <description>
          <parlist>
[...]
```

I've emphasized the element I'm going to refer to in the example with a comment. We'll see a simple script that counts how many such `location` elements there are in the document with the text value "Zimbabwe". Here's a standard approach using `ET.parse`:

```
tree = ET.parse(sys.argv[2])

count = 0
for elem in tree.iter(tag='location'):
    if elem.text == 'Zimbabwe':
        count += 1

print count
```

All elements in the XML tree are examined for the desired characteristic. When invoked on a ~100MB XML file, the peak memory usage of the Python process running this script is ~560MB and it takes 2.9 seconds to run.

Note that we don't really need the whole tree in memory for this task. It would suffice to just detect `location` elements with the desired value. All the other data can be discarded. This is where `iterparse` comes in:

```
count = 0
for event, elem in ET.iterparse(sys.argv[2]):
    if event == 'end':
        if elem.tag == 'location' and elem.text == 'Zimbabwe':
            count += 1
    elem.clear() # discard the element

print count
```

The loop iterates over `iterparse` events, detecting "end" events for the `location` tag, looking for the desired value. The call to `elem.clear()` is key here - `iterparse` still builds a tree, doing it on the fly. Clearing the element effectively discards the tree [7], freeing the allocated memory.

When invoked on the same file, the peak memory usage of this script is just 7MB, and it takes 2.5 seconds to run. The speed improvement is due to our traversing the tree only once here, while it is being constructed. The `parse` approach builds the whole tree first, and then traverses it again to look for interesting elements.

The performance of `iterparse` is comparable to SAX, but its API is much more useful - since it builds the tree on the fly for you; SAX just gives you the events and you build the tree yourself.

Conclusion

Conclusion

Of the many modules Python offers for processing XML, `ElementTree` really stands out. It combines a lightweight, Pythonic API with excellent performance through its C accelerator module. All things considered, it almost never makes sense *not* to use it if you need to parse or produce XML in Python.

This article presents a basic tutorial for ET. I hope it will provide anyone with interest in the subject enough material to start using the module and explore its more advanced capabilities on their own.

[1] As opposed to DOM, which loads the whole document into memory and allows "random access" to its elements at any depth.

[2] expat is an open-source C library for parsing XML. Python carries it around in its distribution, and it serves as the base of Python's XML parsing capabilities.

[3] Fredrik Lundh, the original author of `ElementTree`, collected some benchmarks here. Scroll down on that page to see them.

[4] When I mention `_elementtree` in this article, I mean the C accelerator that's used for `cElementTree`. `_elementtree` is a C extension module for Python that is part of the standard distribution.

[5] Be sure to have the module documentation handy and look up the methods I'm calling to better understand the parameters passed.

[6] *Attributes* is an overloaded term here. There are Python object attributes, and there are XML element attributes. Hopefully it will be obvious from the context which one is implied.

[7] To be precise, the root element of the tree is still alive. In the unlikely event that the root element is very large, you can discard it too, but that would need a bit more code.